Skbuffs - A tutorial

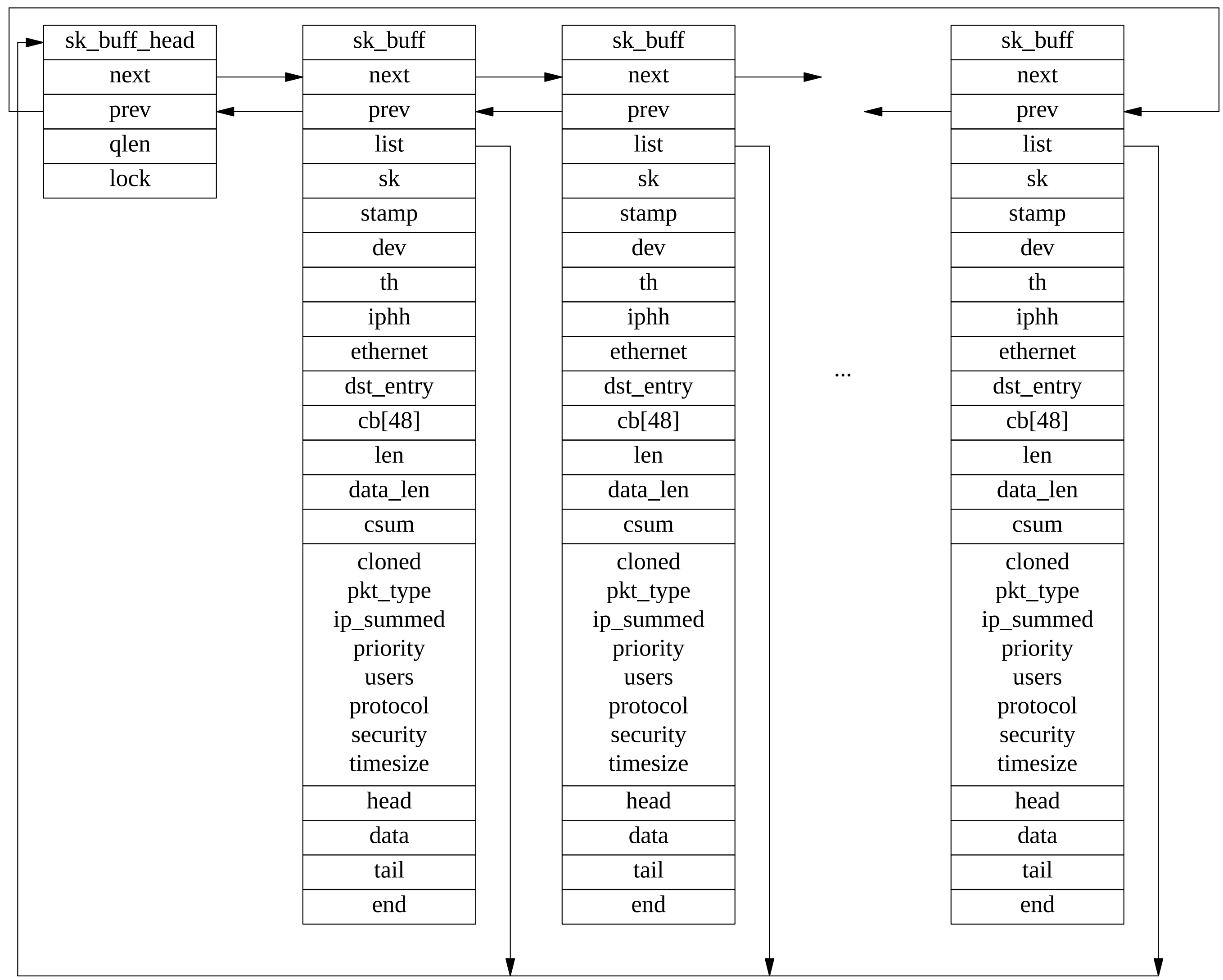
ri-oa

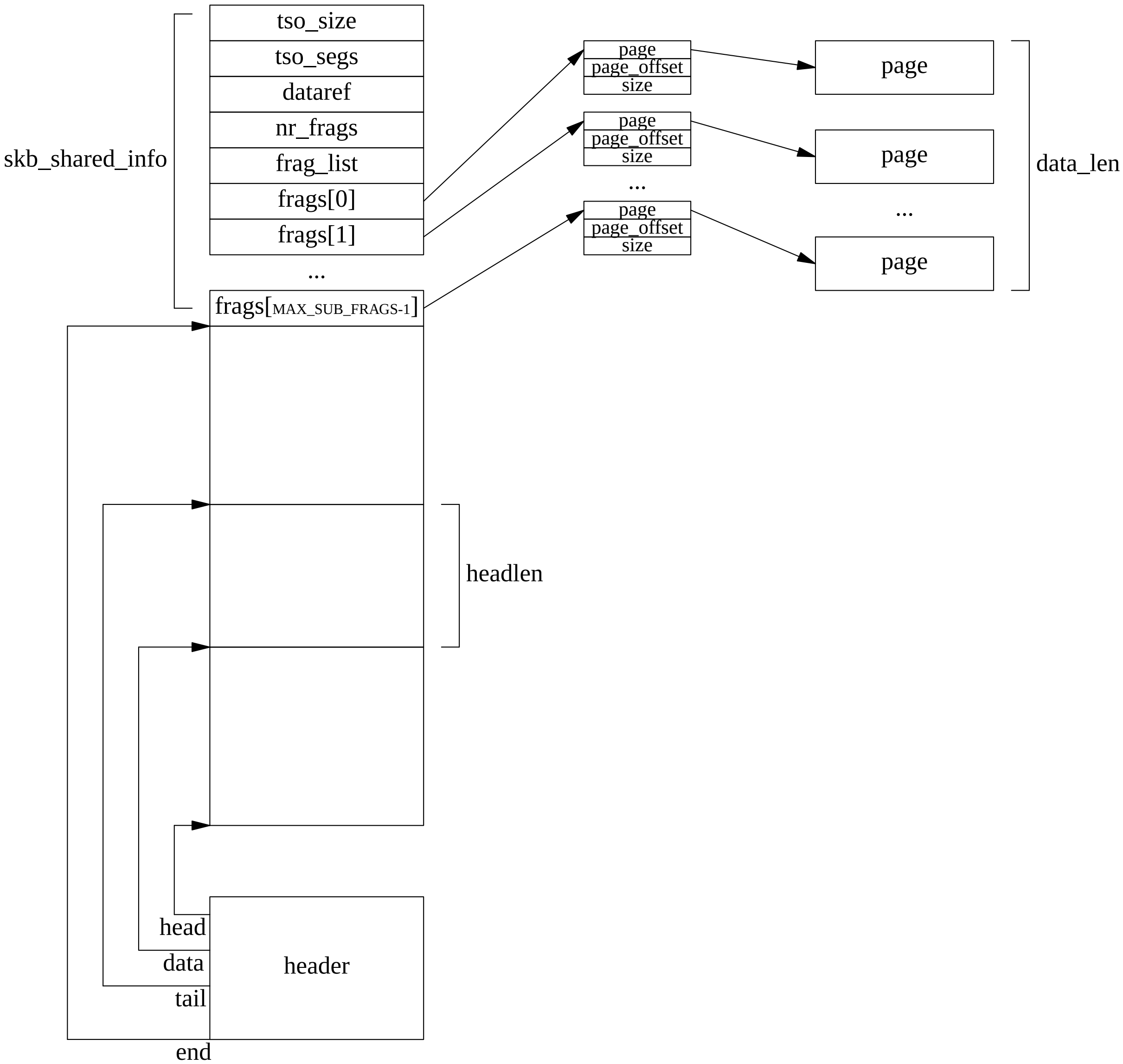
sissa

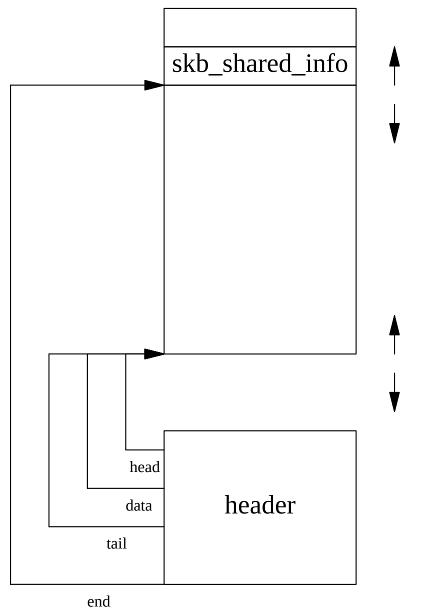
1. Introduction

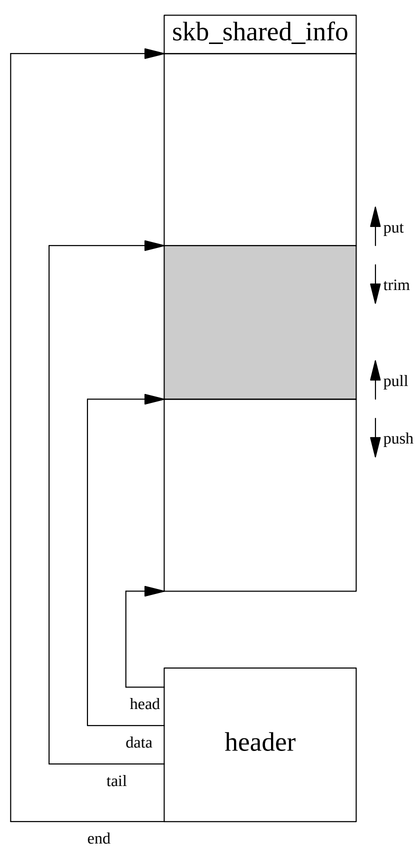
The skbuff system is a memory management facility explicitly thought for the network code. It is a small software layer that makes use of the general memory allocation facilities offered by the Linux kernel. Starting with kernel 2.2 the Linux kernel introduced the slab system for allocation of small memory areas and eventually the caching of information between allocations for specific structures. The slab allocator keeps continguous physical memory for each slab.

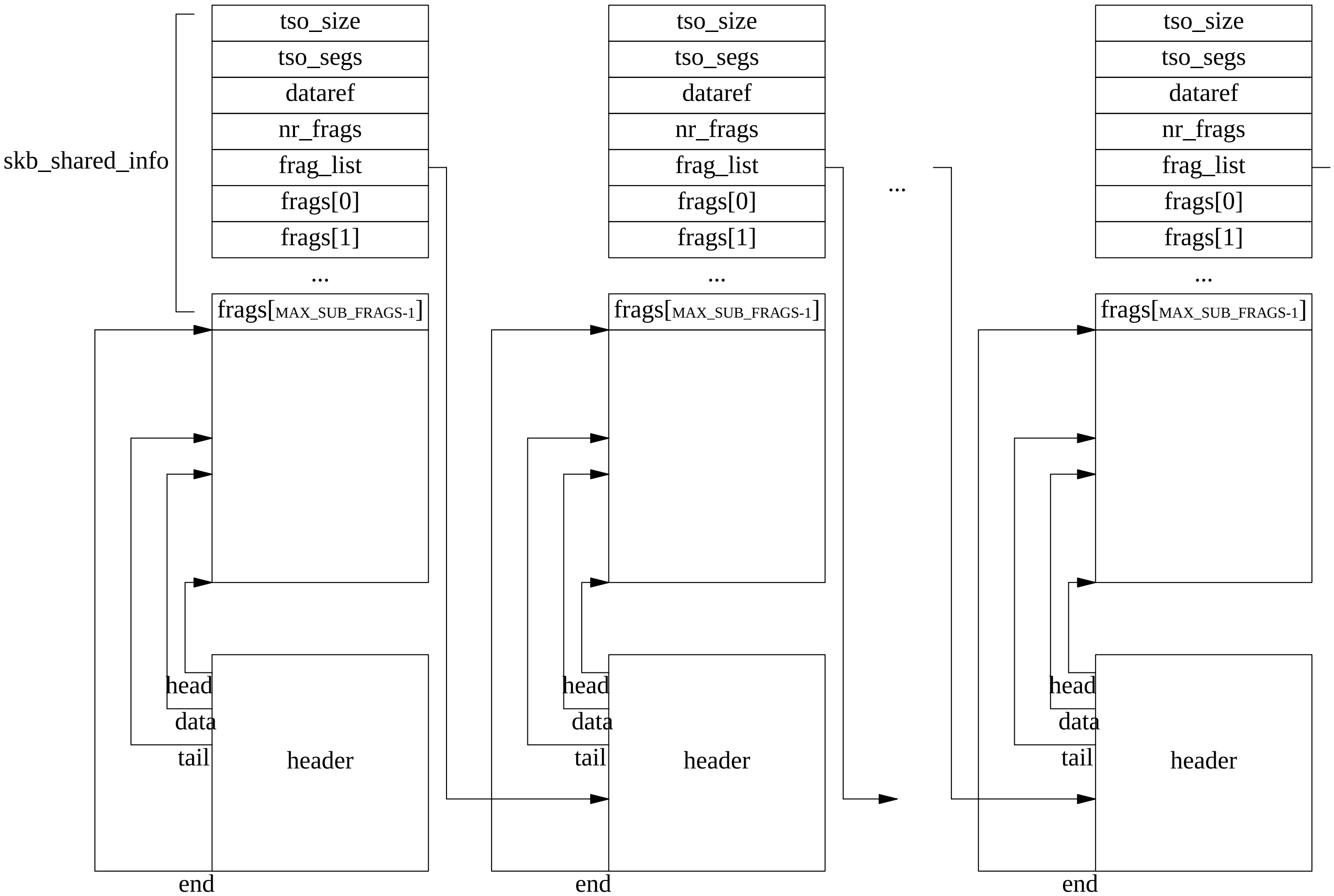
Skbuff network buffer layer




skb_shared_info
tso_size
tso_segs
dataref
nr_frags
frag_list
frags[0]
frags[1]
...
frags[MAX_SUB_FRAGS-1]
page
page_offset
size
page
page_offset
size
...
page
page_offset
size
page
page
...
page
data_len
headlen
head
data
tail
end
header


skb_shared_info
head
data
tail
end
header


skb_shared_info
put
trim
pull
push
head
data
tail
end
header



The skbuff system uses in two ways the slab allocator.[1] The skbuff heads are allocated from a named slab called `skbuff_head_cache.`

[net/core/skbuff.c]

```
void __init skb_init(void)
{
        skbuff_head_cache = kmem_cache_create("skbuff_head_cache",
                                              sizeof(struct sk_buff),
                                              0,
                                              SLAB_HWCACHE_ALIGN,
                                              NULL, NULL);
        if (!skbuff_head_cache)
                panic("cannot create skbuff cache");
}
```

[net/core/skbuff.c]

Instead the data areas of skbuffs, not being able of taking any advantage from the previous allocations are allocated from size-N generic slabs using the `kmalloc()` function.

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

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

        /* 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;
```

---------------------- comment : i think is better to call descriptor the fixed size part of the skbuff that keeps pointers to data areas and control info (the sk_buff). and skbuff list head the sk_buff_head structure. -- Since the paper many things have changed in the skbuff system. The two most important changes are : - now skbuffs can be nonlinear and stored in two different ways :

- in fragments as an array of pages using the skb_shared_info.frags
  array (now it can have enough pages for 64KB + 2pages)
- in fragments as a list of sk_buff using the skb_shared_info.frag_list
  pointer - the interface with the kernel memory allocator has been completely

changed and now you can use named slabs for structures that
can get benefits from caching freed objects of the same kind

----

2. Constrains imposed by Network APIs and hardware interfaces

- Copy semantic of standard Unix network calls - Network cards usually unable to perform gather/scatter One feature that distinguish some network cards from others is the possibility to perform scatter/gather operations. In this case the frame to be transmitted can lay fragmented in noncontiguous areas of memory and the card is able to collect them based on a list of pointers and to transmit it (gather). The same can happen when a frame is received, even if this feature is less used. The different headers of the frame can be deposited in different noncontiguous areas (scatter). Most of the cheap PC cards never supported this features. Therefore it is required to prepare the complete frame to transmit in a physical contiguous area of memory.

3. Fundamental data structures

File : `[include/linux/skbuff.h]`

3.1. sk_buff

The most important data structure is the `sk_buff.` This is the skbuff header where all the status information for a linear skbuff are kept. Every skbuff has an sk_buff structure that holds all the pointers to the data areas. The skbuff header is allocated from the relative memory slab. The skbuffs are moved from queues at socks to/from queues at devices.
This is done through the use of the `next` and `prev` pointers that can link skbuffs in a doubly linked list. The head of the list where they are linked to is pointed to by the `list` pointer. This head can be the send/receive queue at the sock/device.
The sock, if any, associated with the skbuff is pointed to by the `sk` pointer.
And the device from where the data arrived or is leaving by is pointed to by the `dev` and `real_dev` pointers. The real_dev is used for example in bonding and VLAN drivers. In bonding, when a packet is

received, if the device on which it is received has a master, then the real_dev is set to the dev contents and the dev field is set to the master device :

[net/core/dev.c]

```
/* Deliver skb to an old protocol, which is not threaded well
   or which do not understand shared skbs.
 */
static int deliver_to_old_ones(struct packet_type *pt,
                               struct sk_buff *skb, int last)
{
        int ret = NET_RX_DROP;

        if (!last) {
```

[net/core/dev.c]

Pointers to the transport header(tcp/udp) `h` , network layer header (ip) `nh,` link layer header `mac` are filled as soon as known.
A pointer to a destination cache entry is kept in `dst.` Security information (keys and so on) for IPSec are pointed to by the `sp` pointer.
A free area of 48 bytes called control block (`cb`) is left for specific protocol layers necessities (that area can be used to pass info between protocol layers).
The `len` field keeps the length in bytes of the data area, this is the total data area, encompassing also the eventual pages of data of a fragmented skbuff.
The `data_len` field is the length in bytes of the data area, not in the linear part of the skbuff. If this field is different from zero, then the skbuff is fragmented. The difference data_len`len`- is the amount of data in the linear part of the skbuff, and is also called `headlen` (not to be confused with the size of `headroom).` The `csum` field keeps the eventual checksum of the data. The `local_df` field is used to signal if the real path mtu discovery was requested or not. local_df == 1 means the IP_PMTUDISC_DO was not requested, local_df == 0 means the IP_PMTUDISC_DO was requested and so an icmp error should be generated if we receive fragments. The `cloned` field signals the skbuff has been cloned and so if a user wants to write on it, the skbuff should be copied. The `pkt_type` field describes the destination of the packet (for us, for someone else, broadcast, multicast ..) according to the following definitions :

[include/linux/if_packet.h]

```
/* Packet types */

#define PACKET_HOST       0         /* To us        */
#define PACKET_BROADCAST  1         /* To all       */
#define PACKET_MULTICAST  2         /* To group          */
#define PACKET_OTHERHOST  3         /* To someone else   */
#define PACKET_OUTGOING        4         /* Outgoing of any type */
/* These ones are invisible by user level */
#define PACKET_LOOPBACK        5         /* MC/BRD frame looped back */
#define PACKET_FASTROUTE  6         /* Fastrouted frame  */
```

[include/linux/if_packet.h]

The `ip_summed` field tells if the driver supplied an ip checksum. It can be NONE, HW or UNNECESSARY :

[include/linux/skbuff.h]

```

#define CHECKSUM_NONE 0
#define CHECKSUM_HW 1
#define CHECKSUM_UNNECESSARY 2
```

[include/linux/skbuff.h]

On input, `CHECKSUM_NONE` means the device failed to checksum the packet and so csum is undefined, `CHECKSUM_UNNECESSARY` means that the checksum has already been verified, but the problem is that it is not known in which way (for example as an ipv6 or an ipv4 packet ..), so it is an unrecommended flag. `CHECKSUM_HW` means the device provides the checksum in the csum field. On output, `CHECKSUM_NONE` means checksum provided by protocol or not required, `CHECKSUM_HW` means the device is required to checksum the packet (from the header h.raw to the end of the data and put the checksum in the csum field). The `priority` field keeps the priority level according to :

[include/linux/pkt_sched.h]

```
#ifndef __LINUX_PKT_SCHED_H
#define __LINUX_PKT_SCHED_H

/* Logical priority bands not depending on specific packet scheduler.
   Every scheduler will map them to real traffic classes, if it has
   no more precise mechanism to classify packets.

   These numbers have no special meaning, though their coincidence
   with obsolete IPv6 values is not occasional :-). New IPv6 drafts
   preferred full anarchy inspired by diffserv group.

   Note: TC_PRIO_BESTEFFORT does not mean that it is the most unhappy
   class, actually, as rule it will be handled with more care than
   filler or even bulk.
 */

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

#define TC_PRIO_MAX            15

```

[include/linux/pkt_sched.h]

they are used by traffic control mechanisms.

The `security` field keeps the level of security.

The `truesize` field keeps the real size occupied by the skbuff, that is it adds the size of the header to the size of the data when the skbuff is allocate in `alloc_skb()` :

[net/core/skbuff.c]

```

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

[net/core/skbuff.c]

When a copy is made, the skbuff header is copied up to the truesize field, because the remaining fields are pointers to the data areas and so need to be replaced.

The `head,`end pointers, are pointers to the boundaries of the available space.

The `data,` tail pointers are pointers to the beginning and end of the already used data area.

[include/linux/skbuff.h]

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

     struct sk_buff_head  *list;
     struct sock          *sk;
     struct timeval       stamp;
     struct net_device    *dev;
     struct net_device    *real_dev;

     union {
          struct tcphdr   *th;
          struct udphdr   *uh;
          struct icmphdr  *icmph;
          struct igmphdr  *igmph;
          struct iphdr    *ipiph;
          unsigned char   *raw;
     } h;

     union {
          struct iphdr    *iph;
          struct ipv6hdr  *ipv6h;
          struct arphdr   *arph;
          unsigned char   *raw;
     } nh;

     union {
          struct ethhdr   *ethernet;
          unsigned char   *raw;
     } mac;

     struct  dst_entry    *dst;
     struct    sec_path   *sp;

     /*
      * 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 do a skb_clone()
      * first. This is owned by whoever has the skb queued ATM.
      */
     char            cb[48];

     unsigned int         len,
                     data_len,
                     csum;
     unsigned char        local_df,
                     cloned,
                     pkt_type,
                     ip_summed;
     __u32           priority;
```

```
        unsigned short       protocol,
                             security;

        void                 (*destructor)(struct sk_buff *skb);
#ifdef CONFIG_NETFILTER
           unsigned long          nfmark;
        __u32                nfcache;
        struct nf_ct_info    *nfct;
#ifdef CONFIG_NETFILTER_DEBUG
           unsigned int      nf_debug;
#endif
#if defined(CONFIG_BRIDGE) || defined(CONFIG_BRIDGE_MODULE)
        struct nf_bridge_info     *nf_bridge;
#endif
#endif /* CONFIG_NETFILTER */
#if defined(CONFIG_HIPPI)
        union {
             __u32     ifield;
        } private;
#endif
#ifdef CONFIG_NET_SCHED
          __u32              tc_index;               /* traffic control index */
#endif

        /* These elements must be at the end, see alloc_skb() for details.  */
        unsigned int         truesize;
        atomic_t             users;
        unsigned char        *head,
                             *data,
                             *tail,
                             *end;
};

```

[include/linux/skbuff.h]

3.2. skb_shared_info

The `skb_shared_info` structure is used by the fragmented skbuffs. It has a meaning when the `data_len` field in the skbuff header is different from zero. This field counts the data not in the linear part of the skbuff.

The `dataref` field counts the number of references to the fragmented part of the skbuff, so that a writer knows if it is necessary to copy it.

The `nr_frags` field keeps the number of pages in which this skbuff is fragmented. This kind of fragmentation is done for interfaces supporting scatter and gather. This feature is described in the netdevice structure by the NETIF_F_SG flag. (3com 3c59x , 3com typhoon, Intel e100, ...) When an skbuff is to be allocated, if the mss is larger than a page then if the interface supports scatter and gather a linear skbuff of a single page is allocated with alloc_skb and then the other pages are allocated and added to the frags array.

The `tso_size,tso_segs` fields were added to support cards able to perform by themselves the tcp segmentation (they are described by the NETIF_F_TSO TCP Segmentation Offload). The tso_size comes from the mss, and is the max size that should be used by the card for segments. (3Com Typhoon family 3c990, 3cr990 supports it if the array of pages is <= 32)

The `frag_list` pointer is used when the skbuff is fragmented in a list. This is eventually done when the

interface supports the NETIF_F_FRAG_LIST feature. There are no devices in the standard linux kernel tree that support this feature at the moment (except the trivial loopback).
The `frags` array keeps the pointers to the page structures in which the skbuff has been fragmented. The last used page pointer is `nr_frags-1` and there is space for up to MAX_SKB_FRAGS. This was only 6 in previous versions, now it is sufficient to accomodate a maximum length tcp segment (64 KB).

[include/linux/skbuff.h]

```
/* To allow 64K frame to be packed as single skb without frag_list */
#define MAX_SKB_FRAGS (65536/PAGE_SIZE + 2)
```

[include/linux/skbuff.h]

[include/linux/skbuff.h]

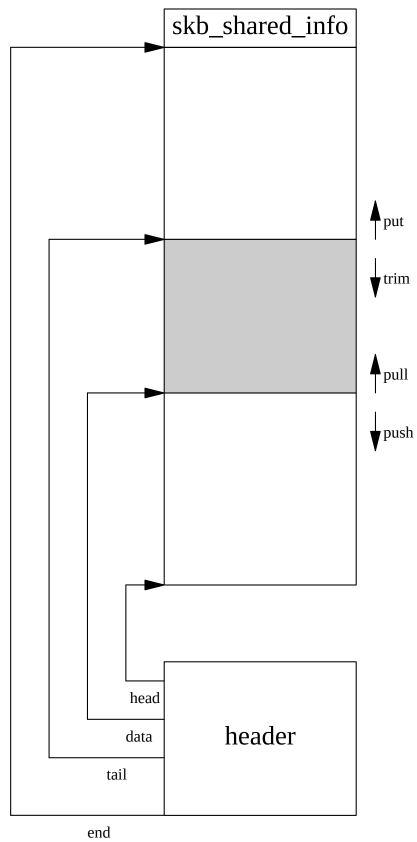
```
struct skb_shared_info {
     atomic_t  dataref;
     unsigned int   nr_frags;
     unsigned short  tso_size;
     unsigned short  tso_segs;
     struct sk_buff  *frag_list;
     skb_frag_t frags[MAX_SKB_FRAGS];
};
```

[include/linux/skbuff.h]

4. Skbuff organizations

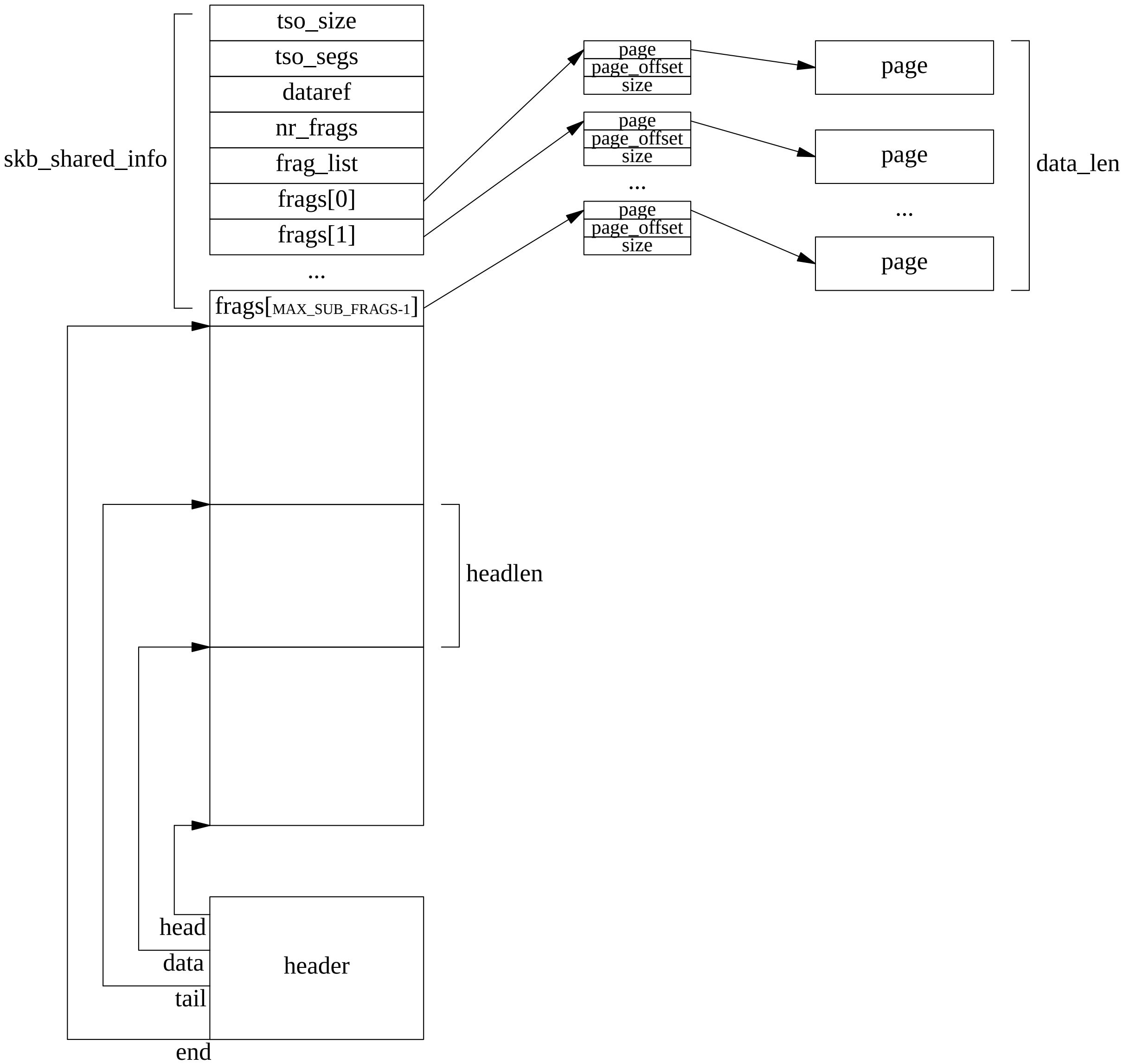
Until recently the data area of the skbuff was unique and physically contiguous. And the Linux kernel was cited because of the efficiency it could obtain with dumb interfaces against other popular OSs like bsd, in which frequently because of the small size of the network buffers, you could have a list of them for a single net packet.

4.1. Linear skbuffs

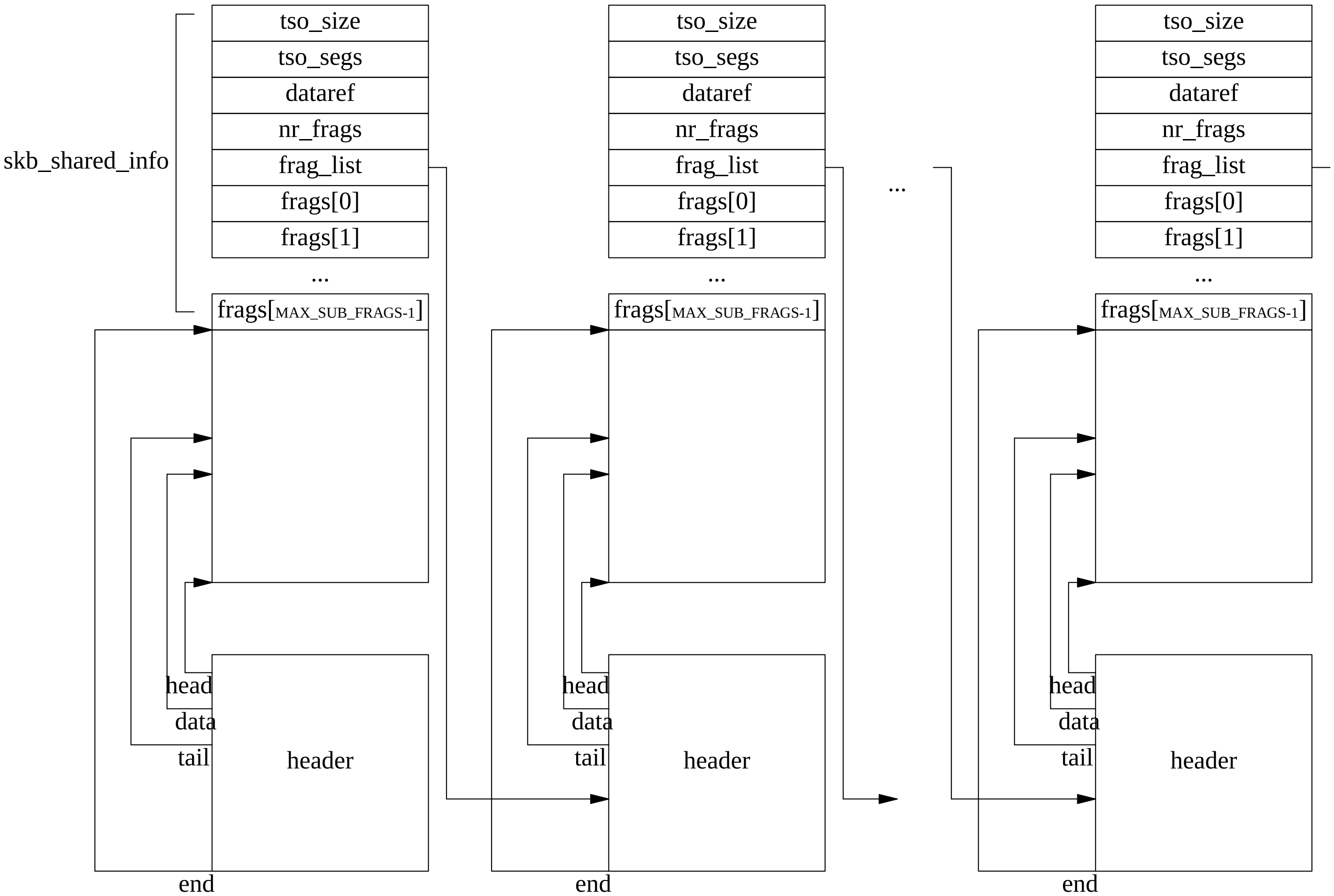


4.2. Nonlinear skbuffs

4.2.1. array of pages fragmentation

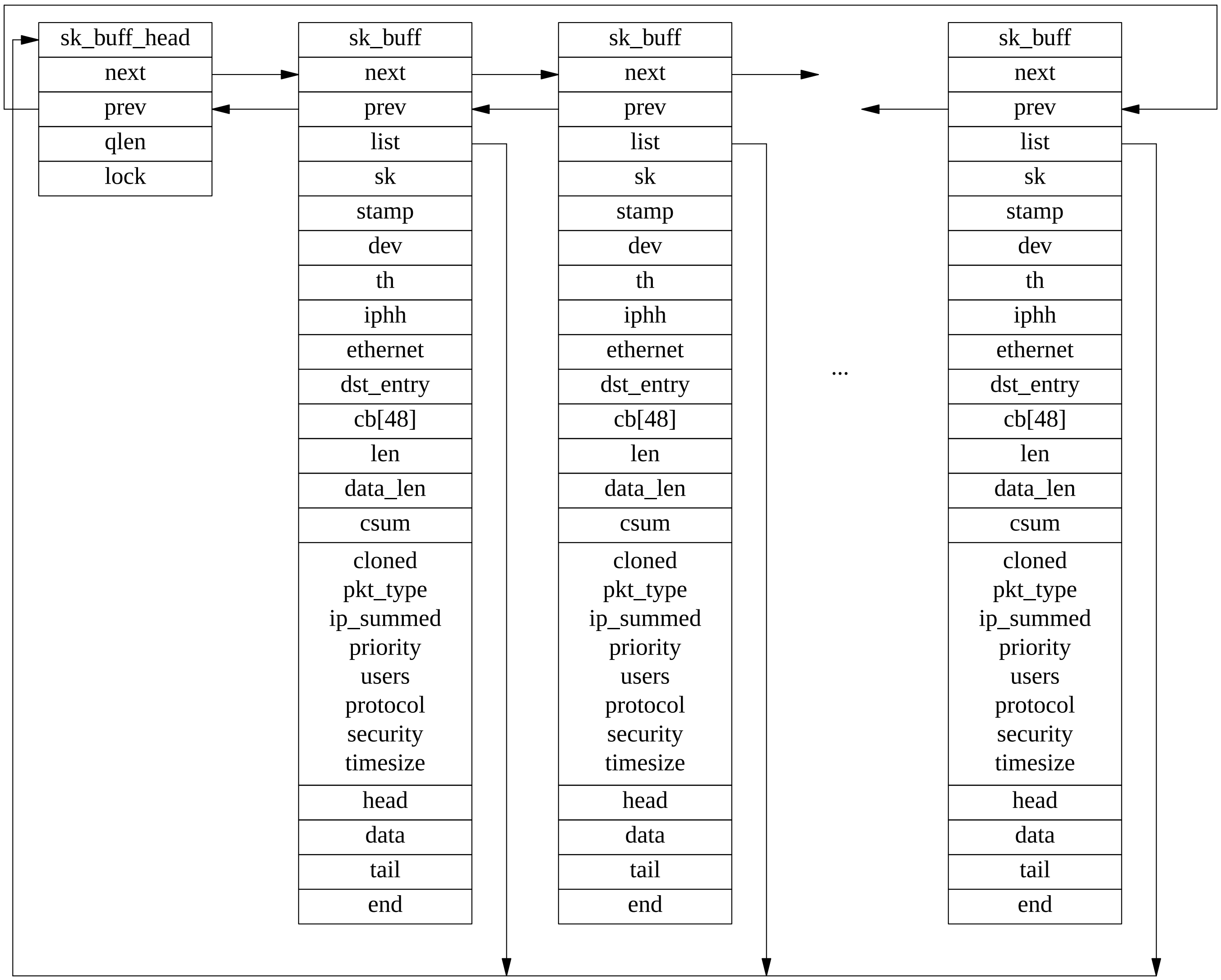


4.2.2. skbuff list fragmentation



5. Queues of skbuffs

In output skbuff is queued first on the socket, and then when the output interface is determined the skbuff is moved to the device queue. In input the skbuff is queued on the device queue and then when the owner socket is found it is moved to the owner socket queue.



5.1. Functions to manage skbuff queues

5.1.1. skb_queue_len

This function returns the number of skbuff queued on the list that you pass as an argument. *File :* `[include/linux/skbuff.h]`

[include/linux/skbuff.h]

```
static inline __u32 skb_queue_len(struct sk_buff_head *list_)
{
```

[include/linux/skbuff.h]

5.1.2. skb_queue_head_init

This function initializes a queue of skbuffs. It locks the list setting the list establishing a spin lock on its lock variable and sets the `prev` and `next` pointers to the list head. *File :* `[include/linux/skbuff.h]`

[include/linux/skbuff.h]

```
static inline void skb_queue_head_init(struct sk_buff_head *list)
{
        spin_lock_init(&list->lock);
        list->prev = list->next = (struct sk_buff *)list;
        list->qlen = 0;
}
```

[include/linux/skbuff.h]

5.1.3. skb_queue_head and __skb_queue_head

These two functions queue an skbuff buffer at the start of a list. The `skb_queue_head` function establishes a spin lock on the queue lock variable and so it is safe to be called with interrupts enable, it then calls the `__skb_queue_head` function to queue the buffer. The `__skb_queue_head` function can be called by itself only with interrupts disabled. *File :* `[include/linux/skbuff.h]`

[include/linux/skbuff.h]

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

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

[include/linux/skbuff.h]

[include/linux/skbuff.h]

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

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

[include/linux/skbuff.h]

5.1.4. skb_queue_tail and __skb_queue_tail

These two functions queue an skbuff buffer at the tail of a list. The `skb_queue_tail` function establishes a spin lock on the queue lock variable and so it is safe to be called with interrupts enable, it then calls the `__skb_queue_tail` function to queue the buffer. The `__skb_queue_tail` function can be called by itself only with interrupts disabled. *File :* `[include/linux/skbuff.h]`

[include/linux/skbuff.h]

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

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

[include/linux/skbuff.h]

[include/linux/skbuff.h]

```
static inline 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);
}
```

[include/linux/skbuff.h]

5.1.5. skb_dequeue and __skb_dequeue

These two functions dequeue an skbuff buffer from the head of a list. The `skb_dequeue` function establishes a spin lock on the queue lock variable and so it is safe to be called with interrupts enable, it then calls the `__skb_dequeue` function to dequeue the buffer. The `__skb_dequeue` function can be called by itself only with interrupts disabled. *File :* `[include/linux/skbuff.h]`

[include/linux/skbuff.h]

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

     spin_lock_irqsave(&list->lock, flags);
```

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

[include/linux/skbuff.h]

[include/linux/skbuff.h]

```
static inline struct sk_buff *__skb_dequeue(struct sk_buff_head *list)
{
        struct sk_buff *next, *prev, *result;

        prev = (struct sk_buff *) list;
        next = prev->next;
        result = NULL;
        if (next != prev) {
                result         = next;
                next      = next->next;
                list->qlen--;
                next->prev   = prev;
                prev->next   = next;
                result->next = result->prev = NULL;
                result->list = NULL;
        }
        return result;
}
```

[include/linux/skbuff.h]

5.1.6. skb_insert and __skb_insert

5.1.7. skb_append and __skb_append

5.1.8. skb_unlink and __skb_unlink

5.1.9. skb_dequeue_tail and __skb_dequeue_tail

6. Skbuff Functions

The following functions distinguish between the three kind of skbuffs : linear, fragmented in an array of additional pages, fragmented in a list of skbuffs.

6.1. SKB_LINEAR_ASSERT and skb_is_nonlinear

File : `[include/linux/skbuff.h]`
the SKB_LINEAR_ASSERT macro will raise a bug if the skb is nonlinear, this condition is checked

looking at the data_len field that reports the size of the data in the nonlinear part of the skbuff.

[include/linux/skbuff.h]

```
#define SKB_LINEAR_ASSERT(skb)  BUG_ON(skb_is_nonlinear(skb))
```

[include/linux/skbuff.h]

[include/linux/skbuff.h]

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

[include/linux/skbuff.h]

6.2. SKB_PAGE_ASSERT

File : [include/linux/skbuff.h]

This macro will raise a bug if the skbuff is fragmented in additional pages. We have already discussed that if the skbuff is fragmented in pages then the number of pages used is kept in the skb_shared_info structure, nr_frags variable.

[include/linux/skbuff.h]

```
#define SKB_PAGE_ASSERT(skb)   BUG_ON(skb_shinfo(skb)->nr_frags)
```

[include/linux/skbuff.h]

We have already discussed that if the skbuff is fragmented in pages then the number of pages used is kept in the skb_shared_info structure, nr_frags variable.

6.3. SKB_FRAG_ASSERT

File : [include/linux/skbuff.h]

This macro will raise a bug if the skbuff is fragmented in a list of skbuffs. We have already discussed that if the skbuff is fragmented in a list of skbuffs then the the pointer to the next skbuff is kept in the skb_shared_info structure, frag_list variable.

[include/linux/skbuff.h]

```
#define SKB_FRAG_ASSERT(skb)   BUG_ON(skb_shinfo(skb)->frag_list)
```

[include/linux/skbuff.h]

6.4. skb_headlen and skb_pagelen

File : [include/linux/skbuff.h]

The skb_headlen function returns the size of the data occupied in the linear part of the skbuff. This is the total data stored in the skbuff len, minus the data stored in the nonlinear part of the skbuff : data_len.

[include/linux/skbuff.h]

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

[include/linux/skbuff.h]

The skb_pagelen function returns the size of the data stored in the array of pages in which the skbuff is fragmented.

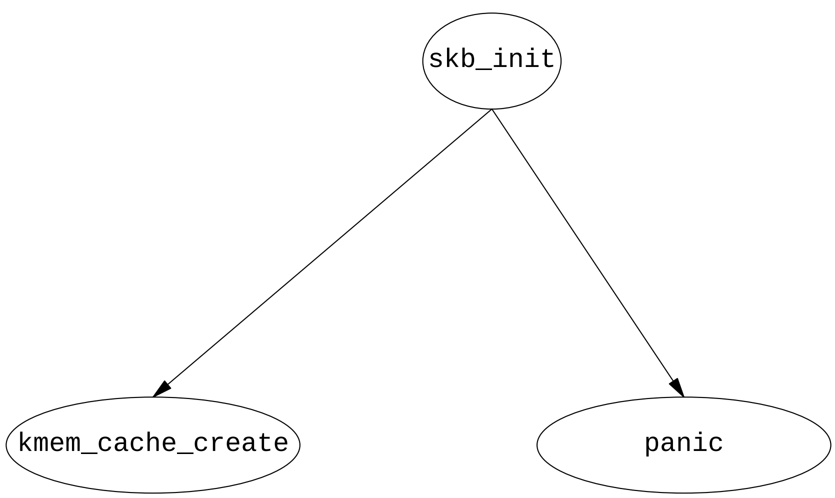
[include/linux/skbuff.h]

```
static inline int skb_pagelen(const struct sk_buff *skb)
{
     int i, len = 0;

     for (i = (int)skb_shinfo(skb)->nr_frags - 1; i >= 0; i--)
          len += skb_shinfo(skb)->frags[i].size;
     return len + skb_headlen(skb);
}
```

[include/linux/skbuff.h]

6.5. skb_init

File : `[include/net/socket.h]`



This function initializes the skbuff system. It is called by the `sock_init()` function in the `[net/socket.c]` at socket initialization time. It creates a slab named `skbuff_head_cache` for skbuff head objects. It doesnt specifiy any specific constructor or destructor for them.

[include/net/socket.h]

```
void __init skb_init(void)
{
     skbuff_head_cache = kmem_cache_create("skbuff_head_cache",
                                  sizeof(struct sk_buff),
                                  0,
                                  SLAB_HWCACHE_ALIGN,
                                  NULL, NULL);
     if (!skbuff_head_cache)
```

```
            panic("cannot create skbuff cache");
}
```

[include/net/socket.h]

6.6. alloc_skb

File : `[include/net/tcp.h]`

This function allocates a network buffer. It takes 2 arguments the size in bytes of the data area requested and the set of flags that tell the memory allocator how to behave. For the most part the network code calls the memory allocator with the `GFP_ATOMIC` set of flags: do not return without completing the task (for the moment this is equivalent to the `__GFP_HIGH` flag : can use emergency pools). The tcp code for instance uses its own skb allocator `tcp_alloc_skb` to request additional `MAX_TCP_HEADER` bytes to accomodate a headroom sufficient for the header (usually this is 128+32=160 bytes).

[include/net/tcp.h]

```
static inline struct sk_buff *tcp_alloc_pskb(struct sock *sk, int size, int mem, int gfp)
{
      struct sk_buff *skb = alloc_skb(size+MAX_TCP_HEADER, gfp);

      if (skb) {
            skb->truesize += mem;
            if (sk->sk_forward_alloc >= (int)skb->truesize ||
                tcp_mem_schedule(sk, skb->truesize, 0)) {
                  skb_reserve(skb, MAX_TCP_HEADER);
                  return skb;
            }
            __kfree_skb(skb);
      } else {
            tcp_enter_memory_pressure();
            tcp_moderate_sndbuf(sk);
      }
      return NULL;
}

static inline struct sk_buff *tcp_alloc_skb(struct sock *sk, int size, int gfp)
{
      return tcp_alloc_pskb(sk, size, 0, gfp);
}

```

[include/net/tcp.h]

In the ip fragmentation case for instance, additional bytes for the ip header and the link layer header properly aligned are requested :

[net/ipv4/ip_output.c]

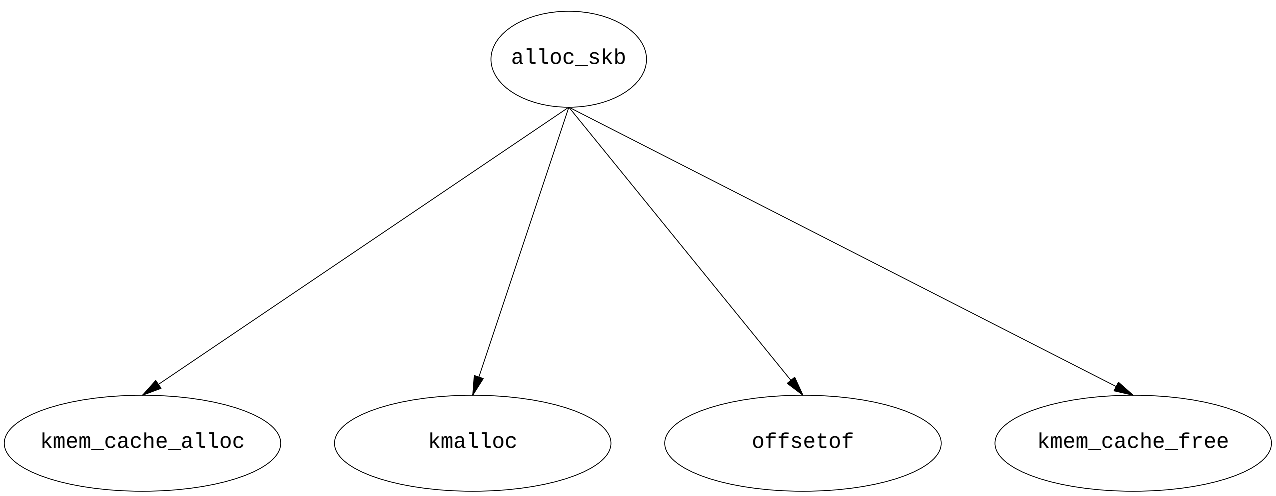
```
            /*
             *   Allocate buffer.
             */

```

```
                if ((skb2 = alloc_skb(len+hlen+LL_RESERVED_SPACE(rt->u.dst.dev), GFP_ATOMIC)) == NULL) {
                        NETDEBUG(printk(KERN_INFO "IP: frag: no memory for new fragment!0));
                        err = -ENOMEM;
                        goto fail;
                }
```

[net/ipv4/ip_output.c]

(`LL_RESERVED_SPACE` is the link layer header size + space to properly align it to mod `HH_DATA_MOD` = 16 this).

The memory allocated for the data area, of course, is contiguous physical memory. The current slab allocator provides size-N caches in power of 2 sizes from 32 bytes to 128 KB.



[net/core/skbuff.c]

```
/**
 *   alloc_skb -    allocate a network buffer
 *   @size: size to allocate
 *   @gfp_mask: allocation mask
 *
 *   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 using a @gfp_mask of
 *   %GFP_ATOMIC.
 */
struct sk_buff *alloc_skb(unsigned int size, int gfp_mask)
{
        struct sk_buff *skb;
        u8 *data;

```

[net/core/skbuff.c]

An skbuff head is allocated from the `skbuff_head_cache` slab. DMA suitable memory is not needed for the skbuff header, because you dont perform i/o over its data, so we reset that flag in the call for the allocation in the case the alloc_skb function was called with the flag set. If the allocation fails the function returns `NULL.`

[net/core/skbuff.c]

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

```

[net/core/skbuff.c]

If it succeeds then it allocates the skbuff data area from one of the size-N slabs using the `kmalloc()` function. The size of the data area requested through `kmalloc()` is augmented with the size of the `skb_shared_info` that can store the information on the `frag_list` or `frags[]` array of pages used by fragmented skbuffs. Th SKB_DATA_ALIGN macro adds enough bytes to the requested size so that the skb data area can be aligned with a level 1 cache line (on P4 for example the X86_L1_CACHE_SHIFT is 2ˆ7=128 bytes and so 127 is added)

[net/core/skbuff.c]

```
        /* 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;

```

[net/core/skbuff.c]

If it fails in allocating the data area it gives back the area for the skbuff head and returns `NULL.`

[net/core/skbuff.c]

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

[net/core/skbuff.c]

Then it initializes to 0 all bytes of the skbuff head up to the truesize field. The remaining bytes are not zeroed because they will be immediately initialized with the appropriate values (pointers to the data area of the skbuff and size). The skb truesize is initialized to the total allocated size : the requested data size plus the size of the skbuff header.

[net/core/skbuff.c]

```
        memset(skb, 0, offsetof(struct sk_buff, truesize));
        skb->truesize = size + sizeof(struct sk_buff);
```

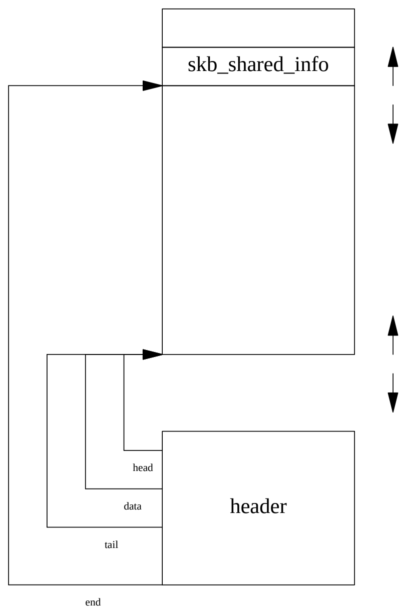
[net/core/skbuff.c]

Then the skbuff pointers inside the data area are initialized to a 0 size area :

[net/core/skbuff.c]

```
        skb->head = data;
        skb->data = data;
        skb->tail = data;
        skb->end  = data + size;

```

[net/core/skbuff.c]



And the shared_info are then initialized to an unfragmented skbuff :

[net/core/skbuff.c]

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

[net/core/skbuff.c]

only 1 reference to this skb, itself.

[net/core/skbuff.c]

```
        skb_shinfo(skb)->nr_frags  = 0;
```

[net/core/skbuff.c]

no pages in the frags array. The tso_ fields refer to the Tcp Segmentation Offloading experimental kernel feature to support some intellignet network cards that can perform the tcp segmentation (Myricom Gigabit Ethernet, National DP83820,..). This feature is described by the NETIF_F_TSO flag in the netdev structure. The tso_size is set to the mtu - hlen and tso_segs is the number of segments required to transmit this skbuff.

[net/core/skbuff.c]

```
        skb_shinfo(skb)->tso_size = 0;
        skb_shinfo(skb)->tso_segs = 0;
        skb_shinfo(skb)->frag_list = NULL;
```

[net/core/skbuff.c]

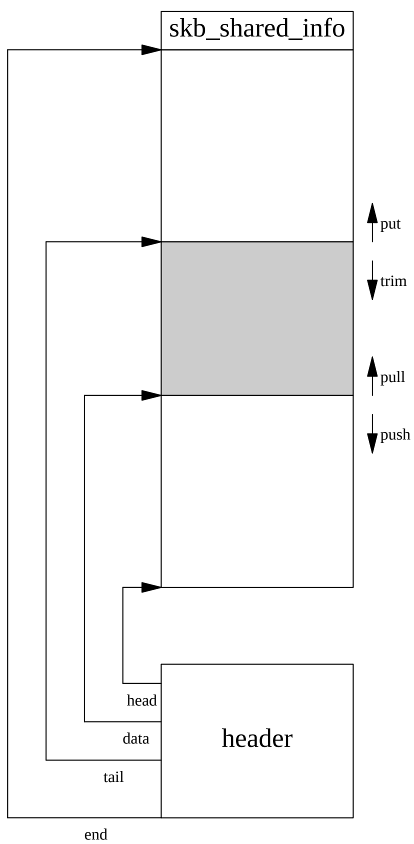
and an empty fragment list.

Fig. - After alloc_sk

---------------------------------- The `skb->data_len` field .. it is different from zero only on nonlinear skbuffs. In fact the function `skb_is_nonlinear()` returns that field. It seems to represent the number of bytes in the remaining skbuff (after the 1st). The function `skb_headlen()` returns the bytes in the linear part of the first skbuff : skb->data_len.`skb->len`-

------------------- The pskb_.. and __pskb_.. functions refer to the fragmented skbuffs. As usual the __ functions perform less or no check at all. -------------------

So the head and end pointers are fixed for an skbuff. The head points to the very beginning of the data area as obtained from kmalloc while the end points to the last useable area by the data. After it the skb_shared_info structure is kept.



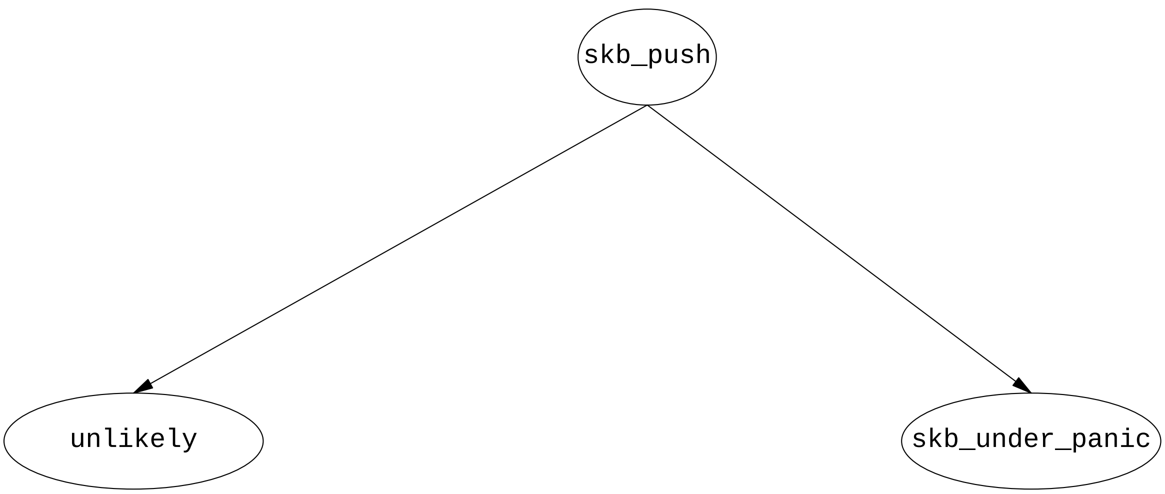
The data and tail instead can be moved with the following operations :

```
skb->data : push .. extends the used data area towards the beginning of
                    the buffer skb->head
            pull .. shrinks the beginning of the used data area
skb->tail : trim .. shrinks the end of the used data area
            put .. extends the end of the used data area towards skb->end
```

There are 2 implementations of each of these operations on skbuffs. One with consistency checks named ude/linux/skbiff.h] skb_put() kb_push() .. and so on. And one named with a prepended double underscore (`__skb_push()`,...) that doesnt apply any consistency check, this is used for eficiency reasons when it is clear that the checks are not needed.

6.7. skb_push

File : `[include/linux/skbuff.h]`



This function is usually called to prepare the space where to prepend protocol headers. For example in the net/ipv4/tcp_output.c file the skbuff is adjusted for the tcp header space with

[include/linux/skbuff.h]

```
                th = (struct tcphdr *) skb_push(skb, tcp_header_size);
                skb->h.th = th;
```

[include/linux/skbuff.h]

after calling this function the result is the new `skb->data` pointer (the new beginning of the data area) and the header is then copied from there on.

[include/linux/skbuff.h]

```

/**
 *   skb_push - add data to the start of a buffer
 *   @skb: buffer to use
 *   @len: amount of data to add
 *
 *   This function extends the used data area of the buffer at the buffer
 *   start. If this would exceed the total buffer headroom the kernel will
 *   panic. A pointer to the first byte of the extra data is returned.
```

```
 */
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;
}
```

[include/linux/skbuff.h]

The skb->data pointer pointing at the beginning of the used data area is shrunk of len bytes and the len of the data area used in the skbuff is increased of the same number. In the unlikely case in which the skb->data pointer with this operation goes outside the skbuff available data area the kernel panics with an "skput: under .. " message. At the end the function returns the updated skb->data pointer. This is the very easily understandable implementation without the check :

[include/linux/skbuff.h]

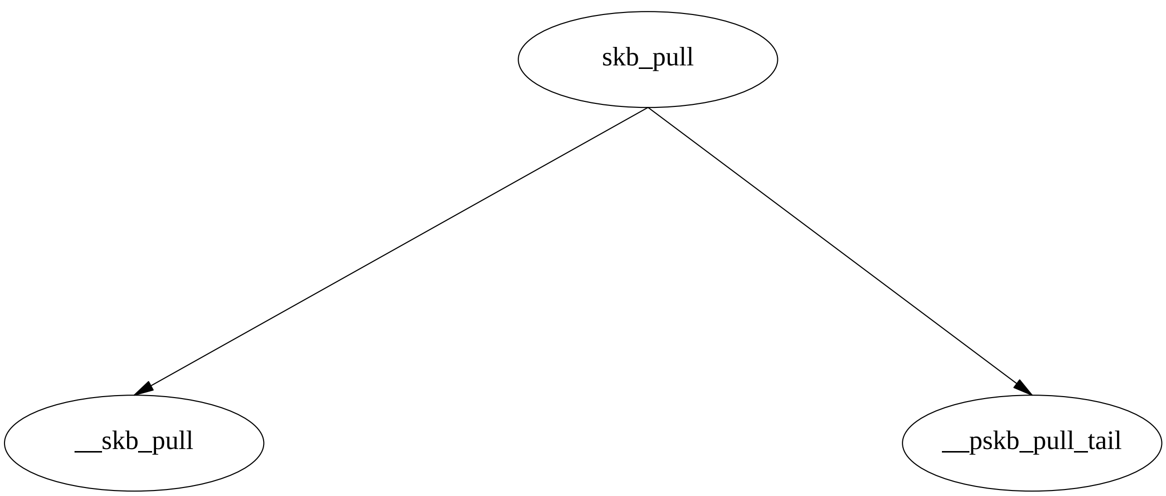
```

static inline unsigned char *__skb_push(struct sk_buff *skb, unsigned int len)
{
     skb->data -= len;
     skb->len  += len;
     return skb->data;
}

```

[include/linux/skbuff.h]

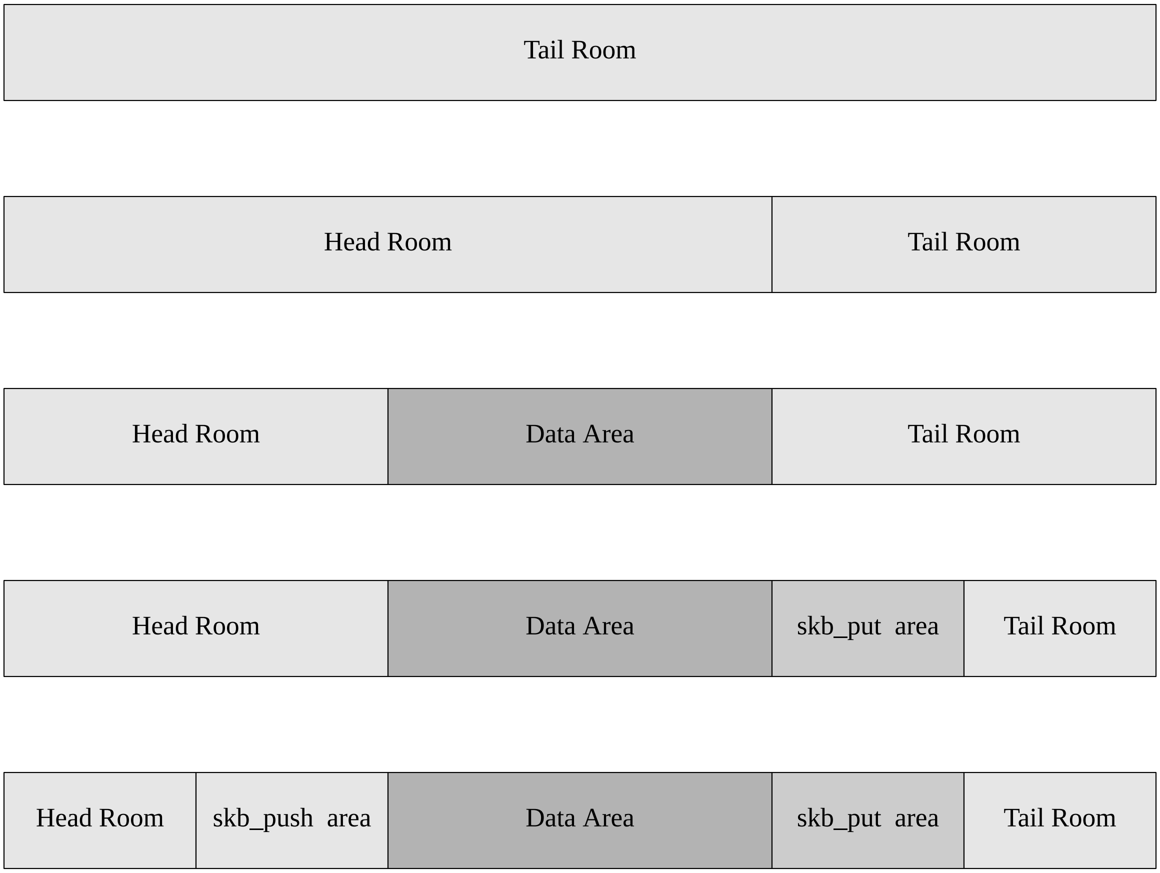
I dont know why this is not called inside skb_push.

6.8. skb_pull()

File : [include/linux/skbuff.h]



In the pull operation if the len by which you ask to decrease the used data area is larger then the actual skb->len then the function returns a NULL.



Otherwise the actual skb->len is decreased by the requested amount and the `skb->data`pointer is augmented by the same amount. A consistency check here is eventually performed to check that `skb->len` is less than `skb->data_len.`

[include/linux/skbuff.h]

```
/**
 *   skb_pull - remove data from the start of a buffer
 *   @skb: buffer to use
 *   @len: amount of data to remove
 *
 *   This function removes data from the start of a buffer, returning
 *   the memory to the headroom. A pointer to the next data in the buffer
 *   is returned. Once the data has been pulled future pushes will overwrite
 *   the old data.
 */
static inline unsigned char *skb_pull(struct sk_buff *skb, unsigned int len)
{
     return (len > skb->len) ? NULL : __skb_pull(skb, len);
}

```

[include/linux/skbuff.h]

[include/linux/skbuff.h]

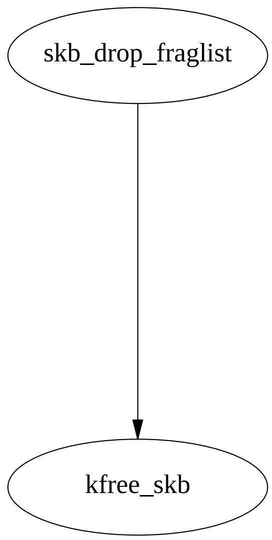
```

static inline char *__skb_pull(struct sk_buff *skb, unsigned int len)
```

```
{
        skb->len -= len;
        BUG_ON(skb->len < skb->data_len);
        return skb->data += len;
}

```

[include/linux/skbuff.h]

6.9. skb_drop_fraglist

File : `[net/core/skbuff.c]`



this function drops (`kfree_skb(skb)`) all the fragments of this skbuff and resets to null the skb->frag_list pointer. This function is called when all the data can be discarded or all the data in the skbuff sbk->len is in this skbuff and so the skbuff is reset to a linear one.

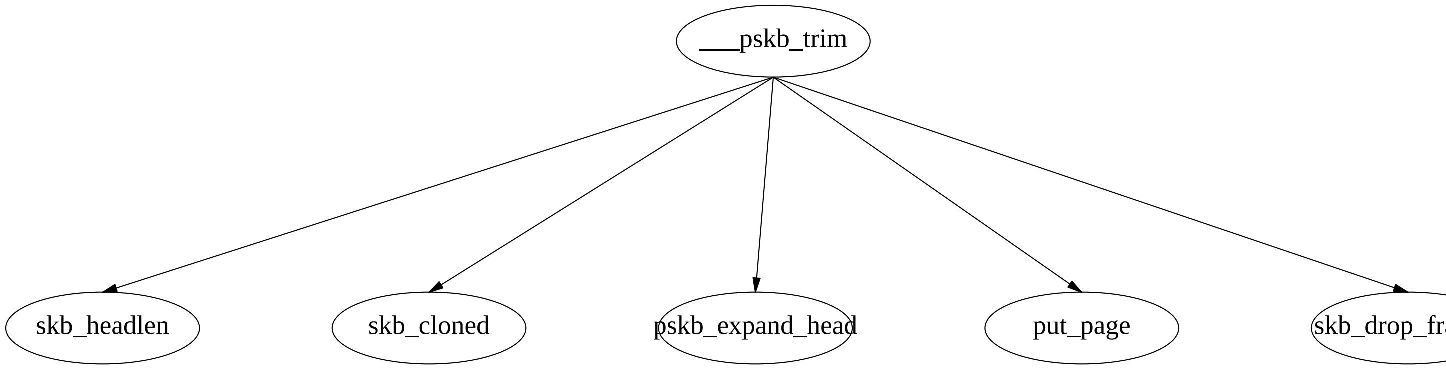
[net/core/skbuff.c]

```
static void skb_drop_fraglist(struct sk_buff *skb)
{
        struct sk_buff *list = skb_shinfo(skb)->frag_list;

        skb_shinfo(skb)->frag_list = NULL;

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

[net/core/skbuff.c]

6.10. ___pskb_trim

File : `[net/core/skbuff.c]`



This function is called to trim a nonlinear skbuff. An skbuff can be constituted by an array of pages (`skb->frags[]`) and/or a list of skbuffs (`skb->fraglist`).

[net/core/skbuff.c]

```

/* 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)
{
     int offset = skb_headlen(skb);
     int nfrags = skb_shinfo(skb)->nr_frags;
     int i;

     for (i = 0; i < nfrags; i++) {
          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 (len <= offset) {
                    put_page(skb_shinfo(skb)->frags[i].page);
                    skb_shinfo(skb)->nr_frags--;
               } else {
                    skb_shinfo(skb)->frags[i].size = len - offset;
               }
          }
          offset = end;
     }
```

[net/core/skbuff.c]

We know that an skbuff has multiple data pages associated with it if the number in the `skb_shared_info` structure `skb_shinfo(skb)->nr_frags` is different from zero. In this case we run through the pages until eventually their total size reaches the requested len. If this happen before the end of the array the page is relinquished and the number of fragments is decresed by 1.

[net/core/skbuff.c]

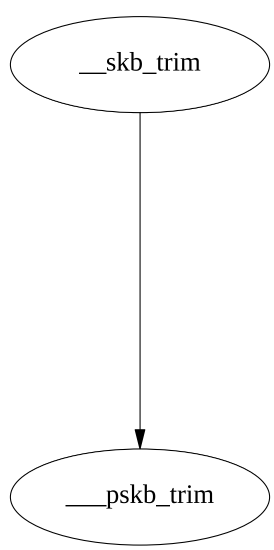
```

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

        return 0;
   }
```

[net/core/skbuff.c]

if the data in the pages associated with this skbuff is not enough to satisfy the request then we simply reset the total data len of the skbuff to len and we decrease the length of data in the remaining skbuffs (skb->data_len) by the proper amount. offset is here the total data in all the array of pages associated with the 1st skbuff (while headlen is the data in this skbuff) Otherwise there are 2 possibilities : - if the data in the skbuff is enough we reset the skbuff to
a linear one, we set the len to the requested one, we just
trime the tail of this skbuff, and eventually we drop all the
other fragments in the fraglist - we still need some fragments .. in this case we reset the length
to len and we decrease the skb->data_len of the proper amount

6.11. __skb_trim

File : `[include/linux/skbuff.h]`



If the skbuff is linear (skb->data_len == 0) simply sets the total data length to len and trims the skb->tail pointer. Otherwise if the skbuff is nonlinear it calls the ___pskb_trim() function.

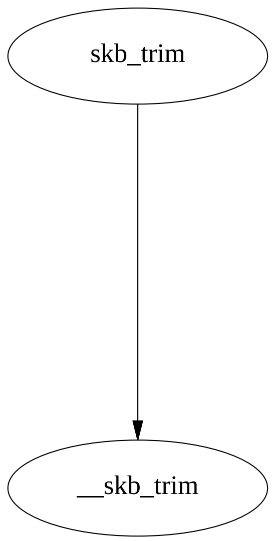
[include/linux/skbuff.h]

```
static inline void __skb_trim(struct sk_buff *skb, unsigned int len)
{
        if (!skb->data_len) {
                skb->len  = len;
                skb->tail = skb->data + len;
        } else
                ___pskb_trim(skb, len, 0);
}
```

[include/linux/skbuff.h]

6.12. skb_trim

File : `[include/linux/skbuff.h]`



This wrapper function just makes a consistency check to see if the total data in the skbuff is sufficient to satisfy the request and then calls __skb_trim()

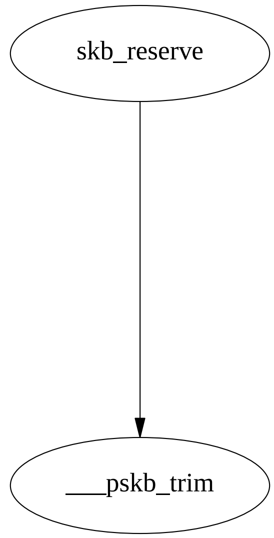
[include/linux/skbuff.h]

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

[include/linux/skbuff.h]

6.13. skb_reserve

File : `[include/linux/skbuff.h]`



This function adjusts the skbuff headroom (at the beginning there is no headroom and all tailroom) just after the creation (the skbuff should be empty). This is done moving the data and tail pointers, that just after creation point to the `skb->head`, by `len` bytes.

[include/linux/skbuff.h]

```
/**
 *   skb_reserve - adjust headroom
 *   @skb: buffer to alter
 *   @len: bytes to move
 *
 *   Increase the headroom of an empty &sk_buff by reducing the tail
 *   room. This is only allowed for an empty buffer.
 */
static inline void skb_reserve(struct sk_buff *skb, unsigned int len)
{
     skb->data += len;
     skb->tail += len;
}

```

[include/linux/skbuff.h]

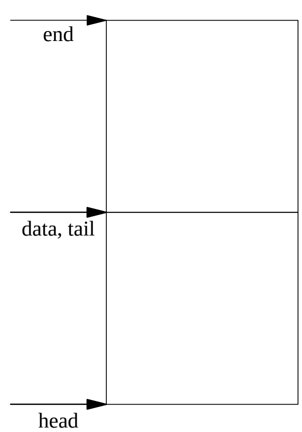
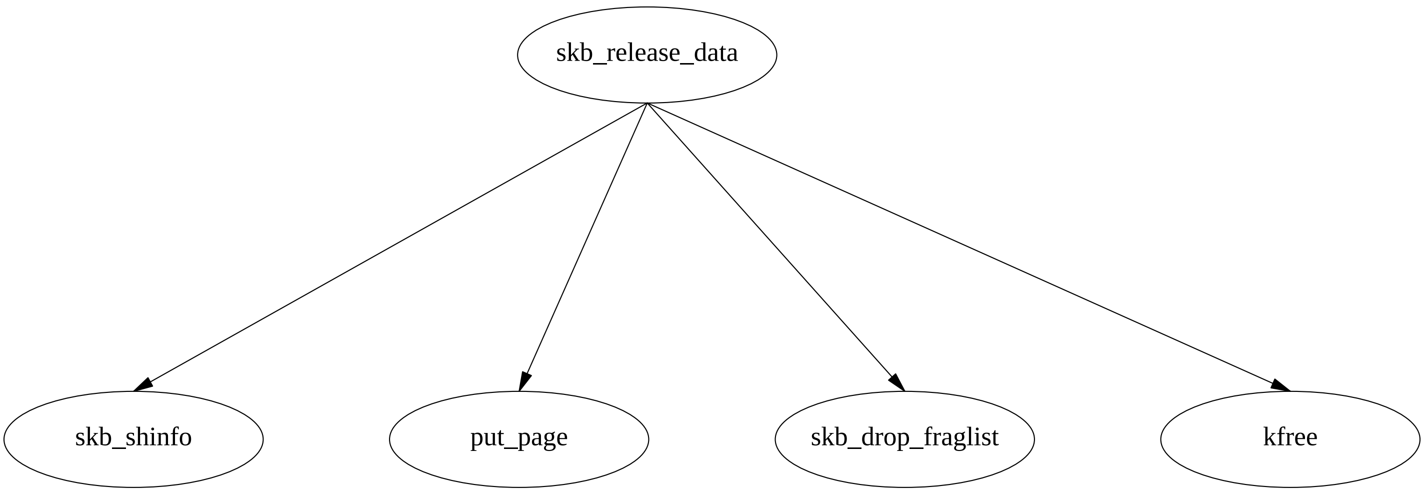


Fig. - After skb_reserve

6.14. skb_release_data

File : `[net/core/skbuff.c]`



This function releases all the data areas associated with an skbuff if this skbuff was not cloned or the number of users is 0. In that case the function goes through the array of fragments and puts back to the page allocator the pages associated. Then if there is a frag_list it drops all the fragments (skb_drop_fraglist). and finally it frees the data area associated with the skbuff and consequently the skb_shared_info area.

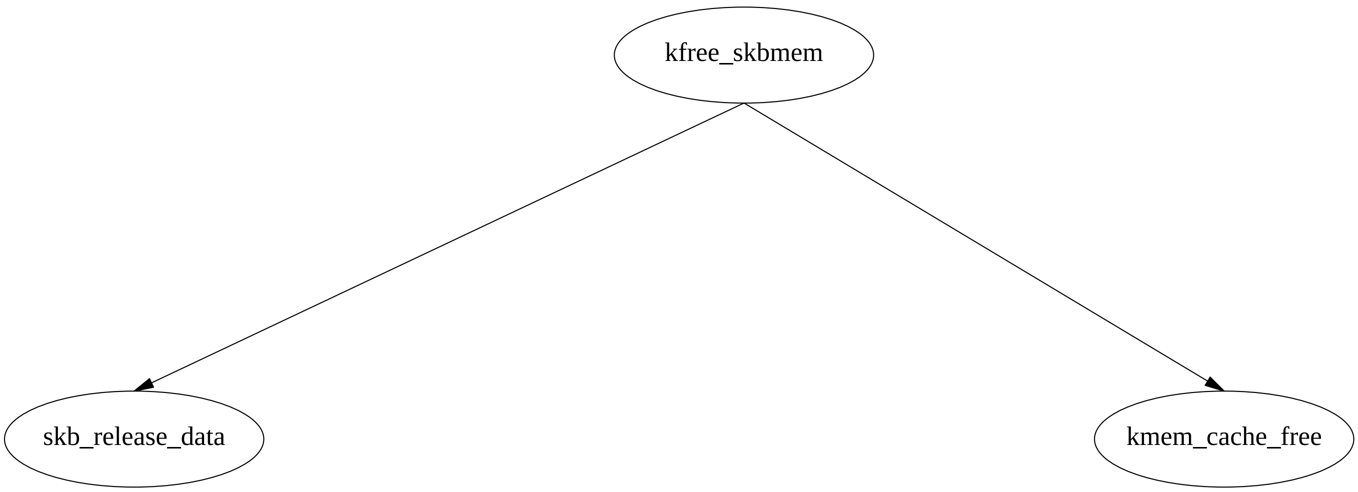
[net/core/skbuff.c]

```
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);
     }
}
```

[net/core/skbuff.c]

6.15. kfree_skbmem

File : `[net/core/skbuff.c]`



This function releases all the memory associated with an skbuff. It doesnt clean its state. First it tries to release all data areas (this is not done if the data areas are in use). Then it frees the skbuff header from the appropriate slab.

[net/core/skbuff.c]

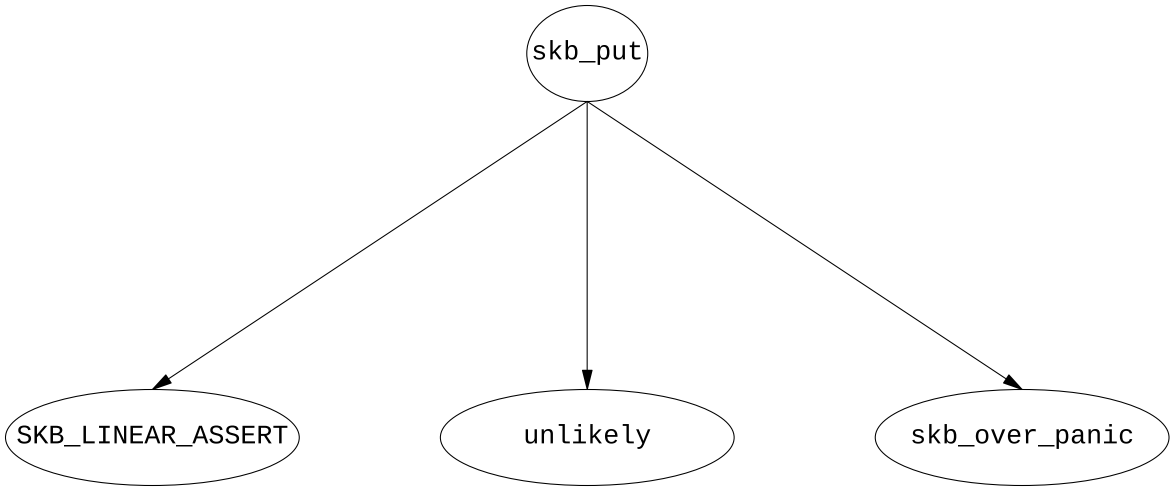
```
/*
 *   Free an skbuff by memory without cleaning the state.
 */
void kfree_skbmem(struct sk_buff *skb)
{
     skb_release_data(skb);
     kmem_cache_free(skbuff_head_cache, skb);
}

```

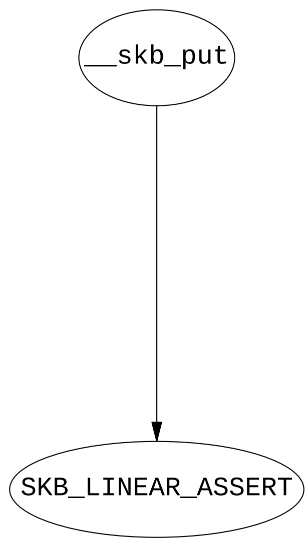
[net/core/skbuff.c]

6.16. [include/linux/skbuff.h]

File : `[include/linux/skbuff.h]`

As for other functions there are 2 versions of the skb_put function. The __skb_put() function saves just a consistency check on the sufficency of data space (tail > end). This function is usually called after the skb_reserve() function has been called on a newly allocated skbuff moving the data pointer, to move the tail pointer and prepare the space to copy over the data. It updates the len field of the skbuff header. This operation can be applied only on linear skbuffs.





[include/linux/skbuff.h]

```
#define SKB_LINEAR_ASSERT(skb)  BUG_ON(skb_is_nonlinear(skb))

/*
 *   Add data to an sk_buff
 */
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;
     return tmp;
}

/**
```

```
 *   skb_put - add data to a buffer
 *   @skb: buffer to use
 *   @len: amount of data to add
 *
 *   This function extends the used data area of the buffer. If this would
 *   exceed the total buffer size the kernel will panic. A pointer to the
 *   first byte of the extra data is returned.
 */
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;
}
```

[include/linux/skbuff.h]

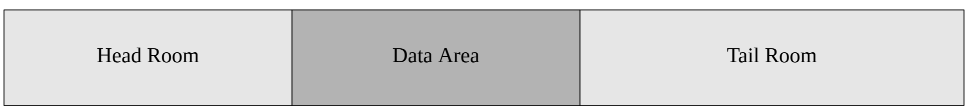
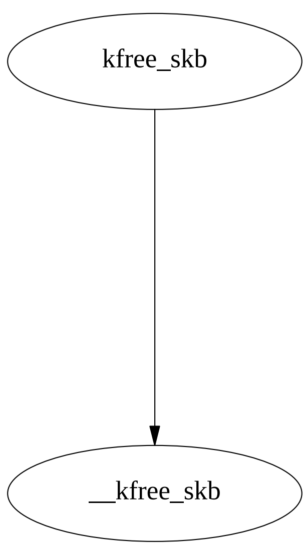


Fig. - An sk_buff containing data

6.17. [include/linux/skbuff.h] kfree_skb()

File : `[include/linux/skbuff.h]`



Look at this code !!!!!!! It means :

if there is only 1 user (then this user of the skbuff is freeing it)
of the skbuff call __kfree_skb(skb) and return.
otherwise just decrement the number of users and return.

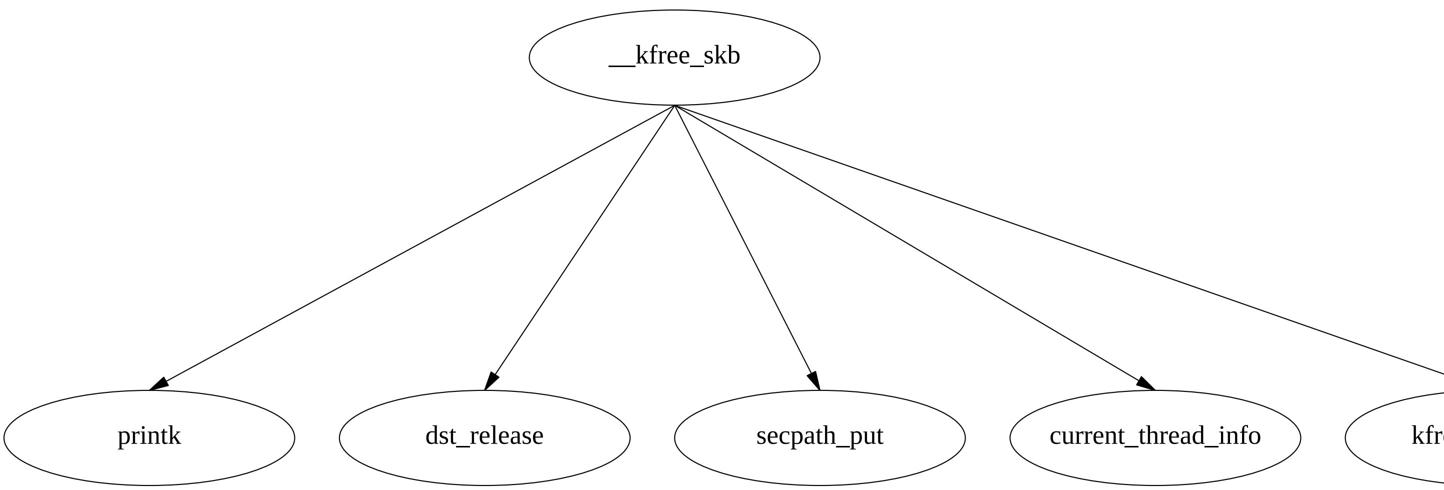
[include/linux/skbuff.h]

```
/**
 *   kfree_skb - free an sk_buff
 *   @skb: buffer to free
 *
 *   Drop a reference to the buffer and free it if the usage count has
 *   hit zero.
 */
static inline void kfree_skb(struct sk_buff *skb)
{
     if (atomic_read(&skb->users) == 1 || atomic_dec_and_test(&skb->users))
          __kfree_skb(skb);
}

```

[include/linux/skbuff.h]

6.18. __kfree_skb

File : `[net/core/skbuff.c]`



This function cleans the state of the skbuff and releases any data area associated with it. Something went wrong if we came here and the skbuff is still on a list (a socket or device list), print a kernel warning msg. Release the dst entry in the dst cache. If there is a destructor defined for the skb call it and eventually print a warning if we are executing out of an IRQ. Release all the memory associated with the skb calling kfree_skbmem.

[net/core/skbuff.c]

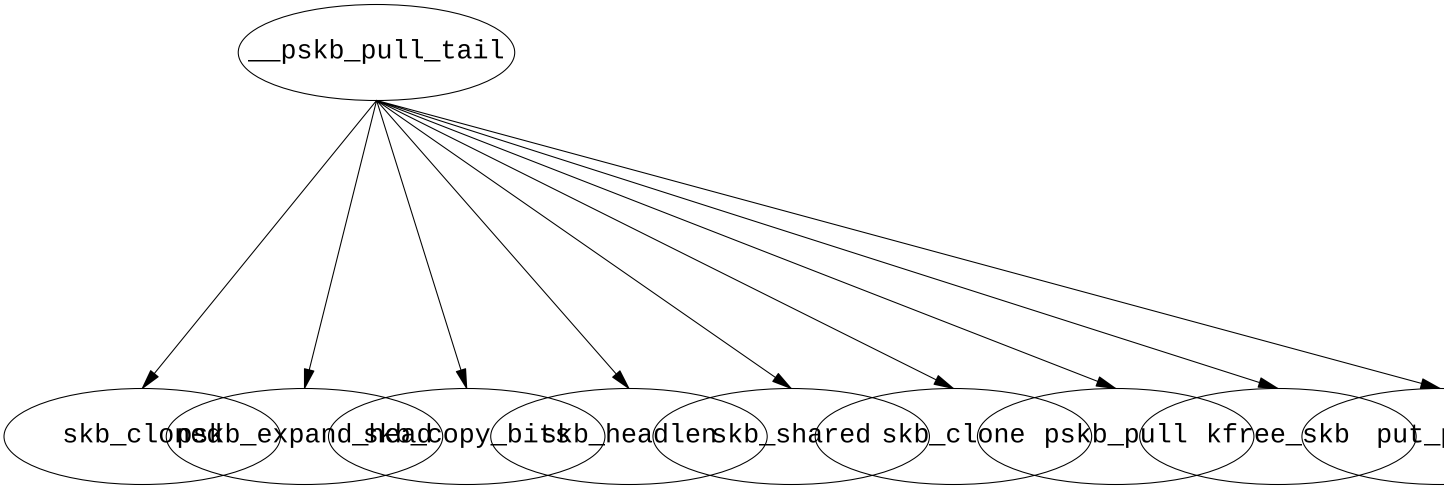
```
/*
 *   Free an skbuff by memory without cleaning the state.
 */
void kfree_skbmem(struct sk_buff *skb)
{
     skb_release_data(skb);
```

```
        kmem_cache_free(skbuff_head_cache, skb);
}

/**
 *      __kfree_skb - private function
 *      @skb: buffer
 *
 *      Free an sk_buff. Release anything attached to the buffer.
 *      Clean the state. This is an internal helper function. Users should
 *      always call kfree_skb
 */

void __kfree_skb(struct sk_buff *skb)
{
        if (skb->list) {
                printk(KERN_WARNING "Warning: kfree_skb passed an skb still "
                       "on a list (from %p).0, NET_CALLER(skb));
                BUG();
        }

        dst_release(skb->dst);
#ifdef CONFIG_XFRM
        secpath_put(skb->sp);
#endif
        if(skb->destructor) {
                if (in_irq())
                        printk(KERN_WARNING "Warning: kfree_skb on "
                                            "hard IRQ %p0, NET_CALLER(skb));
                skb->destructor(skb);
        }
#ifdef CONFIG_NETFILTER
        nf_conntrack_put(skb->nfct);
#if defined(CONFIG_BRIDGE) || defined(CONFIG_BRIDGE_MODULE)
        nf_bridge_put(skb->nf_bridge);
#endif
#endif
        kfree_skbmem(skb);
}
```

[net/core/skbuff.c]

6.19. __pskb_pull_tail

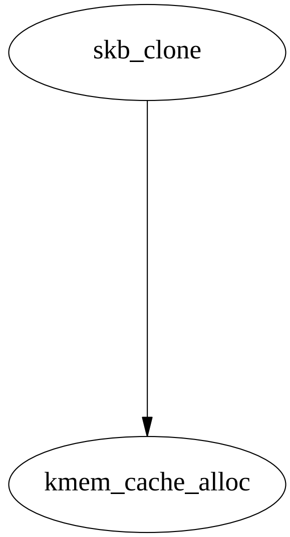
File : `[net/core/skbuff.c]`



This function expands the data area in the linear skbuff part of a fragmented skbuff copying the data from the remaining fragments. If that space is not sufficient, then it allocates a new data area and copies the data from the old to the new one and updates pointers in the descriptor. delta are the more bytes requested in the linear skbuff part. eat is the part of them that is not possible to allocate in the current linear part of the skbuff. If eat <=0 then we can keep the current skbuff data area and just update the pointers. If eat > 0 then we have to allocate a new linear part and in this case we will request 128 additional bytes to accomodate eventual future requests. We allocate a new linear part also in the case the skbuff has been cloned since we want to change that data area. Then we copy delta bytes from the fragmented tail of the old skbuff (those after headlen) to the the tail. To update the skbuff now if there is no frag_list then we just have to pull the array of pages, otherwise we have to go through the frag_list. If the array of pages associated with this skbuff is large enough then again we just have to pull the array. Then you go through the frag_list and you eat (kfree_skb) all the complete skbuffs that you can. When you are here it means that you are on an skbuff that you cant eat completely . For this you go through the array of pages and you free all those that you can completely eat (put_page). For the last page you update the page_offset and size values in the frags[k] structure.

6.20. skb_clone

File : `[net/core/skbuff.c]`



You clone an skbuff allocating a new skbuff header from the slab and initilizing its fields from the values of the old one. The data area is not copied, it is shared, so you increment the counter skb_shared_info->data_ref in the shared info area. The number of users of the new header (n->users) is set to 1, and you put the cloned flag in the old and new header to 1. The pointers of the doubly linked list to which the skbuff can be linked are initialized to NULL in the new header. And also the destructor in the new header is initialized to NULL.

[net/core/skbuff.c]

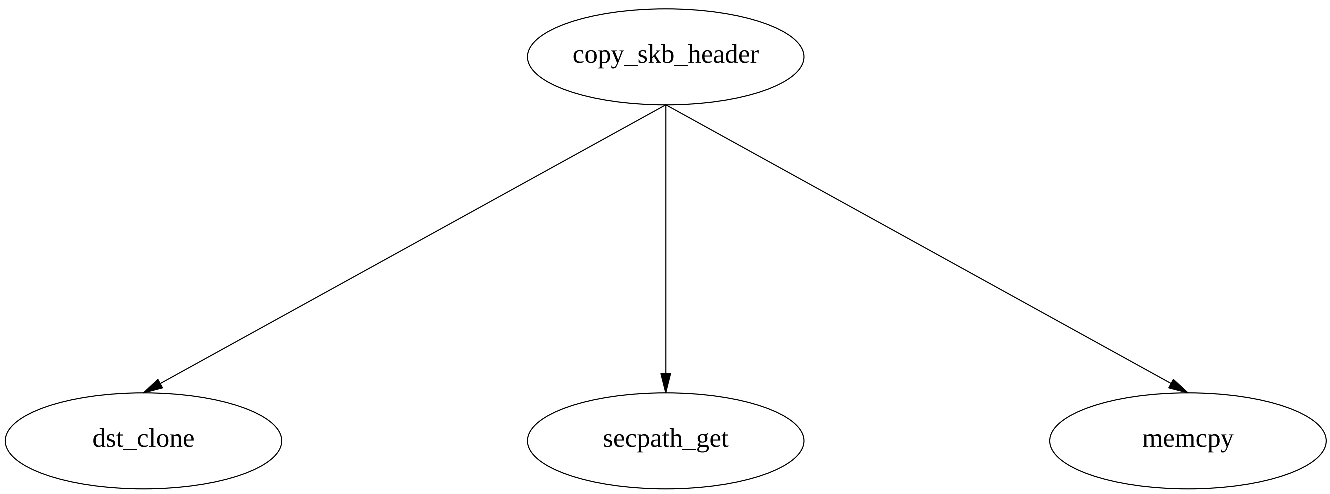
```
/**
 *   skb_clone -    duplicate an sk_buff
 *   @skb: buffer to clone
 *   @gfp_mask: allocation priority
 *
 *   Duplicate an &sk_buff. The new one is not owned by a socket. Both
 *   copies share the same packet data but not structure. The new
 *   buffer has a reference count of 1. If the allocation fails the
 *   function returns %NULL otherwise the new buffer is returned.
 *
 *   If this function is called from an interrupt gfp_mask() must be
 *   %GFP_ATOMIC.
 */

struct sk_buff *skb_clone(struct sk_buff *skb, int gfp_mask)
{
     struct sk_buff *n = kmem_cache_alloc(skbuff_head_cache, gfp_mask);

     if (!n)
          return NULL;

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

     n->next = n->prev = NULL;
     n->list = NULL;
     n->sk = NULL;
     C(stamp);
```

```
        C(dev);
        C(real_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;
        C(pkt_type);
        C(ip_summed);
        C(priority);
        C(protocol);
        C(security);
        n->destructor = NULL;
#ifdef CONFIG_NETFILTER
        C(nfmark);
        C(nfcache);
        C(nfct);
        nf_conntrack_get(skb->nfct);
#ifdef CONFIG_NETFILTER_DEBUG
        C(nf_debug);
#endif
#if defined(CONFIG_BRIDGE) || defined(CONFIG_BRIDGE_MODULE)
        C(nf_bridge);
        nf_bridge_get(skb->nf_bridge);
#endif
#endif /*CONFIG_NETFILTER*/
#if defined(CONFIG_HIPPI)
        C(private);
#endif
#ifdef CONFIG_NET_SCHED
        C(tc_index);
#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;
}
```

[net/core/skbuff.c]

6.21. copy_skb_header

File : `[net/core/skbuff.c]`



This function supposes that a copy of the data area of the old skb has already being done and initializes the pointers to the different layer headers (transport,network,mac ..) in the new skb to the same relative position as in the old skb. It initializes the number of users of the new skb to 1.

[net/core/skbuff.c]

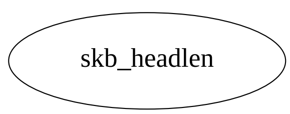
```
static void copy_skb_header(struct sk_buff *new, const struct sk_buff *old)
{
	/*
	 *	Shift between the two data areas in bytes
	 */
	unsigned long offset = new->data - old->data;

	new->list	= NULL;
	new->sk		= NULL;
	new->dev	= old->dev;
	new->real_dev	= old->real_dev;
	new->priority	= old->priority;
	new->protocol	= old->protocol;
	new->dst	= dst_clone(old->dst);
#ifdef CONFIG_INET
	new->sp		= secpath_get(old->sp);
#endif
	new->h.raw	= old->h.raw + offset;
	new->nh.raw	= old->nh.raw + offset;
	new->mac.raw	= old->mac.raw + offset;
	memcpy(new->cb, old->cb, sizeof(old->cb));
```

```
        new->local_df   = old->local_df;
        new->pkt_type   = old->pkt_type;
        new->stamp = old->stamp;
        new->destructor = NULL;
        new->security   = old->security;
#ifdef CONFIG_NETFILTER
        new->nfmark     = old->nfmark;
        new->nfcache    = old->nfcache;
        new->nfct  = old->nfct;
        nf_conntrack_get(old->nfct);
#ifdef CONFIG_NETFILTER_DEBUG
        new->nf_debug   = old->nf_debug;
#endif
#if defined(CONFIG_BRIDGE) || defined(CONFIG_BRIDGE_MODULE)
        new->nf_bridge  = old->nf_bridge;
        nf_bridge_get(old->nf_bridge);
#endif
#endif
#ifdef CONFIG_NET_SCHED
        new->tc_index   = old->tc_index;
#endif
        atomic_set(&new->users, 1);
}
```

[net/core/skbuff.c]

6.22. skb_headlen

File : `[include/linux/skbuff.h]`

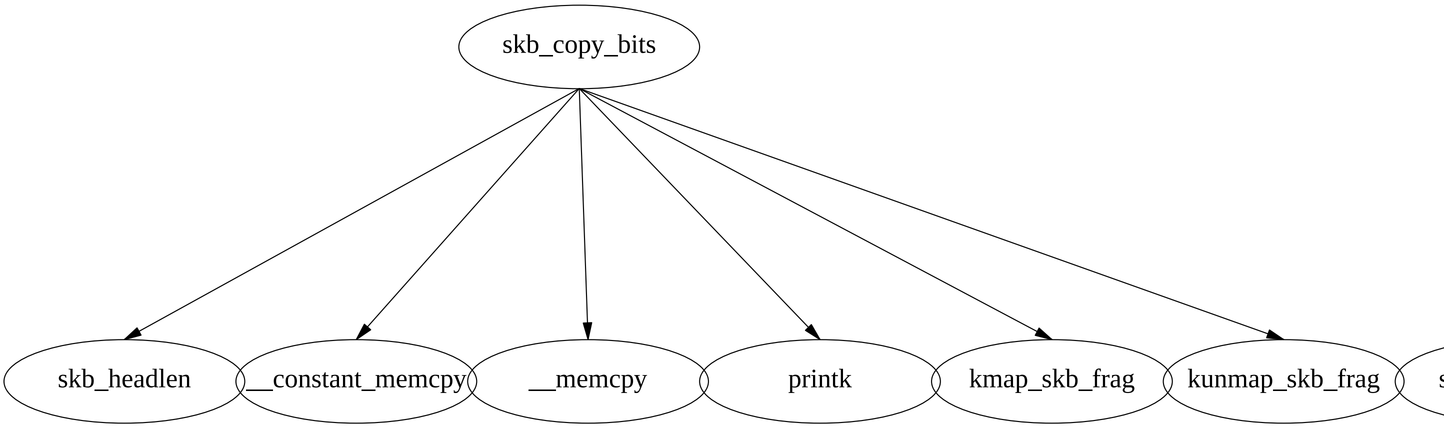


This function returns the number of data bytes in the first skbuff data area. It should be equal to (skb->tail - skb->data) i think.

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

6.23. skb_copy_bits

File : `[net/core/skbuff.c]`



This function copies bytes of data from an skb to a another area of memory. The first argument is a pointer to an skb header from which data area the data should be copied, the 3d arg is a pointer to an area of memory where the data should be put. The offset argument is the quantity that is added to the skb->data pointer to obtain the address from which the copy will start:

skb->data + offset The len argument is the number of bytes that should be copied. This function is able to treat fragmented skbuff and has code to copy all the fragments in the array of pages and all the eventual skbuffs linked together calling iteratively skb_copy_bits for each skbuff in the frag_list.

[net/core/skbuff.c]

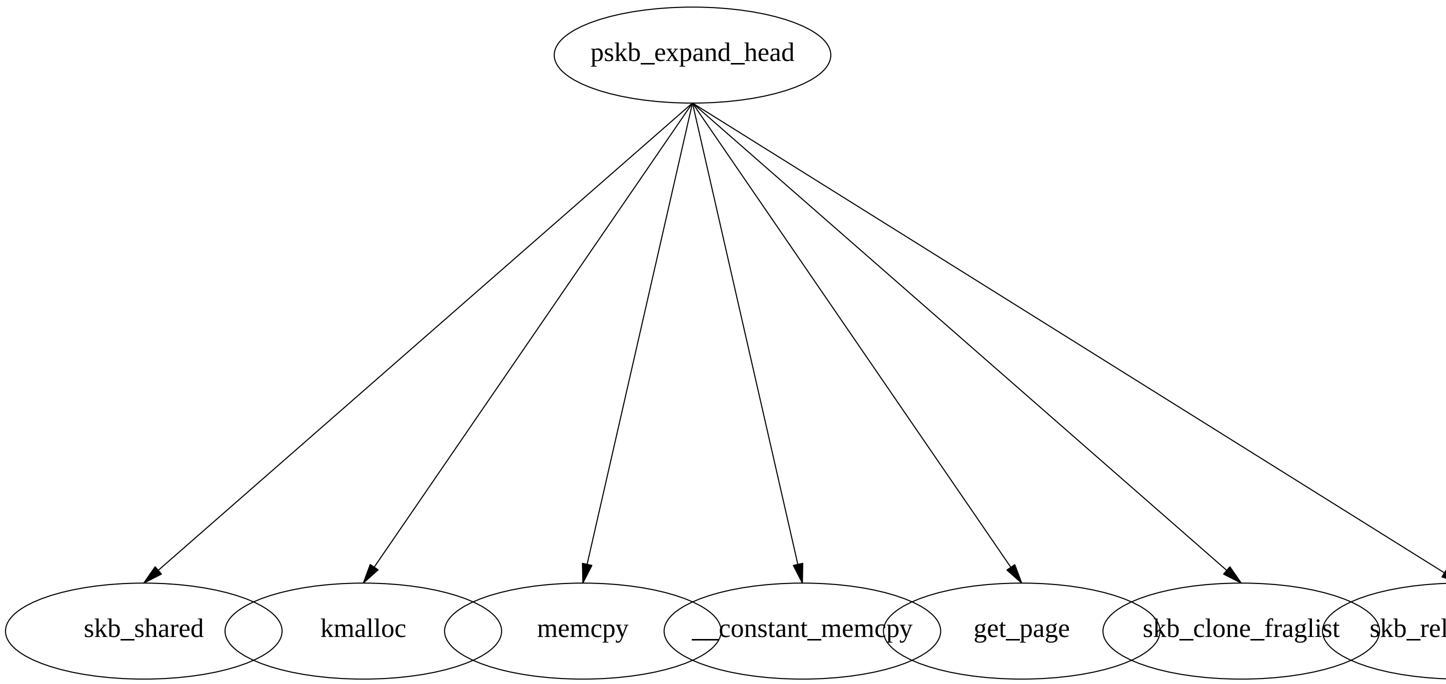
```

/* Copy some data bits from skb to kernel buffer. */

int skb_copy_bits(const struct sk_buff *skb, int offset, void *to, int len)
{
      int i, copy;
      int start = skb_headlen(skb);

      if (offset > (int)skb->len - len)
            goto fault;

      /* Copy header. */
      if ((copy = start - offset) > 0) {
            if (copy > len)
                  copy = len;
            memcpy(to, skb->data + offset, copy);
            if ((len -= copy) == 0)
                  return 0;
            offset += copy;
            to     += copy;
      }

      for (i = 0; i < skb_shinfo(skb)->nr_frags; i++) {
            int end;

            BUG_TRAP(start <= offset + len);

```

```
			end = start + skb_shinfo(skb)->frags[i].size;
			if ((copy = end - offset) > 0) {
				u8 *vaddr;

				if (copy > len)
					copy = len;

				vaddr = kmap_skb_frag(&skb_shinfo(skb)->frags[i]);
				memcpy(to,
				       vaddr + skb_shinfo(skb)->frags[i].page_offset+
				       offset - start, copy);
				kunmap_skb_frag(vaddr);

				if ((len -= copy) == 0)
					return 0;
				offset += copy;
				to     += copy;
			}
			start = end;
		}

		if (skb_shinfo(skb)->frag_list) {
			struct sk_buff *list = skb_shinfo(skb)->frag_list;

			for (; list; list = list->next) {
				int end;

				BUG_TRAP(start <= offset + len);

				end = start + list->len;
				if ((copy = end - offset) > 0) {
					if (copy > len)
						copy = len;
					if (skb_copy_bits(list, offset - start,
							    to, copy))
						goto fault;
					if ((len -= copy) == 0)
						return 0;
					offset += copy;
					to     += copy;
				}
				start = end;
			}
		}
		if (!len)
			return 0;

	fault:
		return -EFAULT;
	}
```

[net/core/skbuff.c]

6.24. pskb_expand_head

File : `[net/core/skbuff.c]`



This function allocates a new linear skbuff data area with enough space to provide the specified headroom and tailroom. Then it copies all the data from the old skbuff to this one, eventually reducing a fragmented skbuff to a linear one. The skbuff header remains the same, just the pointers to the data area are changed.

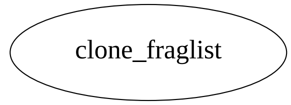
[net/core/skbuff.c]

```
/**
 *   pskb_expand_head - reallocate header of &sk_buff
 *   @skb: buffer to reallocate
 *   @nhead: room to add at head
 *   @ntail: room to add at tail
 *   @gfp_mask: allocation priority
 *
 *   Expands (or creates identical copy, if &nhead and &ntail are zero)
 *   header of skb. &sk_buff itself is not changed. &sk_buff MUST have
 *   reference count of 1. Returns zero in the case of success or error,
 *   if expansion failed. In the last case, &sk_buff is not changed.
 *
 *   All the pointers pointing into skb header may change and must be
 *   reloaded after call to this function.
 */

int pskb_expand_head(struct sk_buff *skb, int nhead, int ntail, int gfp_mask)
{
     int i;
     u8 *data;
     int size = nhead + (skb->end - skb->head) + ntail;
     long off;

     if (skb_shared(skb))
```

```
                BUG();

        size = SKB_DATA_ALIGN(size);

        data = kmalloc(size + sizeof(struct skb_shared_info), gfp_mask);
        if (!data)
                goto nodata;

        /* Copy only real data... and, alas, header. This should be
         * optimized for the cases when header is void. */
        memcpy(data + nhead, skb->head, skb->tail - skb->head);
        memcpy(data + size, skb->end, sizeof(struct skb_shared_info));

        for (i = 0; i < skb_shinfo(skb)->nr_frags; i++)
                get_page(skb_shinfo(skb)->frags[i].page);

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

        skb_release_data(skb);

        off = (data + nhead) - skb->head;

        skb->head     = data;
        skb->end      = data + size;
        skb->data    += off;
        skb->tail    += off;
        skb->mac.raw += off;
        skb->h.raw   += off;
        skb->nh.raw  += off;
        skb->cloned   = 0;
        atomic_set(&skb_shinfo(skb)->dataref, 1);
        return 0;

nodata:
        return -ENOMEM;
}
```

[net/core/skbuff.c]

6.25. [net/core/skbuff.c] clone_fraglist()



This function traverse the list of skbuffs and invokes skb_get for each of them. Skb_get simply increments the number of users of the skbuff. So this clone function works through a copy-on-write mechanism : nothing is really copied now, it is the responsability of those who wants to write on the skbuffs to copy them. This can save some not needed copies.

```
static void skb_clone_fraglist(struct sk_buff *skb)
{
     struct sk_buff *list;

     for (list = skb_shinfo(skb)->frag_list; list; list = list->next)
          skb_get(list);
}

```

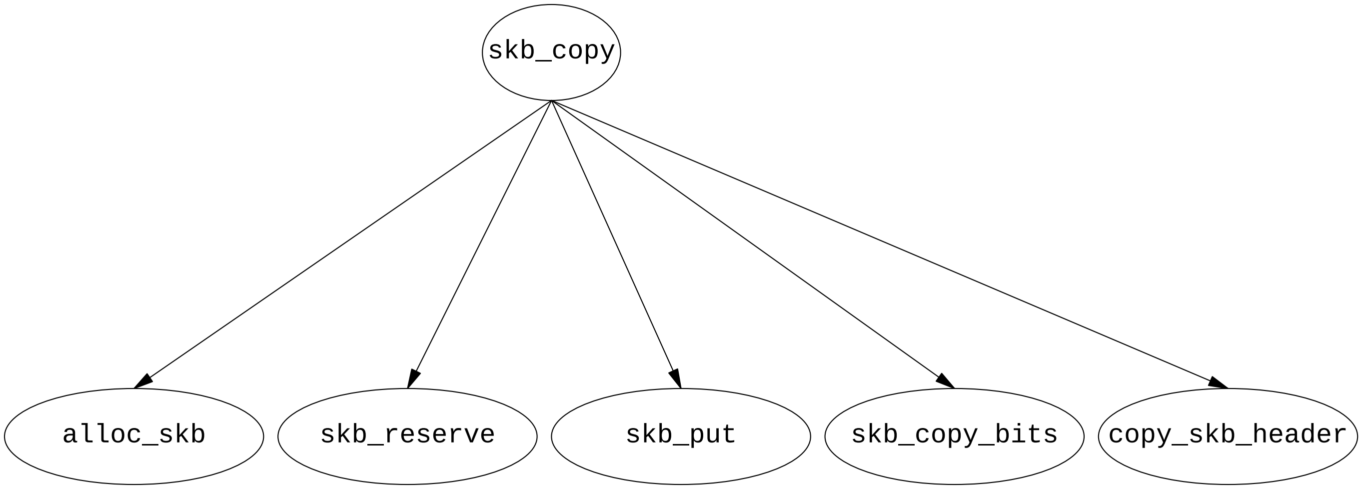
6.26. [include/linux/skbuff.h] skb_get()

It increments the number of users of the skbuff.

```
/**
 *   skb_get - reference buffer
 *   @skb: buffer to reference
 *
 *   Makes another reference to a socket buffer and returns a pointer
 *   to the buffer.
 */
static inline struct sk_buff *skb_get(struct sk_buff *skb)
{
     atomic_inc(&skb->users);
     return skb;
}

```

6.27. [net/core/skbuff.c] skb_copy()

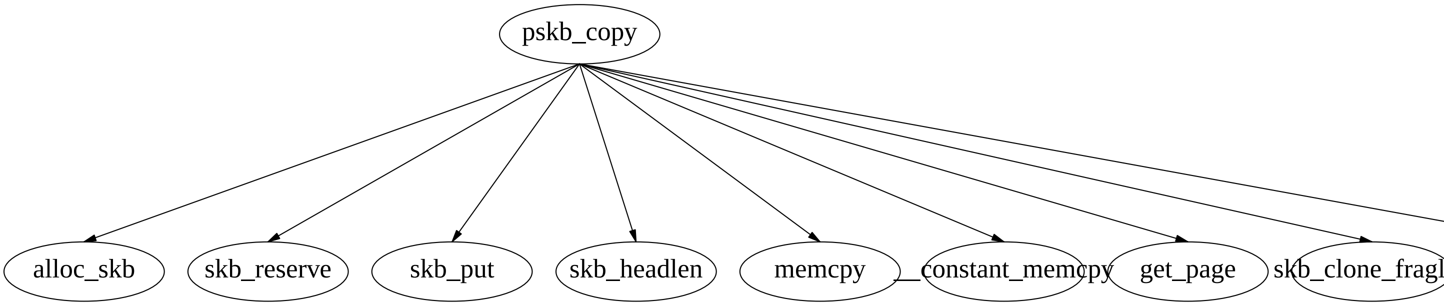
Makes a complete copy of an skbuff header and its data. It converts a nonlinear skbuff to a linear one. The headroom of the old skbuff is computed , and inappropriately called haederlen.



A linear skbuff capable of storing both the data and the headroom of the old skb is allocated with alloc_skb, this function allocates both the header and a contiguous data area. skb->len + headroom = skb->end - skb->head + skb->data_len If it is not possible to allocate such an skbuff returns NULL. A headroom equal to the one in the old skb is reserved in the new skb. It sets the tail pointer in the new skb at skb->data+ skb->len. It copies the checksum and the ip_summed flag. Then it calls the skb_copy_bits function to copy everything since the very beginning of the old skbuff (skb->head not skb->data !!!!!) to the end of the data. This means that it copies also the headroom. Finally it copies the skbuff header. and returns the pointer for the new skbuff.

```
/**
 *   skb_copy  -    create private copy of an sk_buff
 *   @skb: buffer to copy
 *   @gfp_mask: allocation priority
 *
 *   Make a copy of both an &sk_buff and its data. This is used when the
 *   caller wishes to modify the data and needs a private copy of the
 *   data to alter. Returns %NULL on failure or the pointer to the buffer
 *   on success. The returned buffer has a reference count of 1.
 *
 *   As by-product this function converts non-linear &sk_buff to linear
 *   one, so that &sk_buff becomes completely private and caller is allowed
 *   to modify all the data of returned buffer. This means that this
 *   function is not recommended for use in circumstances when only
 *   header is going to be modified. Use pskb_copy() instead.
 */

struct sk_buff *skb_copy(const struct sk_buff *skb, int gfp_mask)
{
     int headerlen = skb->data - skb->head;
     /*
      *   Allocate the copy buffer
      */
     struct sk_buff *n = alloc_skb(skb->end - skb->head + skb->data_len,
                                 gfp_mask);
     if (!n)
          return NULL;

     /* Set the data pointer */
     skb_reserve(n, headerlen);
     /* Set the tail pointer and length */
     skb_put(n, skb->len);
     n->csum        = skb->csum;
     n->ip_summed = skb->ip_summed;

     if (skb_copy_bits(skb, -headerlen, n->head, headerlen + skb->len))
          BUG();

     copy_skb_header(n, skb);
     return n;
}

```

6.28. [net/core/skbuff.c] pskb_copy()



It allocates an skb sufficient for headlen + headroom. It reserves the same headroom available in the old skbuff. It copies the headlen bytes of data from the old skbuff to the new one. Copies checksum, ip_summed flag,data_len,len from old to new. Runs through the array of fragments, copy frag descriptors from the old to the new skbuff. And for each page increments the usage count. Then it has a frag_list, it copies the pointer, and it goes through the list of fragments (frag_list) and increases the usage count (skb_clone_fraglist). Finally it copies the skb header (copy_skb_header). And it returns a pointer to the new skbuff head.

```
/**
 *   pskb_copy -    create copy of an sk_buff with private head.
 *   @skb: buffer to copy
 *   @gfp_mask: allocation priority
 *
 *   Make a copy of both an &sk_buff and part of its data, located
 *   in header. Fragmented data remain shared. This is used when
 *   the caller wishes to modify only header of &sk_buff and needs
 *   private copy of the header to alter. Returns %NULL on failure
 *   or the pointer to the buffer on success.
 *   The returned buffer has a reference count of 1.
 */

struct sk_buff *pskb_copy(struct sk_buff *skb, int gfp_mask)
{
     /*
      *   Allocate the copy buffer
      */
     struct sk_buff *n = alloc_skb(skb->end - skb->head, gfp_mask);

     if (!n)
          goto out;

     /* Set the data pointer */
     skb_reserve(n, skb->data - skb->head);
     /* Set the tail pointer and length */
     skb_put(n, skb_headlen(skb));
     /* Copy the bytes */
     memcpy(n->data, skb->data, n->len);
     n->csum        = skb->csum;
     n->ip_summed = skb->ip_summed;

     n->data_len  = skb->data_len;
     n->len         = skb->len;

     if (skb_shinfo(skb)->nr_frags) {
          int i;

          for (i = 0; i < skb_shinfo(skb)->nr_frags; i++) {
               skb_shinfo(n)->frags[i] = skb_shinfo(skb)->frags[i];
               get_page(skb_shinfo(n)->frags[i].page);
          }
          skb_shinfo(n)->nr_frags = i;
     }
     skb_shinfo(n)->tso_size = skb_shinfo(skb)->tso_size;
     skb_shinfo(n)->tso_segs = skb_shinfo(skb)->tso_segs;

     if (skb_shinfo(skb)->frag_list) {
          skb_shinfo(n)->frag_list = skb_shinfo(skb)->frag_list;
          skb_clone_fraglist(n);
     }

     copy_skb_header(n, skb);
out:
```

```
        return n;
}

```

Table of Contents

References

1. Alan Cox, “Network Buffers and Memory Management,” *Linux Journal*, 29 (September 29, 1996).