show framing

Previous Next

4.1 Socket Buffers

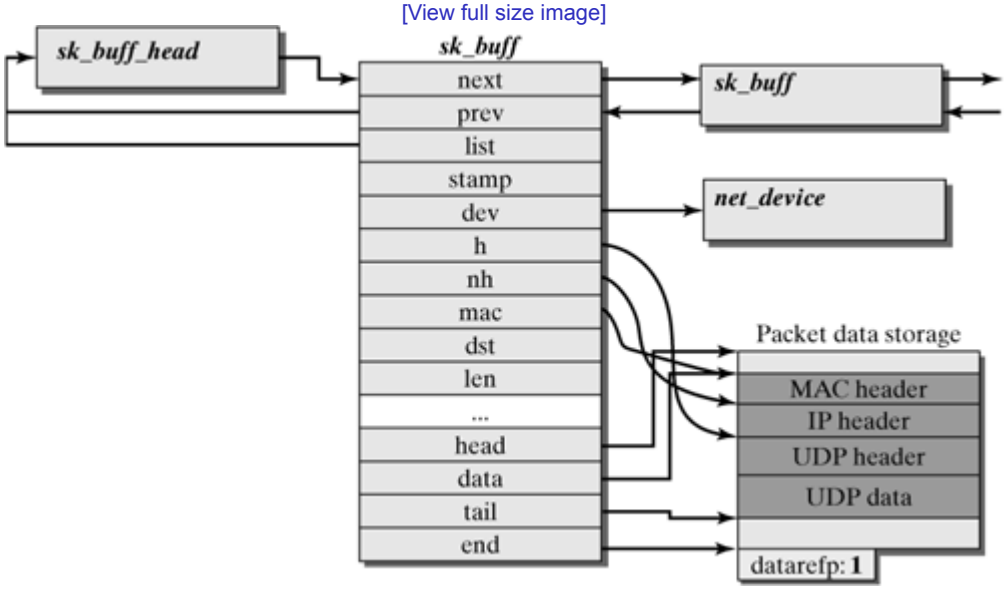
The network implementation of Linux is designed to be independent of a specific protocol. This applies both to the network and transport layer protocols (TCIP/IP, IPX/SPX, etc.) and to network adapter protocols (Ethernet, token ring, etc.). Other protocols can be added to any network layer without a need for major changes. As mentioned before, socket buffers are data structures used to represent and manage packets in the Linux kernel.

A socket buffer consists of two parts (shown in Figure 4-1):

- Packet data: This storage location stores data actually transmitted over a network. In the terminology introduced in Section 3.2.1, this storage location corresponds to the protocol data unit.
- Management data (`struct sk_buff`): While a packet is being processed in the Linux kernel, the kernel requires additional data that are not necessarily stored in the actual packet. These mainly implementation-specific data (pointers, timers, etc.). They form part of the interface control information (ICI) exchanged between protocol instances, in addition to the parameters passed in function calls.

Figure 4-1. Structure of socket buffers (`struct sk_buff`) with packet storage locations.

[View full size image]

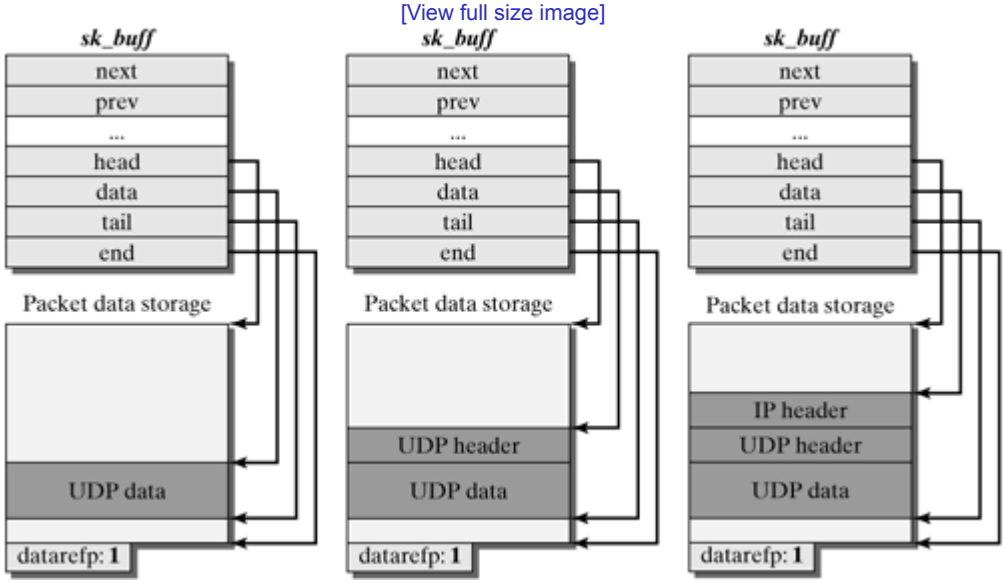


The socket buffer is the structure used to address and manage a packet over the entire time this packet is being processed in the kernel. When an application passes data to a socket, then the socket creates an appropriate socket buffer structure and stores the payload data address in the variables of this structure. During its travel across the layers (see Figure 4-2), packet headers of each layer are inserted

in front of the payload. Sufficient space is reserved for packet headers that multiple copying of the payload behind the packet headers is avoided (in contrast to other operating systems). The payload is copied only twice: once when it transits from the user address space to the kernel address space, and a second time when the packet data is passed to the network adapter. The free storage space in front of the currently valid packet data is called headroom, and the storage space behind the current packet data is called tailroom in Linux.

Figure 4-2. Changes to the packet buffers across the protocol hierarchy.

[View full size image]



When a packet is received over a network adapter, the method `dev_alloc_skb()` is used to request an `sk_buff` structure during the interrupt handling. This structure is then used to store the data from the received packet. Until it is sent, the packet is always addressed over the socket buffer created.

We now explain briefly the parameters of the `sk_buff` structure (Figure 4-3):

- `next, prev` are used to concatenate socket buffers in queues (`struct skb_queue_head`). They should always be provided by special functions available to process socket buffers (`skb_queue_head(), skb_dequeue_tail()`, etc.) and should not be changed directly by programmers. These operations will be introduced in Section 4.1.1.
- `list` points to the queue where the socket buffer is currently located. For this reason, queues should always be of the type `struct sk_buff_head`, so that they can be managed by socket buffer operations. This pointer should point to `null` for a packet not assigned to any queue.
- `sk` points to the socket that created the packet. For a software router, the driver of the network adapters creates the socket buffer structure. This means that the packet is not assigned to a valid socket, and so the pointer points to `null`.
- `stamp` specifies the time when the packet arrived in the Linux system (in `jiffies`).
- `dev` and `rx_dev` are references to network devices, where `dev` states the current network device on which the socket buffer currently operates. Once the routing decision has been taken, `dev`

points to the network adapter over which the packet should leave the computer. Until the output adapter for the packet is known, `dev` points to the input adapter. `rx_dev` always points to the network device that received the packet.

Figure 4-3. The `sk_buff` structure, including management data for a packet.

```
struct sk_buff
{
   struct sk_buff        *next,*prev;
   struct sk_buff_head   *list;
   struct sock           *sk;
   struct timeval        stamp;
   struct net_device     *dev, *rx_dev;

   union /* Transport layer header */
   {
         struct tcphdr   *th;
         struct udphdr   *uh;
         struct icmphdr  *icmph;
         struct igmphdr  *igmph;
         struct iphdr    *ipiph;
         struct spxhdr   *spxh;
         unsigned char   *raw;
   } h;

   union /* Network layer header */
   {
         struct iphdr    *iph;
         struct ipv6hdr  *ipv6h;
         struct arphdr   *arph;
         struct ipxhdr   *ipxh;
         unsigned char   *raw;
   } nh;

   union /* Link layer header */
   {
         struct ethhdr   *ethernet;
         unsigned char   *raw;
   } mac;

   struct dst_entry      *dst;
   char                  cb[48];
   unsigned int          len, csum;
   volatile char         used;
   unsigned char         is_clone, cloned, pkt_type, ip_summed;
   __u32                 priority;
   atomic_t              users;
   unsigned short        protocol, security;
   unsigned int          truesize;
   unsigned char         *head, *data, *tail, *end;
```

```
    void                    (*destructor)(struct sk_buff *);

     ...
};
```

- `h, nh`, and `mac` are pointers to packet headers of the transport layer (`h`), the network layer (`nh`), and the MAC layer (`mac`). These pointers are set for a packet as it travels across the kernel. (See Figure 4-2.) For example, the `h` pointer of an IP packet is set in the function `ip_rcv()` to the IP protocol header (type `iphdr`).
- `dst` refers to an entry in the routing cache, which means that it contains either information about the packet's further trip (e.g., the adapter over which the packet is to leave the computer) or a reference to a MAC header stored in the hard header cache. (See Chapters 15 and 16a.)
- `cloned` indicates that a packet was cloned. Clones will be explained in detail later in this chapter. For now, it is sufficient to understand that clones are several copies of a packet and that, though several `sk_buff` structures exist for a packet, they all use one single packet data location jointly.
- `pkt_type` specifies the type of a packet, which can be one of the following:
  - `PACKET_HOST` specifies packet a sent to the local host.
  - `PACKET_BROADCAST` specifies a broadcast packet.
  - `PACKET_MULTICAST` specifies a multicast packet.
  - `PACKET_OTHERHOST` specifies packets not destined for the local host, but received by special modes (e.g., the promiscuous mode).
  - `PACKET_OUTGOING` specifies packets leaving the computer.
  - `PACKET_LOOPBACK` specifies packets sent from the local computer to itself.
  - `PACKET_FASTROUTE` specifies packets fast-forwarded between special network cards (`fastroute` is not covered in this book).
- `len` designates the length of a packet represented by the socket buffer. This considers only data accessible to the kernel. This means that only the two MAC addresses and the type/length field are considered in an Ethernet packet. The other fields (preamble, padding, and checksum) are added later in the network adapter, which is the reason why they are not handled by the kernel.
- `data, head, tail, end`: The `data` and `tail` pointers point to currently valid packet data. Depending on the layer that currently handles the packet, these parameters specify the currently valid protocol data unit.

  `head` and `end` point to the total location that can be used for packet data. The latter storage location is slightly bigger to allow a protocol to add protocol data before or after the packet, without the need to copy the packet data. This avoids expensive copying of the packet data location. If it has to be copied in rare cases, then appropriate methods can be used to create more space for packet data.

  The space between `head` and `data` is called headroom; the space between `tail` and `end` is

called tailroom.

- The other parameters are not discussed here, because they are of minor importance. Some of them are discussed in other chapters (e.g., `netfilter` in Section 19.3).

The pointer `datarefp` is actually not part of the `sk_buff` structure, because it is located at the end of the packet data space and not defined as a variable of a structure. (See Figure 4-1.) `datarefp` is a reference counter; it can be easily addressed and manipulated by use of the macro `skb_datarefp(skb)`.

The reference counter was arranged in this way because, during cloning of socket buffers, several `sk_buff` structures will still point to the same packet data space. If a socket buffer is released, then no other references to the packet data space should also release the packet data space. Otherwise, this would quickly lead to a huge storage hole. The only location where the number of references to packet data can be managed is the packet data space itself, because there is no list managing all clones of a packet. For this reason, and to avoid having to create another data type, we simply reserve a few more bytes than specified by the user when allocating the packet data space. Using the macro `skb_datarefp`, it is easy to access and test the reference counter to see whether there are other references to the packet data space, in addition to the own reference.

4.1.1 Operations on Socket Buffers

The Linux kernel offers you a number of functions to manipulate socket buffers. In general, these functions can be grouped into three categories:

- Create, release, and duplicate socket buffers: These functions assume the entire storage management for socket buffers and their optimization by use of socket-buffer caches.
- Manipulate parameters and pointers within the `sk_buff` structure: These mainly are operations to change the packet data space.
- Manage socket buffer queues.

Creating and Releasing Socket Buffers

`alloc_skb()`	net/core/skbuff.c

`alloc_skb(size, gpf_mask)` allocates memory for a socket buffer structure and the corresponding packet memory. In this case, size specifies the size of the packet data space, where this space will be increased (aligned) to the next 16-bit address.

In the creation of a new socket buffer, no immediate attempt is made to allocate the memory with `kmalloc()` for the `sk_buff` structure; rather, an attempt is made to reuse previously consumed `sk_buff` structures. Note that requesting memory in the kernel's storage management is very expensive and that, because structures of the same type always require the same size, an attempt is first made to reuse an `sk_buff` structure no longer required. (This approach can be thought of as simple recycling; see Section 2.6.2.)

There are two different structures that manage consumed socket buffer structures:

- First, each CPU manages a so-called `skb_head_cache` that stores packets no longer needed. This is a simple socket buffer queue, from which `alloc_skb()` takes socket buffers.
- Second, there is a central stack for consumed `sk_buff` structures (`skbuff_head_cache`).

If there are no more `sk_buff` structures available for the current CPU, then `kmem_cache_alloc()` is used to try obtaining a packet from the central socket-buffer cache (`skbuff_head_cache`). If this attempt fails, then `kmalloc()` is eventually used. `gfp_mask` contains flags required to reserve memory.

Using these two caches can be justified by the fact that many packets are created and released in a system (i.e., the memory of `sk_buff` structures is frequently released), only to be required again shortly afterwards. The two socket buffer caches were introduced to avoid this expensive releasing and reallocating of memory space by the storage management (similarly to first-level and second-level caches for CPU memory access). This means that the time required to release and reserve `sk_buff` structures can be shortened. When `kmem_cache_alloc()` is used to reserve an `sk_buff` structure, the function `skb_header_init()` is called to initialize the structure. It will be described further below.

Naturally, for the `sk_buff` structure, a socket buffer requires memory for the packet data. Because the size of a packet is usually different from and clearly bigger than that of an `sk_buff` structure, a method like the socket-buffer cache does not provide any benefit. The packet data space is reserved in the usual way (i.e., by use of `kmalloc()`).

The pointers head, `data, tail`, and `end` are set once memory has been reserved for the packet data. The counters `user` and `datarefp` (number of references to these socket buffer structure) are set to one. The data space for packets begins to grow from the top (`data`) (i.e., at that point, the socket buffer has no headroom and has tailroom of `size` bytes).

`dev_alloc_skb()`	include/linux.skbuff.h

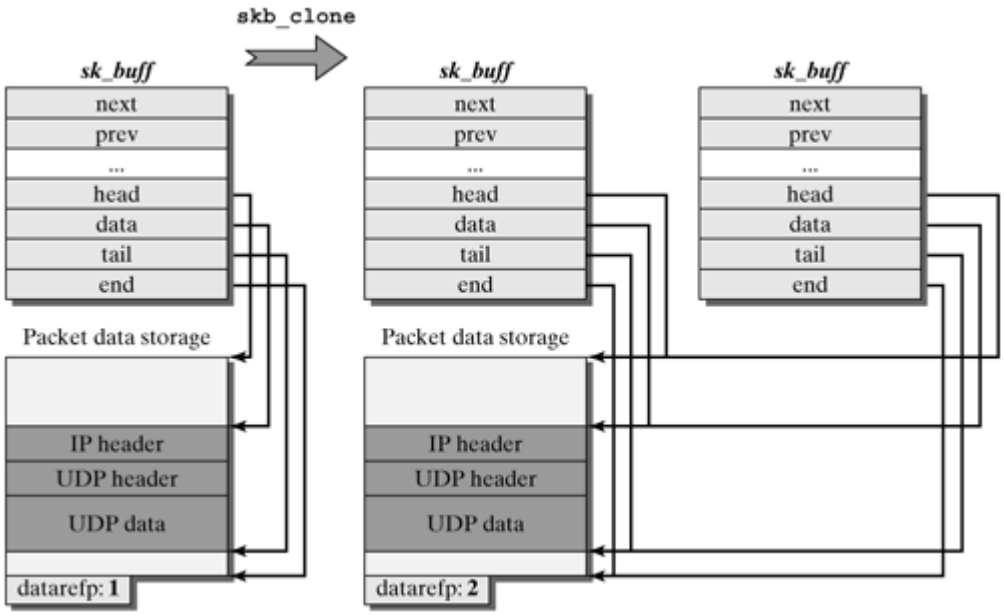
`dev_alloc_skb(length)` uses the function `alloc_skb()` to create a socket buffer. The length of this socket buffer's packet data space is `length` + 16 bytes. Subsequently, `skb_reserve(skb, 16)` is used to move the currently valid packet data space 16 bytes backwards. This means that the packet has now a headroom of 16 bytes and a tailroom of `length` bytes.

`skb_copy()`	net/core/skbuff.c

`skb_copy(skb, gfp_mask)` creates a copy of the socket buffer `skb`, copying both the `sk_buff` structure and the packet data. (See Figure 4-4.) First the function uses `alloc_skb()` to obtain a new `sk_buff` structure; then it sets the attributes. Note that only protocol-specific parameters (`priority, protocol,...`), the relevant network device (`device`), and an entry in the route cache are accepted. All pointers dealing with the concatenation of socket buffers (`next, prev, sk, list`) are set to `null`.

Figure 4-4. Copying socket buffers.

[View full size image]



Memory needed for the payload of the new socket buffers is allocated by `kmalloc()` and copied by `memcopy()`. Subsequently, pointers to the new data space are set in the new `sk_buff` structure. The result of `skb_copy()` is a new socket buffer (with its own packet data space), which exists independently of the original and can be processed independently. This means that the reference counter of the created copy also shows a value of one, in contrast to a using `skb_clone()` to replicate a packet.

`skb_copy_expand()` net/core/skbuff.c

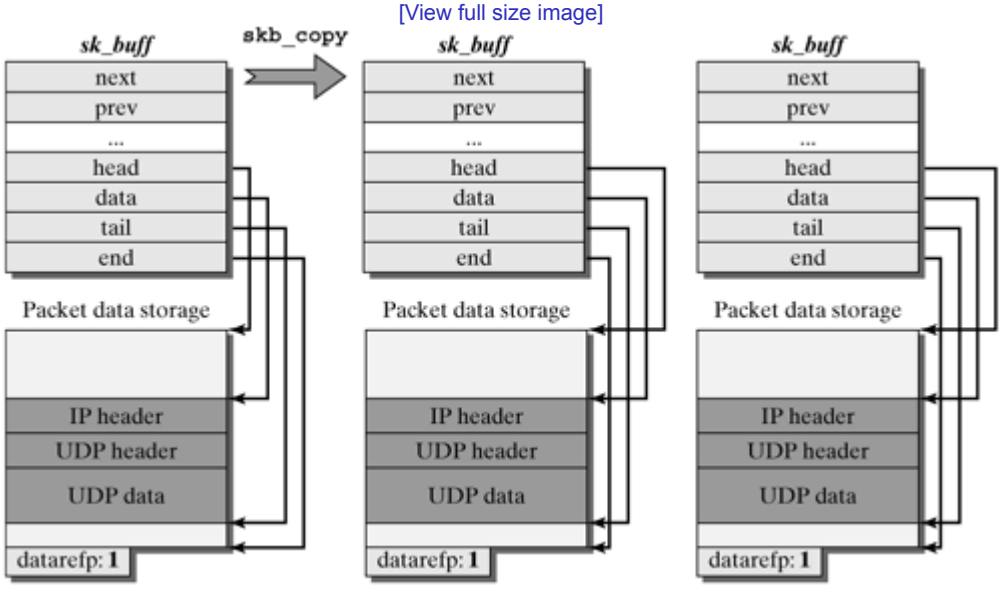
`skb_copy_expand(skb, newheadroom, newtailroom, gfp_mask)` also creates a new and independent copy of the socket buffer and packet data; however, a larger space before and after the packet data can be reserved. `newheadroom` and `newtailroom` specify the size of this space before and behind the packet data space, respectively.

`skb_clone()` net/core/skbuff.c

`skb_clone()` also creates a new socket buffer; however, it allocates only one new `sk_buff` structure, and no second memory space for packet data. The pointers of the original `sk_buff` structure and of the new structure point to the same packet data space. There is no backward reference from the packet memory to the references `sk_buff` structures, so the packet memory should be read-only. Figure 4-5 shows the situation before and after `skb_clone()` is called. Among other things, this function is required in multicast implementation. (See Chapter 17.) This allows us to prevent the time-intensive copying of a complete packet data space when a packet is to be sent to several network devices. The memory containing packet data is not released before the variable `datarefp` contains a value of one (i.e., when there is only one reference to the packet data space left).

Figure 4-5. Cloning socket buffers.

[View full size image]



`kfree_skb()`	include/linux/skbuff.h

`kfree_skb()` does the same thing as `kfree_skbmem()` and is called by `kfree_skb()`, but it additionally tests whether the socket buffer is still in a queue (if so, an error message is output). In addition, it removes the reference from the route cache and, if present, calls a `destructor()` for the socket buffer. `kfree_skb()` should be preferred over other options because of these additional security checks.

`dev_kfree_skb()`	include/linux/skbuff.h

`dev_kfree-skb(skb)` is identical to the method `kfree_skb()` and is mapped to `kfree_skb()` by a preprocessor macro.

`kfree_skbmem()`	include/linux/skbuff.h

`kfree_skbmem()` frees a socket buffer, provided that it was not cloned and that no instance in the kernel refers to it `(datrefp - 1)`. The variable `skb_cloned` is tested for null, and `datarefp` is tested for one. If everything is okay, `kfree()` first releases the packet memory. Then `skb_head_to_pool()` is used to insert the `sk_buff` structure into the socket-buffer cache of the current processor for further use. This means that the memory of the socket-buffer structure is not released for general use `(kfree())`, but instead is buffered for recycling.

skb_header_init()	include/linux/skbuff.h

skb_header_init() initializes some fields of the sk_buff structure with standard values. Most fields are set to null or NULL, and PACKET_HOST is registered as the packet type.

Manipulating the Packet Data Space

The following functions are declared in the include file <Linux/skbuff.h>. Most of them are defined as inline and have only little functionality; nevertheless, they are important and are used often.

skb_get()	include/linux/skbuff.h

This increments the number of user references to the sk_buff structure by one.

skb_unshare()	include/linux/skbuff.h

skb_unshared(skb) uses skb_cloned to check for whether the socket buffer is available for exclusive use. If it isn't, then a copy of skb is created and returned, so that an exclusive socket buffer is available. In the original packet, the reference counter is decremented by a value of one.

skb_put()	include/linux/skbuff.h

skb_put(skb, len) is an inline function that appends data to the end of the current data range of a packet. Though this occurs seldom, because most protocols write their PCI (Protocol Control Information) before the current protocol data unit, there are a few protocol, that require this. More specifically, skb_put() increments the pointer tail and the parameter skb?gt; by len. Note that skb_put() merely sets the pointers again; the caller is responsible for copying the correct data to the packet data space. The return value is the old value of skb?gt;tail, so as to be able to add new packet data to the correct place. Before calling skb_put(), we should confirm that the tailroom is sufficient; otherwise, the kernel will output an error message and call skb_over_panic().

skb_push()	include/linux/skbuff.h

skb_push(skb, len) works like skb_put(), but increases the current packet data space at the beginning of the packet by len bytes. This means that the data pointer is decremented by len, and skb?gt;len is incremented by this amount. The return value of skb_push() points to the new data space (skb?gt;data, in this case). Again, we should first check the headroom size.

skb_pull()	include/linux/skbuff.h

skb_pull(skb, len) serves to truncate len bytes at the beginning of a packet. The pointer skb?gt;data is adjusted, and the length of the packet (skb?gt;len) is reduced accordingly—but, first, we check on whether there are still len bytes in the free part of the packet data space.

skb_tailroom() include/linux/skbuff.h

skb_tailroom(skb) returns the bytes still free at the end of the data space. If skb_put() requests more data in this space than skb_tailroom states, then this will lead to a kernel panic.

skb_headroom() include/linux/skbuff.h

skb_headroom(skb) returns (data ?head). This corresponds to the amount of free bytes in the front space of the packet data memory. Exactly skb_headroom bytes can still be inserted into the packet by skb_push().

skb_realloc_headroom() include/linux/skbuff.h

skb_realloc_headroom(skb, newheadroom) is required when the memory space between skb?gt;data and skb?gt;head is getting too small. This function can be used to create a new socket buffer with a headroom corresponding to the size newheadroom (and not one single byte more). The data part of the old socket buffer is copied into the new one, and most parameters of the sk_buff structure are taken from the old one. Only sk and list are set to NULL. skb_realloc_headroom() is implemented by calling the function skb_copy_expand().

skb_reserve() include/linux/skbuff.h

skb_reserve(skb, len) shifts the entire current data space backwards by len bytes. This means that the total length of this space remains the same. Of course, this function is meaningful only when there are no data in the current space yet, and only if the initial occupancy of this space has to be corrected.

skb_trim() include/linux/skbuff.h

skb_trim(skb, len) sets the current packet data space to len bytes, which means that this space now extends from the initial occupancy of data to tail - data + len. This function is normally used to truncate data at the end (i.e., we call skb_trim() with a length value smaller than the current packet size).

skb_cow() include/linux/skbuff.h

`skb_cow(skb, headroom)` checks on whether the passed socket buffer has still at least `headroom` bytes free in the front packet data space and whether the packet is a clone. If either of the two situations is true, then `skb_alloc_headroom(skb, headroom)` creates and returns a new independent packet. If none of the two tests is true, then the socket buffer `skb` is returned. `skb_cow()` is used when a protocol requires an independent socket buffer with sufficient headroom.

4.1.2 Other Functions

`skb_cloned()`	include/linux/skbuff.h

`skb_cloned(skb)` specifies whether this socket buffer was cloned and whether the corresponding packet data space is exclusive. The reference counter `datarefp` is used to check this.

`skb_shared()`	include/linux/skbuff.h

`skb_shared(skb)` checks whether `skb?gt;users` specifies one single user or several users for the socket buffer.

`skb_over_panic()`, `skb_under_panic()`	include/linux/skbuff.h

These functions are used as error-handling routines during an attempt to increase too small a headroom or tailroom of a socket buffer. A debug message is output after each function, and the function `BUG()` is called.

`skb_head_to_pool()`	net/core/skbuff.c

`skb_head_to_pool(skb)` is used to register a socket buffer structure with the socket-buffer pool of the local processor. It is organized as a simple socket-buffer queue, so this product is simply added to the front of the queue by `skb_queue_head()`. This means that the memory of the socket buffer is not released, but buffered for use by other network packets. This method is much more efficient than to repeatedly allocate and release the memory of a socket buffer by the more complex memory management of the kernel.

The queue `skb_head_pool[smp_processor_id()].list` cannot grow to an arbitrary length; it can contain a maximum of `sysctl_hot_list_len`. As soon as this size is reached, additional socket buffers are added to the central pool for reusable socket buffers `(skbuff_head_cache)`.

`skb_head_from_pool()`	net/core/skbuff.c

This function is used to remove and return a socket buffer from the pool of used socket buffers of the current processor.

Previous Next