Management of *sk_buffs*

The buffers used by the kernel to manage network packets are referred to as *sk_buffs* in Linux. (Their BSD counterparts are referred to as *mbufs*). The buffers are always allocated as *at least* two separate components: a fixed size header of type *struct sk_buff*; and a variable length area large enough to hold all or part of the data of a single packet.

The header is a large structure in which the function of many of the elements is fairly obvious, but the use of some others, especially the length related fields and the pointers into the data component are sometimes not especially clear.

```
struct sk_buff {
    /* These two members must be first. */
    struct sk_buff          *next;
    struct sk_buff          *prev;

    struct sock             *sk;      /* owner socket */
    struct skb_timeval      tstamp;   /* arrival time */
    struct net_device       *dev;     /* output dev   */
    struct net_device       *input_dev;  /* input dev */

```

Protocol header pointers

The next major section contains definitions of pointers to transport, network, and link headers as unions so that only a single word of storage is allocated for each layer's header pointer. Not all of these pointers will be valid all of the time. In fact on the output path, all of the pointers will be *invalid initially* and should thus be used with care!

```

        union {
                struct tcphdr   *th;
                struct udphdr   *uh;
                struct icmphdr  *icmph;
                struct igmphdr  *igmph;
                struct iphdr    *ipiph;
                struct ipv6hdr  *ipv6h;
                unsigned char   *raw;
        } h;      // <--- Transport header address

        union {
                struct iphdr    *iph;
                struct ipv6hdr  *ipv6h;
                struct arphdr   *arph;
                unsigned char   *raw;
        } nh;     // <--- Network header address

        union {
                unsigned char   *raw;
        } mac;    // <--- MAC header address
```

Routing related entries

Th *dst_entry* pointer is an extremely important field is a pointer to the route cache entry used to route the *sk_buff*. This route cache element to which it points contains pointers to functions that are invoked to forward the packet. This pointer *dst* must point to a valid route cache element *before* a buffer is passed to the IP layer for transmission.

```
        struct   dst_entry       *dst;
```

The *sec_path* pointer is a relatively new optional field which supports additional "hooks" for network security.

```
        struct   sec_path        *sp;
```

Scratch pad buffer

The control buffer is an e-junkyard that can be used as a scratch pad during processing by a given layer of the protocol. Its main use is by the IP layer to compile header options.

```
/*
 * This is the control buffer. It is free to use for every
 * layer. Please put your private variables there. If you
 * want to keep them across layers you have to skb_clone()
 * first. This is owned by whoever has the skb queued ATM.
*/
   char      cb[48];
```

Length fields

The usage of *len, data_len,* and *truesize* are easy to confuse.

- The value of *truesize* is the length of the variable size data component(s) plus the size of the *sk_buff* header. This is the amount that is charged against the *sock*'s send or receive quota.

The values of the other two are set to zero at allocation time.

- When a packet is received, the *len* field is set to the size of a complete input packet including headers. This value includes data in the *kmalloc'd* part, fragment chain and/or unmapped page buffers. As headers are removed or added the value of *len* is decremented and incremented accordingly.

- The value of the *data_len* field is the number of bytes in the fragment chain and in unmapped page buffers and is normally 0.

```
 unsigned int             len,
                          data_len,
                          mac_len,
                          csum;
__u32                     priority;
__u8                      local_df:1,
                          cloned:1,
                          ip_summed:2,
                          nohdr:1,
                          nfctinfo:3;
__u8                      pkt_type:3,
                          fclone:2,
                          ipvs_property:1;
__be16                    protocol;


void                      (*destructor)(struct sk_buff *skb);
/* These must be at end, see alloc_skb() for details.  */

  unsigned int              truesize;
```

Reference counting

Reference counting is a critically important technique that is used to prevent both memory leaks and invalid pointer accesses. It is used in all network data structures that are dynamically allocated and freed. Unfortunately there is no standard name for either the variable that contains the reference count nor the helper function (if any) that manipulates it :-(

- The atomic variable *users* counts the number of processes that hold a reference to the *sk_buff* structure itself.
- It is incremented whenever the buffer is *shared.*
- It is decremented when a buffer is logically *freed*.
- The buffer is physically freed only when the reference count reaches 0.

```
315         atomic_t                    users;
```

Pointers into the data area

These pointers all point into the variable size component of the buffer which actually contains the packet data. At allocation time

- *data, head,* and *tail* initially point to the start of the allocated packet data area and
- *end* points to the *skb_shared_info* structure which begins at next byte beyond the area available for packet data.

A large collection of inline functions defined in *include/linux/skbuff.h* may be used in adjustment of *data, tail,* and *len* as headers are added or removed.

```
        unsigned char           *head,
                                *data,
                                *tail,
                                *end;
};
```

MAC Header definition

Linux prefers the standard DIX ethernet header to 802.x/803.x framing. However, the latter are both also supported.

```
struct ethhdr
{
    unsigned char h_dest[ETH_ALEN];   /* dest eth addr */
    unsigned char h_source[ETH_ALEN]; /* src eth addr  */
    unsigned short  h_proto;          /* packet type*/
};
```

IP Header

```
struct iphdr {
#if defined(__LITTLE_ENDIAN_BITFIELD)
    __u8 ihl:4,
         version:4;
#elif defined (__BIG_ENDIAN_BITFIELD)
    __u8 version:4,
         ihl:4;
#endif
    __u8 tos;
    __u16 tot_len;
    __u16 id;
    __u16 frag_off;
    __u8 ttl;
    __u8 protocol;
    __u16 check;
    __u32 saddr;
    __u32 daddr;
};
```

The *skb_shared_info* structure

The *struct skb_shared_info* defined in *include/linux/skbuff.h* is used to manage fragmented buffers and unmapped page buffers. This structure resides at the end of the *kmalloc'd* data area and is pointed to by the *end* element of the *struct sk_buff* header. The atomic *dataref* is a reference counter that counts the number of entities that hold references to the *kmalloc'd* data area.

When a buffer is *cloned* the *sk_buff* header is copied but the data area is shared. Thus cloning increments *dataref* but *not users.*

```
/* This data is invariant across clones and lives at
 * the end of the header data, ie. at skb->end.
 */
struct skb_shared_info {
    atomic_t        dataref;
    unsigned short  nr_frags;
    unsigned short  gso_size;
 /* Warning: this field is not always filled in (UFO)! */
    unsigned short  gso_segs;
    unsigned short  gso_type;
    unsigned int    ip6_frag_id;
    struct sk_buff  *frag_list;
    skb_frag_t      frags[MAX_SKB_FRAGS];
};
```

Functions of structure elements:

dataref	The number of users of the *data* of this *sk_buff* this value is incremented each time a buffer is *cloned*.
frag_list	If not NULL, this value is pointer to the next *sk_buff* in the chain. The fragments of an IP packet undergoing reassembly are chained using this pointer.
frags	An array of pointers to the page descriptors of up unmapped page buffers.
nr_frags	The number of elements of *frags* array in use.

Support for fragmented data in *sk_buffs*

The *skb_shared_info* structure is used when the data component of a single *sk_buff* consists of multiple fragments. There are actually two mechanisms with which fragmented packets may be stored:

- The **frag_list* pointer is used to link a list of *sk_buff* headers together. This mechanism is used at receive time in the reassembly of fragmented IP packets.

- The *nr_frags* counter and the *frags[]* array are used for unmapped page buffers. This facility was added in kernel 2.4 and is presumably designed to support some manner of zero-copy facility in which packets may be received directly into pages that can be mapped into user space.

- The value of the *data_len* field represents the sum total of bytes resident in fragment lists and unmapped page buffers.

- Except for reassembly of fragmented packets the value of *data_len* is always 0.

Typical buffer organization

The fragment list and unmapped buffer structures lead to a recursive implementation of checksumming and data movement code that is quite complicated in nature.

Fortunately, in practice, an unfragmented IP packet always consists of only:

- An instance of the *struct sk_buff* buffer header.

- The kmalloc'd ``data'' area allocated holding both packet headers an data.

Unmapped page buffers

The *skb_frag_t* structure represents an unmapped page buffer.

```
/* To allow 64K frame to be packed as single skb without
   frag_list */

#define MAX_SKB_FRAGS (65536/PAGE_SIZE + 2)

struct skb_frag_struct {
     struct page *page;
    __u16 page_offset;
    __u16 size;
};
```

Functions of structure elements:

page	Pointer to a *struct page* which controls the real memory page frame.
offset	Offset in page from where data is stored.
size	Length of the data.

Management of buffer content pointers

Five fields are most important in the management of data in the *kmalloc'd* component of the buffer.

head	Points to the first byte of the *kmalloc'd* component. It is set at buffer allocation time and never adjusted thereafter.
end	Points to the start of the *skb_shared_info* structure (i.e. the first byte beyond the area in which packet data can be stored.) It is also set at buffer allocation time and never adjusted thereafter.
data	Points to the start of the "data" in the buffer. This pointer may be adjusted forward or backward as header data is removed or added to a packet.
tail	Points to the byte following the "data" in the buffer. This pointer may also be adjusted.
len	The value of *tail - data.*

Other terms that are commonly encountered include:

headroom	The space between the *head* and *data* pointers
tailroom	The space between that *tail* and end pointers.

Initially *head = data = tail* and *len = 0.*

Buffer management convenience functions

- Linux provides a number of convenience functions for manipulating these fields. Note that *none of these functions actually copies any data into or out of the buffer!*

- They are good to use because they provide built in checks for various overflow and underflow errors that if undetected *can cause unpredictable behavior for which the cause can be very hard to identify!*

Reserving space at the head of the buffer

The *skb_reserve()* function defined in *include/linux/skbuff.h* is called to reserve *headroom* for the hardware header which shall be filled in later. Since the *skb->head* pointer always points to the start of the *kmalloc'd* area, the size of the *headroom* is defined as *skb->data - skb->head*. The *head* pointer is left unchanged, the *data* and *tail* pointers are advanced by the specified amount.

A transport protocol send routine might use this function to reserve space headers and point *data* to where the data should be copied from user space.

```
static inline void skb_reserve(struct sk_buff *skb, int len)
{
        skb->data += len;
        skb->tail += len;
}
```

Appending data to the tail of a buffer

The *skb_put()* function can be used to increment the *len* and *tail* values after data has been placed in the *sk_buff()*. The actual filling of the buffer is most commonly performed by

- a DMA transfer on input or
- a *copy_from_user()* on output.

The transport protocol might use this function after copying the data from user space.

```
static inline unsigned char *skb_put(struct sk_buff *skb,
                                      unsigned int len)
{
   unsigned char *tmp = skb->tail;
   SKB_LINEAR_ASSERT(skb);
   skb->tail += len;
   skb->len  += len;
   if (unlikely(skb->tail>skb->end))
            skb_over_panic(skb, len, current_text_addr());
   return tmp;
}
```

Inserting new data at the front of buffer.

The *skb_push()* function decrements the *data* pointer by the *len* passed in and increments the value of *skb->len* by the same amount. It is used to extend the data area back toward the head end of the buffer. It returns a pointer the new value of *skb->data.*

The transport layer protocol might use this function when preparing to build transport and IP headers.

```
static inline unsigned char *skb_push(struct sk_buff *skb,
                                               unsigned int len)
{
        skb->data -= len;
        skb->len  += len;
        if (unlikely(skb->data<skb->head))
                skb_under_panic(skb, len, current_text_addr());
        return skb->data;
}
```

Removing data from the front of the buffer

The *skb_pull()* function logicially removes data from the start of a buffer returning the space to the headroom. It increments the *skb->data* pointer and decrements the value of *skb->len* effectively removing data from the head of a buffer and returning it to the headroom. It returns a pointer to the new start of *data*.

The receive side of the transport layer might use this function during reception *when removing a header from the packet.*

The BUG_ON condition will raised if an attempt is made to pull more data than exists causing *skb->len* to become negative or if at attempt is made to pull across the boundary between the kmalloc'd part and the fragment chain.

```
static inline unsigned char *__skb_pull(struct sk_buff *skb,
                                        unsigned int len)
{
        skb->len -= len;
        BUG_ON(skb->len  <  skb->data_len);
        return skb->data += len;
}

static inline unsigned char *skb_pull(struct sk_buff *skb,
                                          unsigned int len)
{
        return unlikely(len > skb->len) ? NULL :
                                 __skb_pull(skb, len);
}
```

Removing data from the tail of a buffer

The *skb_trim()* function can be used to decrement the length of a buffer and move the *tail* pointer toward the head. The *new length* not the amount to be trimmed is passed in. This might be done to remove a trailer from a packet. The process is straightforward *unless* the buffer is non-linear. In that case, *___pskb_trim ()*must be called and it becomes *your worst nightmare.*

```
static inline void __skb_trim(struct sk_buff *skb,
                              unsigned int len)
{
   if (unlikely(skb->data_len)) {
         WARN_ON(1);
         return;
   }
   skb->len  = len;
   skb->tail = skb->data + len;
}


static inline void skb_trim(struct sk_buff *skb,
                                   unsigned int len)
{
   if (skb->len > len)
       __skb_trim(skb, len);
}

static inline int __pskb_trim(struct sk_buff *skb,
     unsigned int len)
{
   if (skb->data_len)
       return ___pskb_trim(skb, len);
  __skb_trim(skb, len);
  return 0;
}

static inline int pskb_trim(struct sk_buff *skb,
                              unsigned int len)
{
       return (len < skb->len) ? __pskb_trim(skb, len) : 0;
}
```

Obtaining the available head and tail room.

The following functions may be used to obtain the length of the *headroom* and *tailroom.* If the buffer is nonlinear, the tailroon is 0 by convention.

```
static inline int skb_headroom(const struct sk_buff *skb)
{
        return skb->data - skb->head;
}

static inline int skb_tailroom(const struct sk_buff *skb)
{
    return skb_is_nonlinear(skb) ? 0 : skb->end - skb->tail;
}
```

Determining how much data is in the *kmalloc'd* part of the buffer..

The *skb_headlen()* function returns the length of the data presently in the *kmalloc'd* part of the buffer. This section is sometimes referred to as the header (even though the *struct sk_buff* itself is more properly referred to as the buffer header.)

```
static inline unsigned int skb_headlen(const struct sk_buff
                                                     *skb)
{
       return skb->len - skb->data_len;
}
```

Non-linear buffers

A buffer is linear if and only if all the data is contained in the *kmalloc'd* header.
The *skb_is_nonlinear()* returns true if there is data in the fragment list or in unmapped page buffers.

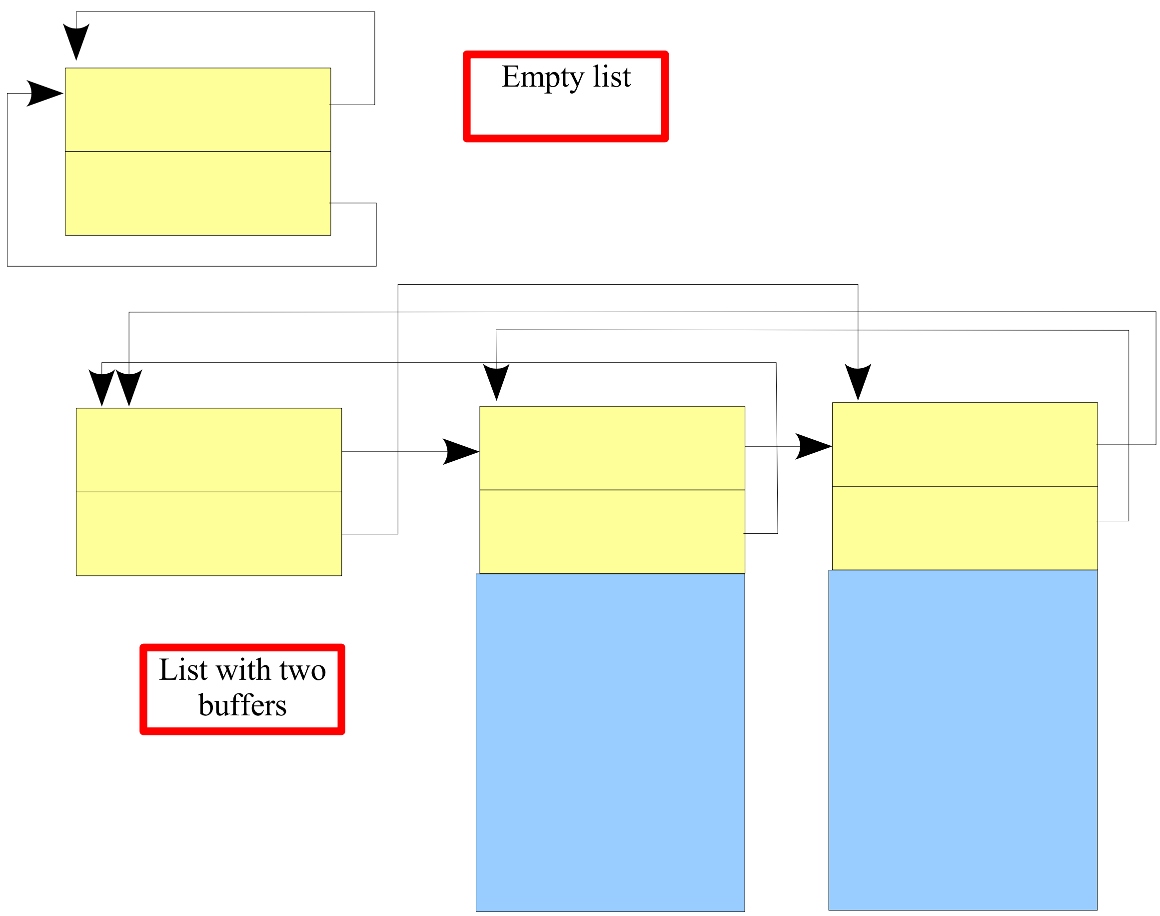
```
static inline int skb_is_nonlinear(const struct
                                        sk_buff *skb)
{
         return skb->data_len;
}
```

Managing lists of *sk_buffs*

Buffers awaiting processing by the next layer of the network stack typically reside in linked lists that are called buffer queues. The structure below defines a buffer queue header. Because the *sk_buff* structure also begins with **next* and **prev* pointers, *pointers to sk_buff and sk_buff_head are sometimes used interchangably.*

```
struct sk_buff_head {
/* These two members must be first. */
        struct sk_buff   *next;
        struct sk_buff   *prev;

        __u32            qlen;    /* # of buffers in the list */
        spinlock_t       lock;    /* MUST be held when adding */
};                                /* or removing buffers      */
```



Queue management functions

A number of functions are provided by the kernel to simplify queue management operations and thus improve their reliability. These functions are defined in *include/linux/skbuff.h.*

Obtaining a pointer to the first buffer in the queue.

The *skb_peek()* function may be used to obtain a pointer to the first element in a non-empty queue. Note that *sk_buff_head* and *sk_buff* pointers are used interchangably in line 569. This (bad) practice works correctly because the first two elements of the *sk_buff_head* structure are the same as those of the *sk_buff*. If the *next* pointer points back to the header, the list is empty and NULL is returned.

```
567 static inline struct sk_buff *skb_peek(struct sk_buff_head *list_)
568 {
569         struct sk_buff *list = ((struct sk_buff *)list_)->next;
570         if (list == (struct sk_buff *)list_)
571                 list = NULL;
572         return list;
573 }
```

Testing for an empty queue.

The *skb_queue_empty()* functions returns *true* if the queue is empty an *false* if it is not.

```
static inline int skb_queue_empty(const struct sk_buff_head
                                                       *list)
{
        return list->next == (struct sk_buff *)list;
}
```

Removal of buffers from queues

The *skb_dequeue()* function is used to remove the first buffer from the *head* of the specified specified queue.   It calls *__skb_dequeue after obtaining the list's associated lock*.

```
static inline struct sk_buff *skb_dequeue(struct
                              sk_buff_head *list)
{
    long flags;
    struct sk_buff *result;

    spin_lock_irqsave(&list->lock, flags);
    result = __skb_dequeue(list);
    spin_unlock_irqrestore(&list->lock, flags);
    return result;
}
```

The mechanics of dequeue

The *__skb_dequeue()* function does the work of actually removing an *sk_buff* from the receive queue. Since the *sk_buff_head* structure contains the same link pointers as an actual *sk_buff* structure, it can masquerade as a list element as is done via the cast in line 708.

In line 708 *prev* is set to point to the *sk_buff_head.* Then in line 709, the local variable *next* receives the value of the *next* pointer in the *sk_buff_head.* The test in line 711 checks to see if the *next* pointer still points to the *sk_buff_head.* If so the list was empty. If not the first element is removed from the list and its link fields are zeroed.

```
704 static inline struct sk_buff *__skb_dequeue(struct
                                              sk_buff_head *list)
705 {
706         struct sk_buff *next, *prev, *result;
707
708         prev = (struct sk_buff *) list;
709         next = prev->next;
710         result = NULL;
711         if (next != prev) {
712                 result       = next;
713                 next         = next->next;
714                 list->qlen--;
715                 next->prev   = prev;
716                 prev->next   = next;
717                 result->next = result->prev = NULL;
718         }
719         return result;
720 }
```

Adding buffers to queues

Since buffer queues are usually managed in a FIFO manner and buffers are removed from the head of the list, they are typically added to a list with *skb_queue_tail()*.

```
void skb_queue_tail(struct sk_buff_head *list,
                         struct sk_buff *newsk)
{
       unsigned long flags;

       spin_lock_irqsave(&list->lock, flags);
       __skb_queue_tail(list, newsk);
       spin_unlock_irqrestore(&list->lock, flags);
}
```

The mechanics of enqueue

The actual work of enqueuing a buffer on the tail of a queue is done in *__skb_queue_tail()*.

```
static inline void __skb_queue_tail(struct sk_buff_head
                                    *list,
                                    struct sk_buff *newsk)
{
        struct sk_buff *prev, *next;

```

The *sk_buff_head* pointer in the *sk_buff* is set and the length field in the *sk_buff_head* is incremented. (These two lines are reversed from kernel 2.4.x.)

```
        list->qlen++;
        next = (struct sk_buff *)list;
```

Here *next* points to the *sk_buff_head* structure and *prev* point to the *sk_buff* structure that was previously at the tail of the list. Note that the list structure is circular with the *prev* pointer of the *sk_buff_head* pointing to the *last* element of the list.

```
        prev = next->prev;
        newsk->next = next;
        newsk->prev = prev;
        next->prev  = prev->next = newsk;
}
```

Removal of *all* buffers from a queue

The *skb_queue_purge()* function may be used to remove all buffers from a queue and free them. This might be used when a socket is being closed and there exist received packets that have not yet been consumed by the application.

When a buffer is being freed be *sure* to use *kfree_skb()* and *not kfree().*

```
void skb_queue_purge(struct sk_buff_head *list)
{
        struct sk_buff *skb;
        while ((skb = skb_dequeue(list)) != NULL)
                kfree_skb(skb);
}
```

This version from *skbuff.h* may be used if and only if the list lock is held.

```
static inline void __skb_queue_purge(struct
                               sk_buff_head *list)
{
        struct sk_buff *skb;
        while ((skb = __skb_dequeue(list)) != NULL)
                kfree_skb(skb);
}
```

Allocation of *sk_buffs* for transmission

The *sock_alloc_send_skb()* function resides in *net/core/sock.c*. It is normally called for this purpose. It is a minimal wrapper routine that simply invokes the *sock_alloc_send_pskb()* function defined in net/core/sock.c, with *data_len* parameter set to zero. The size field passed has historically had the value: *user data size + transport header length + IP header length + device hardware header length + 15*. There may be a new helper function to compute the size now. When you call *sock_alloc_send_skb(),* you must set *noblock* to 0.

And when you allocate a supervisory packet in the context of a *softirq* you must use *dev_alloc_skb()*.

```
struct sk_buff *sock_alloc_send_skb(struct sock *sk,
                                    unsigned long size,
                                    int noblock, int *errcode)
{
    return sock_alloc_send_pskb(sk, size, 0, noblock,
                                errcode);
}
```

When *sock_alloc_send_pskb()* is invoked on the UDP send path via the fast IP build routine, the variable *header_len* will carry the length as computed on the previous page and the variable *data_len* will always be 0. Examination of the network code failed to show any evidence of a non-zero value of *data_len.*

```
static struct sk_buff *sock_alloc_send_pskb(struct sock *sk,
                                  unsigned long header_len,
                                  unsigned long data_len,
                                  int noblock, int *errcode)
{
       struct sk_buff *skb;
       gfp_t gfp_mask;
       long timeo;
       int err;

       gfp_mask = sk->sk_allocation;
       if (gfp_mask & __GFP_WAIT)
               gfp_mask |= __GFP_REPEAT;

       timeo = sock_sndtimeo(sk, noblock);
```

The *sock_sndtimeo()* function defined in include/net/sock.h returns the *sndtimeo* value set by *sock_init_data to MAX_SCHEDULE_TIMEOUT* which in turn is defined as *LONG_MAX* for blocking calls and returns timeout as zero for nonblocking calls.

```
static inline long sock_sndtimeo(struct sock *sk,
                                  int noblock)
{
    return noblock ? 0 : sk->sndtimeo;
}
```

The main allocation loop.

A relatively long loop is entered here. If no transmit buffer space is available the *process will sleep* via the call to *sock_wait_for_wmem()* which appears at line 1213. The function *sock_error()* retrieves any error code that might be present, and clears it atomically from the sock structure.

Exit conditions include

- successful allocation of the *sk_buff*,
- an error condition returned by *sock_error,* closing of the socket, and
- receipt of a signal.

```
1157     while (1) {
1158        err = sock_error(sk);
1159        if (err != 0)
1160             goto failure;
1161
1162        err = -EPIPE;
1163        if (sk->sk_shutdown & SEND_SHUTDOWN)
1164             goto failure;
```

Verifying that quota is not exhausted.

sock_alloc_send_pskb() will allocate an *sk_buff* only if the amount of send buffer space, *sk->wmem_alloc*, that is currently allocated to the socket is less than the send buffer limit, *sk->sndbuf*. The buffer limit is inherited from the system default set during socket initialization.

```
        if (atomic_read(&sk->sk_wmem_alloc) < sk->sk_sndbuf) {
             skb = alloc_skb(header_len, gfp_mask);
```

If allocation worked, *skb* will hold the address of the buffer otherwise it will be 0. Allocation will fail only in case of some catastrophic kernel memory exhaustion.

```
             if (skb) {
                int npages;
                int i;

            /* No pages, we're done... */
               if (!data_len)
                     break;
```

At this point in the code is some awful stuff in which unmapped page buffers are allocated. We will skip over this.

Arrival here means *alloc_skb()* returned 0.

```
                err = -ENOBUFS;
                goto failure;
        }
```

Sleeping until *wmem* is available

If control reaches the bottom of the loop in *sock_alloc_send_pskb(),* then no space was available and if the request has not timed out and there is no signal pending then it is necessary to sleep while the link layer consumes some packets, transmits them and then releases the buffer space they occupy.

```
        set_bit(SOCK_ASYNC_NOSPACE, &sk->sk_socket->flags);
        set_bit(SOCK_NOSPACE, &sk->sk_socket->flags);
        err = -EAGAIN;
        if (!timeo)
                goto failure;
        if (signal_pending(current))
                goto interrupted;
        timeo = sock_wait_for_wmem(sk, timeo);
    }

```

This is the end of *sock_alloc_send_pskb*. The function *skb_set_owner_w()*

- sets the *owner* field of the *sk_buff* to *sk*
- calls *sock_hold()* to increment the *refcount* of the struct sock.
- adds the *truesize* to *sk_wmem_alloc*
- and sets the destructor function field of the skb to *sock_wfree*.

```
        skb_set_owner_w(skb, sk);
        return skb;

interrupted:
        err = sock_intr_errno(timeo);
failure:
        *errcode = err;
        return NULL;
}
```

The *alloc_skb()* function

The actual allocation of the *sk_buff* header structure and the data area is performed by the *alloc_skb()* function which is defined in *net/core/skbuff.c* Comments at the head of the function describe its operation:

``Allocate a new sk_buff. The returned buffer has no headroom and a tail room of size bytes. The object has a reference count of one. The return is the buffer. On a failure the return is NULL. Buffers may only be allocated from interrupts/bottom halves using a gfp_mask of GFP_ATOMIC."

The hardcoded 0 in the call to *__alloc_skb()* says *not* to allocate from the *fclone cache.*

```
static inline struct sk_buff *alloc_skb(unsigned int size,
                                        gfp_t priority)
{
        return __alloc_skb(size, priority, 0);
}
```

The __*alloc_skb()* function

The real work is done here. The wrapper on the previous page only sets the *fclone* flag to 0. A cloned buffer is one in which *two* struct sk_buffs control the same data area. Because reliable transfer protocols usually make exactly one clone of EVERY buffer, each allocation from the *fclone* cache returns two adjacent sk_buff headers.

```
struct sk_buff *__alloc_skb(unsigned int size, gfp_t gfp_mask,
                            int fclone)
{
        kmem_cache_t *cache;
        struct skb_shared_info *shinfo;
        struct sk_buff *skb;
        u8 *data;

```

Cloned and non-cloned buffer headers now are allocated from separate caches.

```
        cache = fclone ? skbuff_fclone_cache :
                         skbuff_head_cache;

```

The *head* is the *struct sk_buff*

```
    /* Get the HEAD */
        skb = kmem_cache_alloc(cache, gfp_mask &
                    ~__GFP_DMA);
        if (!skb)
                goto out;

```

The data portion is allocated from one of the "general" caches. These caches consists of blocks that are multiples of page size, and allocation occurs using a *best fit* strategy.

```
/* Get the DATA. Size must match skb_add_mtu(). */
    size = SKB_DATA_ALIGN(size);
    data = ____kmalloc(size + sizeof(struct
                   skb_shared_info), gfp_mask);
    if (!data)
            goto nodata;

```

All elements of the *struct sk_buff* up to the *truesize* field are set to 0. Then the head, tail, data, and end pointers are set to correct initial state.

```
    memset(skb, 0, offsetof(struct sk_buff, truesize));
    skb->truesize = size + sizeof(struct sk_buff);
    atomic_set(&skb->users, 1);
    skb->head = data;
    skb->data = data;
    skb->tail = data;
    skb->end  = data + size;
```

Finally the *skb_shared_info* structure at the tail of the *kmalloc'ed* part is initialized. Why must it be done sequentially?

```
/* make sure we initialize shinfo sequentially */
    shinfo = skb_shinfo(skb);
    atomic_set(&shinfo->dataref, 1);
    shinfo->nr_frags  = 0;
    shinfo->gso_size = 0;
    shinfo->gso_segs = 0;
    shinfo->gso_type = 0;
    shinfo->ip6_frag_id = 0;
    shinfo->frag_list = NULL;

```

Managing *fclones*.

This looks seriously ugly... An *fclone must* immediately follow the parent in memory. The term *child* refers to the potential clone that immediately follows the *parent* in memory. Furthermore there is an *unnamed atomic variable following* the *child* buffer in the *fclone* cache. This variable is always accessed using the pointer name *fclone_ref* and counts the *total* number of references currently held for the *parent + child.*

Here the atomic *fclone_ref* is set to 1. The *fclone state* of the parent is set to FCLONE_ORIG which makes sense, but the state of the child is set to FCLONE_UNAVAILABLE which seems just backward to me because the child is now AVAILABLE for use in cloning.

It appears that if the buffer *didn't* come from the *fclone* cache that the *skb->fclone* flag is implicitly set to FCLONE_UNAVAILABLE (0) by the *memset(). Ugh.*

```
enum {
        SKB_FCLONE_UNAVAILABLE,
        SKB_FCLONE_ORIG,
        SKB_FCLONE_CLONE,
};
```

```
        if (fclone) {
             struct sk_buff *child = skb + 1;
             atomic_t *fclone_ref = (atomic_t *) (child + 1);

             skb->fclone = SKB_FCLONE_ORIG;
             atomic_set(fclone_ref, 1);

             child->fclone = SKB_FCLONE_UNAVAILABLE;
        }
```

```
out:
        return skb;
nodata:
        kmem_cache_free(cache, skb);
        skb = NULL;
        goto out;
}
```

The old version

```
struct sk_buff *alloc_skb(unsigned int size,int gfp_mask)
{
        struct sk_buff *skb;
        u8 *data;
```

alloc_skb() ensures that when called from an interrupt handler, it is called using the *GFP_ATOMIC* flag. In earlier incarnations of the code it logged up to 5 instances of a warning messages if such was not the case. Now it simply crashes the system!

```
        if (in_interrupt() && (gfp_mask & __GFP_WAIT)) {
                static int count = 0;
                if (++count < 5) {
                        printk(KERN_ERR "alloc_skb called
                                nonatomically "
                           "from interrupt %p\n", NET_CALLER(size));
                        BUG();
                }
                gfp_mask &= ~__GFP_WAIT;
        }
```

Allocation of the header

The *struct sk_buff* header is allocated either from the pool or from the cache via the slab allocator. A *pool* is a typically small list of objects normally managed by the slab allocator that have recently been released by a specific processor in an SMP complex. Thus there is one pool per object type per processor. The objective is of pool usage is to:

- to avoid spin locking and
- to obtain better cache behavior by attempting to ensure that an object that has been recently used is reallocated to the CPU that last used it.

```
/* Get the HEAD */
    skb = skb_head_from_pool();
    if (skb == NULL) {
         skb = kmem_cache_alloc(skbuff_head_cache,
                                gfp_mask & ~__GFP_DMA);
         if (skb == NULL)
              goto nohead;
}
```

Allocating the data buffer

SKB_DATA_ALIGN increments *size* to ensure that some manner of cache line alignment can be achieved. Note that the actual alignment does not occur here.

```
/* Get the DATA. Size must match skb_add_mtu(). */
    size = SKB_DATA_ALIGN(size);
    data = kmalloc(size + sizeof(struct skb_shared_info),
                                        gfp_mask);
    if (data == NULL)
         goto nodata;
```

Header initialization

truesize holds the requested buffer's size + the sizeof of the *sk_buff* header. It does not include slab overhead or the *skb_shared_info*. Initially, all the space in the buffer memory is assigned to the tail component.

```
    /* XXX: does not include slab overhead */
     skb->truesize = size + sizeof(struct sk_buff);

   /* Load the data pointers. */
     skb->head = data;
     skb->data = data;
     skb->tail = data;
     skb->end = data + size;
    /* Set up other state */
     skb->len = 0;
     skb->cloned = 0;
     skb->data_len = 0;

```

Not shared and not cloned.

```
     atomic_set(&skb->users, 1);
     atomic_set(&(skb_shinfo(skb)->dataref), 1);
```

No fragments

```
     skb_shinfo(skb)->nr_frags = 0;
     skb_shinfo(skb)->frag_list = NULL;
     return skb;

 nodata:
     skb_head_to_pool(skb);
 nohead:
     return NULL;
 }
```

Waiting until memory becomes available

If a process enters a rapid send loop, data will accumulate in *sk_buffs* far faster than it can be transmitted. When the sending process has consumed its *wmem* quota it is put to sleep until space is recovered through successful transmission of packets and subsequent release of the *sk_buffs.*

For the UDP path the value of *timeo* is either

- 0 for sockets with the non-blocking attribute or
- the maximum possible *unsigned int* for all others.

When we build a connection protocol, you can copy this code as a basis for waiting inside a call to *cop_listen().*

Sleep/wakeup details

A *timeo of* 0 will have caused a jump to the *failure* exit. Arrival here generally means *wait forever.* The somewhat complex, multi-step procedure used to sleep is necessary to avoid a nasty race condition that could occur with traditional *interruptible_sleep_on() / wake_up_interruptible()* synchronization.

- A process might test for available memory,
- then memory becomes available in a *softirq* and a *wakeup* be issued,
- then the process goes to sleep –
- possibly for a *long* time.

Mechanics of wait

This situation is avoided by putting the *task_struct* on the *waitqueue* before testing for available memory and is explained well in the *Linux Device Drivers* book.

The *struct sock* contains a variable, *wait_queue_head_t   *sk_sleep,* that defines the wait queue on which the process will sleep. The local variable *wait* is the wait queue element that the *prepare_to_wait()* function will put on the queue. The call to *schedule_timeo()* actually initiates the wait.

```
static long sock_wait_for_wmem(struct sock * sk, long timeo)
{
    DEFINE_WAIT(wait);

    clear_bit(SOCK_ASYNC_NOSPACE, &sk->sk_socket->flags);
    for (;;) {
         if (!timeo)
            break;
         if (signal_pending(current))
             break;
         set_bit(SOCK_NOSPACE, &sk->sk_socket->flags);
         prepare_to_wait(sk->sk_sleep, &wait,
                           TASK_INTERRUPTIBLE);
         if (atomic_read(&sk->sk_wmem_alloc) < sk->sk_sndbuf)
             break;
         if (sk->sk_shutdown & SEND_SHUTDOWN)
              break;
         if (sk->sk_err)
              break;
         timeo = schedule_timeout(timeo);
        }
        finish_wait(sk->sk_sleep, &wait);
        return timeo;
}
```

Charging the owner for allocated write buffer space.

The *skb_set_owner_w()* function sets up the destructor function and "bills" the owner for the amount of space consumed. The call to *sock_hold* increments *sk->refcnt* on the *struct sock* to indicate that this *sk_buff* holds a pointer to the *struct sock.* This reference will not be released until *sock_put* is called by the destructor function, *sock_wfree(),* at the time the *sk_buff* is freed.

```
static inline void skb_set_owner_w(struct sk_buff *skb,
                                   struct sock *sk)
{
        sock_hold(sk);
        skb->sk = sk;
        skb->destructor = sock_wfree;
        atomic_add(skb->truesize, &sk->sk_wmem_alloc);
}
```

The *kfree_skb* function.

The *kfree_skb()* function atomically decrements the number of users and invokes *__kfree_skb()* to actually free the buffer when the number of users becomes 0. The standard technique of reference counting is employed, but in a way that is somewhat subtle.

If the *atomic_read()* returns 1, then this thread of control is the only entity that holds a pointer to this *skb_buff.* The subtle part of the procedure is that this also implies there is *no way any other entity is going to be able to obtain a reference*. Since this entity holds the only reference , it would have to provide it and this entity is not going to do that.

If the *atomic_read()* returns 2, for example, there *is* an exposure to a race condition. Both entities that hold references could simultaneously decrement with the result being that both references were lost without *__kfree_skb()* ever being called at all.

The *atomic_dec_and_test()* defined in *include/asm/atomic.h* resolves that potential problem. It atomically decrements the reference counter and returns *true* only if the decrement operation produced 0 .

```
void kfree_skb(struct sk_buff *skb)
{
        if (unlikely(!skb))
                return;
        if (likely(atomic_read(&skb->users) == 1))
                smp_rmb();
        else if (likely(!atomic_dec_and_test(&skb->users)))
                return;
        __kfree_skb(skb);
}
```

Freeing an *sk_buff* the old way

The *kfree_skb()* function atomically decrements the number of users and invokes *__kfree_skb()* to actually free the buffer when the number of users becomes 0. The standard technique of reference counting is employed, but in a way that is somewhat subtle.

If the *atomic_read()* returns 1, then this thread of control is the only entity that holds a pointer to this *skb_buff.* The subtle part of the procedure is that this also implies there is *no way any other entity is going to be able to obtain a reference*. Since this entity holds the only reference , it would have to provide it and this entity is not going to do that.

If the *atomic_read()* returns 2, for example, there *is* an exposure to a race condition. Both entities that hold references could simultaneously decrement with the result being that both references were lost without *__kfree_skb()* ever being called at all.

The *atomic_dec_and_test()* defined in *include/asm/atomic.h* resolves that potential problem. It atomically decrements the reference counter and returns *true* only if the decrement operation produced 0 .

```
static inline void kfree_skb(struct sk_buff *skb)
{
    if (atomic_read(&skb->users) == 1 ||
              atomic_dec_and_test(&skb->users))
         __kfree_skb(skb);
}
```

The *__kfree_skb()* function

The *__kfree_skb()* function used to ensure that the *sk_buff()* does not belong to any buffer list. It appears that is no longer deemed necessary.

The *dst_entry* entity is also reference counted. The *struct rtable* will actually be released only if this buffer holds the last reference. The call to the *destructor()* function adjusts the amount of *sndbuf* space allocated to *struct sock* that owns the buffer.

```
void __kfree_skb(struct sk_buff *skb)
{
   dst_release(skb->dst);
#ifdef CONFIG_XFRM
   secpath_put(skb->sp);
#endif
   if (skb->destructor) {
        WARN_ON(in_irq());
        skb->destructor(skb);
   }
```

__kfree_skb also used to initialize the state of the *struct sk_buff* header via the *skb_headerinit* function. The *kfree_skbmem()* function releases all associated buffer storage including fragments. The *struct sk_buff* used to be returned to the current processor's pool unless the pool is already full in which case it was returned the cache. Pools seem to have gone away.

```
  kfree_skbmem(skb);
}
```

Freeing the the data and the header with *kfree_skbmem()*

The *kfree_skbmem()* function invokes *skb_release_data()* to the free the data. It used to call *skb_head_to_pool* to return the *struct sk_buff* to the per-processor cache. Now a complex set of operations regarding the *fclone* state are performed.

```
void kfree_skbmem(struct sk_buff *skb)
{
        struct sk_buff *other;
        atomic_t *fclone_ref;

        skb_release_data(skb);
        switch (skb->fclone) {
```

Recall that the possible settings of the *skb->fclone* flag are:

```
enum {
        SKB_FCLONE_UNAVAILABLE,
        SKB_FCLONE_ORIG,
        SKB_FCLONE_CLONE,
};
```

If the buffer didn't come from the *fclone* cache, its flag will be set to *SKB_FCLONE_UNAVAILABLE*. If the buffer is the parent and did come from the *fclone* cache. The flag will be set to *SKB_FCLONE_ORIG*. If the buffer is the child and came from the *fclone* cache, the flag will be set to *SKB_FCLONE_UNAVAILABLE* if the buffer is available for use, but it will be set to *SKB_FCLONE_CLONE* if the buffer is in use. An available buffer will never be freed. Therefore, if the flag says *SKB_FCLONE_UNAVAILABLE,* then this is a standalone buffer not on from the *fclone* cache. Simple, no? To have reached this point in the code *skb->users* is guaranteed to be 1. So no further testing is needed.

```
        case SKB_FCLONE_UNAVAILABLE:
                kmem_cache_free(skbuff_head_cache, skb);
                break;

```

This is the parent of the two buffer pair. The atomic variable following the child counts total references to the parent and child. (It was set to one when the parent was allocated but before any cloning has taken place. Freeing the parent implicitly frees the child clone, and we don't know whether the parent or the child will be freed first. Therefore, the unnamed atomic variable following the child must be 1 in order to free the parent. Since this atomic variable has no name it is somewhat difficult to find all references to it.

```
        case SKB_FCLONE_ORIG:
                fclone_ref = (atomic_t *) (skb + 2);
                if (atomic_dec_and_test(fclone_ref))
                     kmem_cache_free(skbuff_fclone_cache, skb);
                break;

```

This is the child clone. It is made available for cloning again by just resetting the *fclone* flag to FCLONE_UNAVAILABLE. But if the parent has already been freed, then freeing the child will cause a "real" free.

```
        case SKB_FCLONE_CLONE:
                fclone_ref = (atomic_t *) (skb + 1);
                other = skb - 1;

                /* The clone portion is available for
                 * fast-cloning again.
                 */
                skb->fclone = SKB_FCLONE_UNAVAILABLE;

                if (atomic_dec_and_test(fclone_ref))
                   kmem_cache_free(skbuff_fclone_cache, other);
                break;
        };
}
```

Releasing unmapped page buffers, the fragment list, and the kmalloc'd area

The *skb_release_data()* function calls *put_page()* to free any unmapped page buffers, *skb_drop_fraglist()* to free the fragment chain, and then calls *kfree()* to free the *kmalloc'ed* component that normally holds the complete packet.

The data may be released only when *it is assured that no entity holds a pointer to the data.* If the cloned flag is *not* set it is assumed that whoever is attempting to free the *sk_buff* header is the only entity that held a pointer to the data.

If the cloned flag is set, the *dataref* reference counter controls the freeing of the data. Unfortunately the *dataref* field has now been split into two bitfields. It is shown in the *skb_clone()* function that the *cloned* flag is set in the header of both the original buffer and the clone when an *sk_buff* is cloned.

We divide dataref into two halves. The higher 16 bits hold references to the payload part of skb->data. The lower 16 bits hold references to the entire skb->data. It is up to the users of the skb to agree on where the payload starts. All users must obey the rule that the skb->data reference count must be greater than or equal to the payload reference count. Holding a reference to the payload part means that the user does not care about modifications to the header part of skb->data.

```
static void skb_release_data(struct sk_buff *skb)
{
        if (!skb->cloned ||
            !atomic_sub_return(skb->nohdr ? (1 <<
                                 SKB_DATAREF_SHIFT) + 1 : 1,
                                  &skb_shinfo(skb)->dataref)) {
            if (skb_shinfo(skb)->nr_frags) {
                int i;
                for (i = 0; i < skb_shinfo(skb)->nr_frags; i++)
                   put_page(skb_shinfo(skb)->frags[i].page);
            }

            if (skb_shinfo(skb)->frag_list)
                     skb_drop_fraglist(skb);

            kfree(skb->head);
        }
}
```

The old version of *skb_release_data*

```
static void skb_release_data(struct sk_buff *skb)
{
    if (!skb->cloned ||
        atomic_dec_and_test(&(skb_shinfo(skb)->dataref))){
         if (skb_shinfo(skb)->nr_frags) {
             int i;
             for (i = 0; i < skb_shinfo(skb)->nr_frags;i++)
                 put_page(skb_shinfo(skb)->frags[i].page);
         }

         if (skb_shinfo(skb)->frag_list)
                 skb_drop_fraglist(skb);

         kfree(skb->head);
    }
}
```

Releasing the fragment list

The *skb_drop_fraglist()* is defined in net/core/skbuff.c. It frees the *sk_buffs* in the *frag_list* by recursively calling *kfree_skb()*.

```

static void skb_drop_list(struct sk_buff **listp)
{
        struct sk_buff *list = *listp;

        *listp = NULL;

        do {
                struct sk_buff *this = list;
                list = list->next;
                kfree_skb(this);
        } while (list);
}

static inline void skb_drop_fraglist(struct sk_buff *skb)
{
        skb_drop_list(&skb_shinfo(skb)->frag_list);
}
```

Question: How does the loop termination logic work?

Freeing the *struct sk_buff* the old way.

The *skb_head_to_pool()* function releases the *sk_buff* structure. Whether the *sk_buff* is returned to the cache or placed on the per processor *hot list* depends upon the present length of the hot list queue. Recall that the *rmem, wmem* quotas also live in /proc/sys/net/core.

```
/proc/sys/net/core ==> cat hot_list_length
128
```

```
static __inline__ void skb_head_to_pool(
                                struct sk_buff *skb)
{
   struct sk_buff_head *list =
            &skb_head_pool[smp_processor_id()].list;

   if (skb_queue_len(list) < sysctl_hot_list_len) {
       unsigned long flags;

       local_irq_save(flags);
       __skb_queue_head(list, skb);
      local_irq_restore(flags);

       return;
   }
   kmem_cache_free(skbuff_head_cache, skb);
}
```

The write buffer destructor function

When the destructor function *sock_wfree()* is invoked, it decrements the *wmem_alloc* counter by the *truesize* field and will wake up a process that is sleeping on the socket if appropriate.

The call to *sock_put()* undoes the call to *sock_hold()* made in *skb_set_owner_w()* indicating the *sk_buff* no longer holds a pointer to the *struct sock,* The value of *sk->use_write_queue* is set to 1 by TCP but is not set by UDP. Therefore, *sock_def_write_space will be called for a UDP socket.*

```
void sock_wfree(struct sk_buff *skb)
{
    struct sock *sk = skb->sk;

 /* In case it might be waiting for more memory. */
    atomic_sub(skb->truesize, &sk->sk_wmem_alloc);
    if (!sock_flag(sk, SOCK_USE_WRITE_QUEUE))
            sk->sk_write_space(sk);
    sock_put(sk);
}
```

The *sock_put* function

Since the *struct socket* also holds a pointer to the *struct sock* this will alway be just a decrement when *sk_buffs* are being freed. If *sk_refcnt* were to equal 1 when called by *sock_wfree()* it would be a catastrophic failure!!!

```
static inline void sock_put(struct sock *sk)
{
        if (atomic_dec_and_test(&sk->sk_refcnt))
                sk_free(sk);
}
```

Waking a process sleeping on *wmem.*

The default write space function is *sock_def_write_space().* It will not attempt to wake up a waiting process until at least half of the *sndbuf* space is free. It also has to ensure that there is a sleeping process before a wakeup is attempted.

```
static void sock_def_write_space(struct sock *sk)
{
     read_lock(&sk->sk_callback_lock);
  /* Do not wake up a writer until he can make "significant"
   * progress.  --DaveM
  */
  if((atomic_read(&sk->sk_wmem_alloc) << 1) <= sk->sk_sndbuf) {
     if (sk->sk_sleep && waitqueue_active(sk->sk_sleep))
          wake_up_interruptible(sk->sk_sleep);

  /* Should agree with poll, otherwise some programs break */
     if (sock_writeable(sk))
              sk_wake_async(sk, 2, POLL_OUT);
   }

       read_unlock(&sk->sk_callback_lock);
}
```

Device driver allocation of *sk_buffs*

Whereas transport protocols must allocate buffers for transmit traffic, it is necessary for device drivers to allocate the buffers that will hold received packets. The *dev_alloc_skb()* function defined in *linux/skbuff.h* is used for this purpose. The *dev_alloc_skb()* is often called in the context of a hard or soft IRQ and thus *must use* *GFP_ATOMIC* to indicate that sleeping is not an option if the buffer cannot be allocated.

```
static inline struct sk_buff *dev_alloc_skb(unsigned int
                                                  length)
{
     return __dev_alloc_skb(length, GFP_ATOMIC);
}
```

According to comments in the code the reservation of 16 bytes (NET_SKB_PAD) of headroom is done for (presumably cache) optimizations.... not for header space.

```
static inline struct sk_buff *__dev_alloc_skb(unsigned int
                                             length,
                                             gfp_t gfp_mask)
{
     struct sk_buff *skb = alloc_skb(length + NET_SKB_PAD,
                                            gfp_mask);
     if (likely(skb))
          skb_reserve(skb, NET_SKB_PAD);
     return skb;
}
```

Accounting for the allocation of receive buffer space.

A device driver will not call *skb_set_owner_r()* because it does not know which *struct sock* will eventually own the *sk_buff*. However, when a received *sk_buff* is eventually assigned to a *struct sock, skb_set_owner_r()* will be called.

Interestingly, unlike *skb_set_owner_w(), The skb_set_owner_r()* function does not call *sock_hold()* even though it *does* hold a pointer to the *struct sock.* This seems to set up the possibility of an ugly race condition if a socket is closed about the time a packet is received.

```
static inline void skb_set_owner_r(struct sk_buff *skb,
                                   struct sock *sk)
{
        skb->sk = sk;
        skb->destructor = sock_rfree;
        atomic_add(skb->truesize, &sk->sk_rmem_alloc);
}

void sock_rfree(struct sk_buff *skb)
{
      struct sock *sk = skb->sk;

      atomic_sub(skb->truesize, &sk->sk_rmem_alloc);
}

```

Sharing and cloning of *sk_buffs*

There are two related mechanisms by which multiple entities may hold pointers to an *sk_buff* structure or the data it describes. An *sk_buff* is said to be *shared* when more than one process holds a pointer to the *struct sk_buff*. Sharing is controlled by the *skb->users* counter. A buffer may not actually be freed until the use count reaches 0. A buffer is shared via a call to *skb_get().*

Shared buffers must be assumed to be read-only. Specifically, very bad things will happen if two entities that share a buffer try to put the buffer on different queues!!!

```
static inline struct sk_buff *skb_get(struct sk_buff *skb)
{
        atomic_inc(&skb->users);
        return skb;
}
```

As seen previously, the *kfree_skb()* function will actually free a buffer only when called by the last user that holds a reference to the *struct sk_buff.*

Cloned buffers

In contrast, a *cloned* buffer is one in which multiple *struct skbuff* headers reference a single data area. A cloned header is indicated by setting the *skb->cloned* flag. The number of users of the shared data area is counted by the *dataref* element of the *skb_shared_info* structure. Cloning is necessary when multiple users of the same buffer need to make changes to the *struct sk_buff*. For example, a reliable datagram protocol needs to retain a copy of an *sk_buff* that has been passed to the *dev layer* for transmission. Both the transport protocol and the *dev* layer may need to modify the *skb->next* and *skb->prev* pointers.

Creating a clone of an *sk_buff*.

The *skb_clone()* function is defined in net/core/skbuff.c. It duplicates the *struct sk_buff* header, but the data portion remains shared. The use count of the clone to be set to one. If memory allocation fails, NULL is returned . The ownership of the new buffer is not assigned to any *struct sock*. If this function is called from an interrupt handler *gfp_mask* must be GFP_ATOMIC.

```
struct sk_buff *skb_clone(struct sk_buff *skb, gfp_t gfp_mask)
{
        struct sk_buff *n;

```

The pointer *n* is optimistically set to the address of the *fclone*. The test for SKB_FCLONE_ORIG ensures that a broken attempt to *fclone* a buffer from the standard cache will NOT be attempted. If a successful *fclone* occurs then the unnamed *atomic_t* variable following the fclone will become 2.

Here when the buffer is allocated from the *skbuff_head_cache* the *fclone* flag is explicitly set to 0. The *fclone* flag is a two bit bitfield.

```
        n = skb + 1;
        if (skb->fclone == SKB_FCLONE_ORIG &&
            n->fclone == SKB_FCLONE_UNAVAILABLE) {
            atomic_t *fclone_ref = (atomic_t *) (n + 1);
                   n->fclone = SKB_FCLONE_CLONE;
                   atomic_inc(fclone_ref);
        } else {
            n = kmem_cache_alloc(skbuff_head_cache, gfp_mask);
            if (!n)
                 return NULL;
            n->fclone = SKB_FCLONE_UNAVAILABLE;
       }

```

The rest of the function deals with copying specific fields one at time. Why not use *memcpy* and then override the fields that we don't want copied?

```
#define C(x) n->x = skb->x

```

Clone lives on no list and has now owner socket.

```
        n->next = n->prev = NULL;
        n->sk = NULL;
        C(tstamp);
        C(dev);
        C(h);
        C(nh);
        C(mac);
        C(dst);
        dst_clone(skb->dst);
        C(sp);

#ifdef CONFIG_INET
        secpath_get(skb->sp);
#endif
        memcpy(n->cb, skb->cb, sizeof(skb->cb));
        C(len);
        C(data_len);
        C(csum);
        C(local_df);
        n->cloned = 1;
        n->nohdr = 0;
        C(pkt_type);
        C(ip_summed);
        C(priority);
```

```
#if defined(CONFIG_IP_VS) || defined(CONFIG_IP_VS_MODULE)
        C(ipvs_property);
#endif
        C(protocol);
```

Clone must not have a destructor to avoid "double credit" for freeing data. For proper accounting in a realiable protocol, the clone not the original must be passed down the stack for transmission because the original will necessarily be freed last. If multiple retransmissions are required, a new clone must be created for each retransmission.

However, if *fcloning* is in use the new clone just recycle the *fclone* because it will have already been freed by the time the retransmission occurs.

```
        n->destructor = NULL;
#ifdef CONFIG_NETFILTER
        C(nfmark);
        C(nfct);
        nf_conntrack_get(skb->nfct);
        C(nfctinfo);
#if defined(CONFIG_NF_CONNTRACK) ||
       defined(CONFIG_NF_CONNTRACK_MODULE)
        C(nfct_reasm);
        nf_conntrack_get_reasm(skb->nfct_reasm);
#endif
#ifdef CONFIG_BRIDGE_NETFILTER
        C(nf_bridge);
        nf_bridge_get(skb->nf_bridge);
#endif
#endif /*CONFIG_NETFILTER*/
#ifdef CONFIG_NET_SCHED
        C(tc_index);
#ifdef CONFIG_NET_CLS_ACT
        n->tc_verd = SET_TC_VERD(skb->tc_verd,0);
        n->tc_verd = CLR_TC_OK2MUNGE(n->tc_verd);
        n->tc_verd = CLR_TC_MUNGED(n->tc_verd);
        C(input_dev);
#endif
        skb_copy_secmark(n, skb);
#endif
```

```
        C(truesize);
        atomic_set(&n->users, 1);
        C(head);
        C(data);
        C(tail);
        C(end);

        atomic_inc(&(skb_shinfo(skb)->dataref));
        skb->cloned = 1;

        return n;
}
```

Converting a shared buffer to a clone

The *skb_share_check()* function, defined in *include/linux/skbuff.h,* clones a shared *sk_buff.* After the cloning takes place, the call to *kfree_skb()* decrements *skb->users* on the original copy. A shared buffer necessarily has a use count exceeding one, and so the call to *kfree_skb()* simply decrements it.

```
static inline struct sk_buff *skb_share_check(
                                           struct sk_buff *skb,
                                           gfp_t pri)
{
        might_sleep_if(pri & __GFP_WAIT);
        if (skb_shared(skb)) {
                struct sk_buff *nskb = skb_clone(skb, pri);
                kfree_skb(skb);
                skb = nskb;
        }
        return skb;
}
```

The *skb_shared()* inline function returns TRUE if the number of users of the buffer exceeds 1.

```
static inline int skb_shared(struct sk_buff *skb)
{
        return (atomic_read(&skb->users) != 1);
}
```

Obtaining a buffer from one of the *per processor* pools

The *skb_head_from_pool()* function used to provide buffers from a fast access per CPU cache. It detaches and returns the first *sk_buff* header in the list or returns NULL if the list is empty. Interrupt disablement instead of locking can be used because and only because the pool is local to the processor.

```
static __inline__ struct sk_buff
                  *skb_head_from_pool(void)
{
    struct sk_buff_head *list =
         &skb_head_pool[smp_processor_id()].list;

    if (skb_queue_len(list)) {
         struct sk_buff *skb;
         unsigned long flags;

         local_irq_save(flags);
         skb = __skb_dequeue(list);
         local_irq_restore(flags);
         return skb;
    }
    return NULL;
}
```

Non-linear buffers

Non-linear *sk_buffs* are those consisting of unmapped page buffers and additional chained *struct sk_buffs*. Probably 1/2 of the network code in the kernel is dedicated to dealing with the rarely used abomination. A non-zero value of *data_len* is an indicator of non-linearity. For obvious reasons the simple *skb_put()* function neither supports nor tolerates non-linearity. SKB_LINEAR_ASSERT checks value of *data_len* through function *skb_is_nonlinear*. A non-zero value results in an error message to be logged by BUG.

```
#define SKB_LINEAR_ASSERT(skb)
        do { if (skb_is_nonlinear(skb)) BUG(); } while (0)
```

Trimming non-linear buffers

The real trim function is *___pskb_trim()* function which is defined in net/core/skbuff.c. It gets really ugly really fast because it must deal with unmapped pages and buffer chains.

```
   /*   Trims skb to length len. It can change skb
        pointers if "realloc" is 1. If realloc == 0 and
        trimming is impossible without change of data,
        it is BUG().
    */

int ___pskb_trim(struct sk_buff *skb, unsigned int
                  len, int realloc)
{
```

The value of *offset* denotes length of the *kmalloc'd* component of the *sk_buff.*

```
        int offset = skb_headlen(skb);
        int nfrags = skb_shinfo(skb)->nr_frags;
        int i;

```

This loop processes any unmapped page fragments that may be associated with the buffer.

```
    for (i=0; i<nfrags; i++) {
```

Add the fragment size to *offset* and compare it against the length of the IP packet. If *end* is greater than *len*, then this fragment needs to be trimmed. In this case, if the *sk_buff* is a clone, its header and *skb_shared_info* structure are reallocated here.

```
         int end = offset +
                   skb_shinfo(skb)->frags[i].size;
         if (end > len) {
              if (skb_cloned(skb)) {
                   if (!realloc)
                        BUG();
                   if (!pskb_expand_head(skb, 0, 0,
                             GFP_ATOMIC))
                        return -ENOMEM;
              }
```

If the offset of the start of the fragment lies beyond the end of the data, the fragment is freed and number of fragments decremented by one. Otherwise, the fragment size is decremented so that its length is consistent with the size of the packet.

```
              if (len <= offset) {
                   put_page(skb_shinfo(skb)
                                  ->frags[i].page);
                   skb_shinfo(skb)->nr_frags--;
              } else {
                   skb_shinfo(skb)->frags[i].size
                                   = len-offset;
              }
         }
```

Update *offset* so that it reflects the offset to the start position of the next fragment.

```
         offset = end;
    }
```

After processing the unmapped page fragments, some additional adjustments may be necessary. Here *len* holds the target trimmed length and *offset* holds the offset to the first byte of data beyond the unmapped page fragments. Since *skb->len* is greater than *len* it is not clear how *offset* can be smaller than *len.*

```
    if (offset < len) {
        skb->data_len -= skb->len - len;
        skb->len = len;
    }
```

If *len* <= *skb_headlen(skb)* then all of the data now resides in the kmalloc'ed portion of the *sk_buff.* If the *sk_buff* is not cloned then presumably *skb_drop_fraglist()* frees the now unused elements.

```
        else {
        if (len <= skb_headlen(skb)) {
            skb->len = len;
            skb->data_len = 0;
            skb->tail = skb->data + len;
            if (skb_shinfo(skb)->frag_list &&
                                            !skb_cloned(skb))
                skb_drop_fraglist(skb);
        }
```

In this case the *offset* is greater than or equal to *len.* The trimming operation is achieved by decrementing *skb->data_len* by the amount trimmed and settting *skb->len* to the target length.

```
        else {
            skb->data_len -= skb->len - len;
            skb->len = len;
        }
    }

    return 0;
}
```

Miscellaneous buffer management functions

The *skb_cow()* function is defined in *include/linux/skbuff.h.* It ensures that the headroom of the *sk_buff* is at least 16 bytes. The *sk_buff* is reallocated if its headroom is inadequate of small or if it has a clone. Recall that *dev_alloc_skb()* used *skb_reserve()* to establish a 16 byte headroom when the packet was allocated. Thus for the ``normal'' case the value of *delta* will be 0 here.

```
static inline int
skb_cow(struct sk_buff *skb, unsigned int headroom)
{
        int delta = (headroom > 16 ? headroom : 16)
                               - skb_headroom(skb);

        if (delta < 0)
                delta = 0;
```

When the headroom is small or the *sk_buff* is cloned, reallocate the *sk_buff* with specified headroom size.

```
        if (delta || skb_cloned(skb))
                return pskb_expand_head(skb,
                     (delta+15) & ~15, 0, GFP_ATOMIC);
        return 0;

}
```