Network buffers
The BSD, Unix SVR4 and Linux approaches
(BSD4.4,SVR4.2,Linux2.6.2)

Roberto Innocente, Olumide S.Adewale

SISSA - Trieste

1. Introduction

On many operating systems, network buffers are a special category of buffers to be used for network operations. This is because they are used in a peculiar way and there is a need to make the most common of those operations extremely efficient. Most of the network operations result in :

- addition or stripping of headers or trailers
- transfer of data to/from devices
- joining or splitting buffers

According to Gilder's law, an empirical law recently being cited very often, the total communication bandwidth triples every twelve months. In contrast, according to Moore's law, cpu performance only doubles every 18 months. This will create an increasing gap between network and cpu performance.

We analyze how the traditional in-kernel networking is performed on 3 major network stacks : Bsd Unix, Unix SystemV and Linux, looking for possible improvements.

This area has recently seen the emerging of new software paradigms (User Level Networking), to keep up with the performance improvement of communications.
User-level networking gives a boost of performance pinning down some user memory where network cards perform directly their I/O operations and memory connected network cards, provide the protection mechanism by the standard page protection mechanism that most processors have.

2. I/O architecture

Some computers have a uniform view of memory and I/O devices, everything is mapped in a single address space (Vax, Alpha). On these architectures CSRs (Control and Status Registers) and memory on I/O devices are mapped in memory, and read and write memory cycles on the bus may result in reads and writes to/from the I/O devices depending on the address. Other architectures feel I/O devices are peculiar and they have a separate address space for them (x86). On these architectures a signal on the bus denotes an I/O operation instead of a standard memory operation, and special input and output instructions are used (I/O ports and I/O memory 64 KB).

3. History of network code

The roots of the BSD network code are back to the 4.2 BSD TCP/IP implementation in 1983. The 4.4 BSD Lite implementation, which we discuss appeared in 1994. The first STREAMS implementation appeared on SVR3 in 1986, then there was a complete new implementation which we discuss, on SVR4 in 1988 and later on SVR4.2MP in late 1993. The Linux TCP/IP network code started with the Ross Biro NET-1 implementation in release 0.98 in 1992 with device drivers written by Donald Becker. After R.Biro quit, then Fred van Kempen worked on the NET-2 project rewriting major parts of the initial release. NET-2 appeared publicly after Alan Cox debugging of the code as NET-2D in kernel 0.99.10 in 1993. A. Kutnezsov and A. Kleen and D.Miller. NET-3 was incorporated in 1.2.x and the current release of the network code in 2.2 kernels and on is NET-4.

4. The Berkeley approach

The organization of the network buffers in BSD derives from the observation that packets on networks are for the most part either very small or near the maximum size. This bimodal distribution of packets on networks leads to the choice of a very small fixed size combined descriptor and buffer called an `mbuf` (memory buffer), with the possibility to keep there a pointer to an external page of data in the case of a large packet. Therefore the provision of a small structure of 128 bytes, that can keep the header and the data if the packet is small.

sys/sys/mbuf.h

```
struct m_hdr {
        struct    mbuf *mh_next;       /* next buffer in chain */
        struct    mbuf *mh_nextpkt;    /* next chain in queue/record */
        caddr_t   mh_data;        /* location of data */
        int   mh_len;             /* amount of data in this mbuf */
        short mh_type;        /* type of data in this mbuf */
        short mh_flags;       /* flags; see below */
};
```

sys/sys/mbuf.h

The two possibilities are never used at the same time so that if the data is stored in the internal area, the mbuf can't use an external page and viceversa. `m_len` is the size of the data in this `mbuf,` while `m_data` is a pointer to the beginning of the data. `m_next` is a pointer that can link multiple mbufs in a chain (singly linked list). `m_pktnext` is a pointer that links multiple packets (mbuf chains) on the device or socket queue. There are 108 bytes available for data in an `mbuf,` but the code tries to be smart, keeping some space in front and after the data for the addition of headers.
Operations on networks buffers frequently result to addition or stripping of data in the beginning or the end of a packet. `Mbufs` in BSD can be joined in a single linked list through their `m_next` pointer. This is usually called an mbuff chain, and operations on packets are mapped on BSD to insertion or deletion on the head of a chain. In the first mbuf some bytes have to be used for storing the total lenght of a packet and eventually the receiving interface, these are kept in the `pkthdr` structure.

sys/sys/mbuf.h

```
struct    pkthdr {
        struct    ifnet *rcvif;       /* rcv interface */
        int   len;            /* total packet length */
};
```

sys/sys/mbuf.h

The total length of a packet split in multiple mbufs is stored in the `m_pkthdr.len` field of the first mbuf of the chain, while in this case `m_len` is only the amount of data in each mbuf. In the first `mbuf` of a chain, 8 bytes of these 108 available are used to specify the length and interface for the packet and so only 100 bytes of data are available in it. If the data is from 0 to 84 bytes, then a single mbuf is allocated and the data is placed in it leaving 16 bytes free in front of it for an eventual header. If the data is from 85 to 100 bytes then the data is placed at the beginning of the mbuf. If the data is from 101 to 207 bytes[*] a chain of 2 mbufs is allocated and the data is placed starting at the beginning of the first mbuf. If the data is more than 207 then an mbuf with an external data area is allocated. In this case the mbuf has a pointer to the external

[*] in fact there would be space for up to 208 bytes, but because of a little mistake in the code that uses < instead of <=, this is what happens

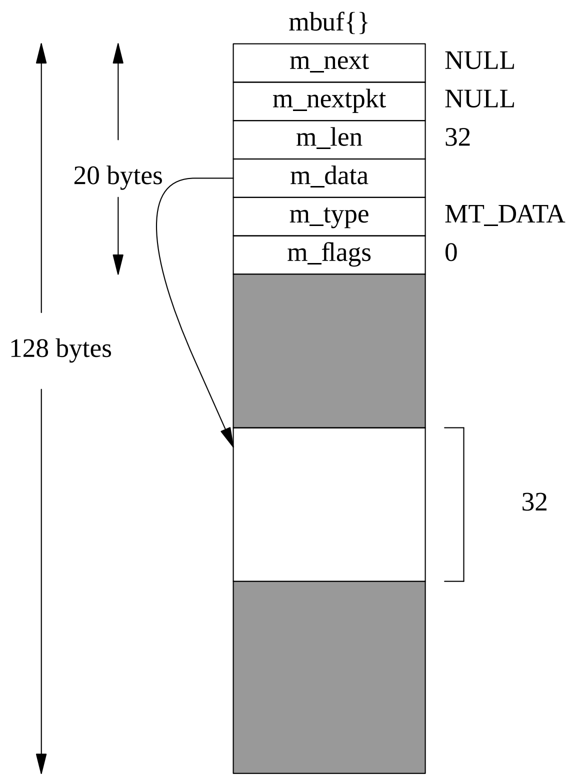


Figure 1 An mbuf with some data

data area (also called a page cluster : because it is the same page used for virtual memory) and the data is stored from the beginning of the pagecluster.

sys/sys/mbuf.h

```
struct m_ext {
        caddr_t   ext_buf;          /* start of buffer */
        void (*ext_free)();         /* free routine if not the usual */
        u_int ext_size;         /* size of buffer, for ext_free */
};
```

sys/sys/mbuf.h

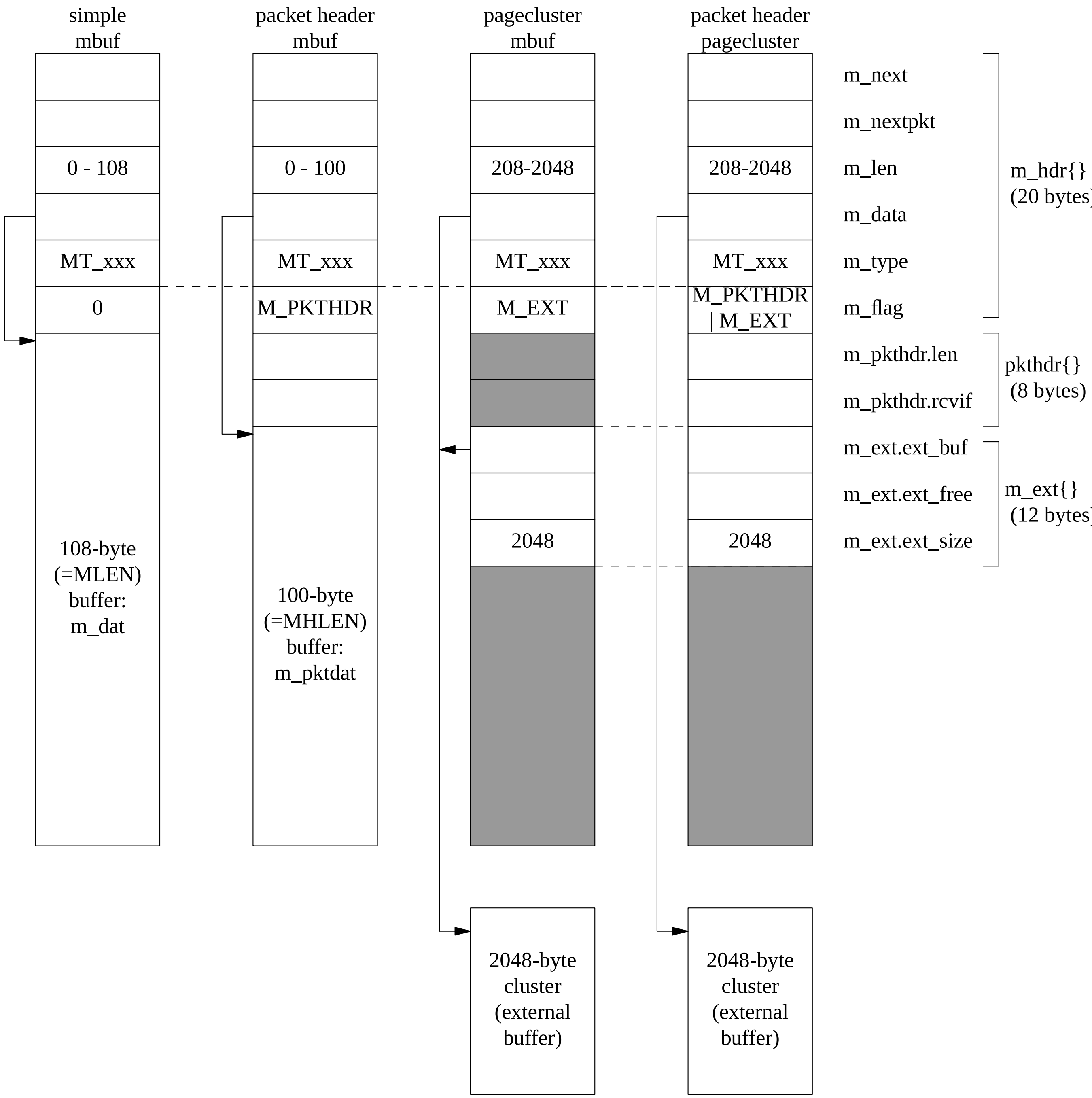


Figure 2 Various mbuf types

Instead of using a general structure with all the fields necessary, BSD chooses to save space and uses a sovrapposition of them specified by a C union. The different layout used is chosen by the `M_EXT`, `M_PKTHDR` flags :

sys/sys/mbuf.h

```
89    struct mbuf {
90          struct    m_hdr m_hdr;
```

```
        union {
            struct {
                struct    pkthdr MH_pkthdr;    /* M_PKTHDR set */
                union {
                    struct    m_ext MH_ext;   /* M_EXT set */
                    char MH_databuf[MHLEN];
                } MH_dat;
            } MH;
            char M_databuf[MLEN];          /* !M_PKTHDR, !M_EXT */
        } M_dat;
    };
    #define   m_next         m_hdr.mh_next
    #define   m_len     m_hdr.mh_len
    #define   m_data         m_hdr.mh_data
    #define   m_type         m_hdr.mh_type
    #define   m_flags        m_hdr.mh_flags
    #define   m_nextpkt m_hdr.mh_nextpkt
    #define   m_act     m_nextpkt
    #define   m_pkthdr  M_dat.MH.MH_pkthdr
    #define   m_ext     M_dat.MH.MH_dat.MH_ext
    #define   m_pktdat  M_dat.MH.MH_dat.MH_databuf
    #define   m_dat     M_dat.M_databuf
```

sys/sys/mbuf.h

4.1. Memory issues

Mbuf clusters are allocated with the standard kernel memory allocator malloc and mapped in a dedicated area of the kernel virtual space. An initial allocation is done at system startup and it can be expanded up to a configurable maximum (sysctl). Memory allocated to mbuf clusters is never given back. An array of reference counts `mclrefcnt` is statically allocated at system startup, its size is `NMBCLUSTERS` that was 512 for gateways and 256 for other systems. But on the x86 there is no need for page clustering like on the Vax and so CLSIZE=1 and NBPG=4096, and so CLBYTES=CLSIZE*NBPG=4096 and MCLBYTES=1024. Then the size of the kernel window into which to map mbuf clusters is 2 MB in pages NKMEMCLUSTERS=(2048*1024/CLBYTES)=512. The initial memory mapped for mbuf clusters is `VM_MBUF_SIZE=NMBCLUSTERS*MCLSIZE=512*1024` enough to map the max number of clusters(kmem_suballoc). Then every time this is needed a page is allocated with kmem_alloc and assigned to the mb_map (the window in the kernel virtual space dedicated to mbuf clusters) if there are still free slots.

4.2. mbuf macros and functions

There are many macros and functions defined to help with the use of mbufs. mbufs are allocated at the device level for an incoming packet with `m_devget:`

usr/src/sys/kern/uipc_mbuf.c

```
    struct mbuf *
    m_devget(buf, totlen, off0, ifp, copy)
        char *buf;
        int totlen, off0;
        struct ifnet *ifp;
        void (*copy)();
```

usr/src/sys/kern/uipc_mbuf.c

This function according to the size of the packet will allocate 1 or 2 normal mbufs or an mbuf with an external cluster, and will copy the data from the linear device buffer to the mbuf chain. In other places mbufs are allocated with the m_get function or MGET macro :

usr/src/sys/kern/uipc_mbuf.c

```
struct mbuf *
m_get(nowait, type)
        int nowait, type;
{
        register struct mbuf *m;

        MGET(m, nowait, type);
        return (m);
}
```

usr/src/sys/kern/uipc_mbuf.c

usr/src/sys/sys/mbuf.h

```
#define    MGET(m, how, type) { \
        MALLOC((m), struct mbuf *, MSIZE, mbtypes[type], (how)); \
        if (m) { \
                (m)->m_type = (type); \
                MBUFLOCK(mbstat.m_mtypes[type]++;) \
                (m)->m_next = (struct mbuf *)NULL; \
                (m)->m_nextpkt = (struct mbuf *)NULL; \
                (m)->m_data = (m)->m_dat; \
                (m)->m_flags = 0; \
        } else \
                (m) = m_retry((how), (type)); \
}
```

usr/src/sys/sys/mbuf.h

mbufs that have to be packet headers are allocated instead with m_gethdr and MGETHDR function and macro respectively, that set the M_PKTHDR flag in the m_flags field of the mbuf header and the m_data pointer is initialized to point after the pkthdr structure. External pages are allocated for mbufs using the MCLGET macro :

usr/src/sys/sys/mbuf.h

```
#define    MCLGET(m, how) \
        { MCLALLOC((m)->m_ext.ext_buf, (how)); \
          if ((m)->m_ext.ext_buf != NULL) { \
                (m)->m_data = (m)->m_ext.ext_buf; \
                (m)->m_flags |= M_EXT; \
                (m)->m_ext.ext_size = MCLBYTES;  \
          } \
        }
```

usr/src/sys/sys/mbuf.h

This function allocates an external page and sets the m_data pointer in the existing mbuf to point to the beginning of the allocated page. mbufs are freed using the macro MFREE or the functions m_free or m_freem :

usr/src/sys/sys/mbuf.h

```
#define   MFREE(m, nn) \
      { MBUFLOCK(mbstat.m_mtypes[(m)->m_type]--;) \
        if ((m)->m_flags & M_EXT) { \
           MCLFREE((m)->m_ext.ext_buf); \
        } \
        (nn) = (m)->m_next; \
        FREE((m), mbtypes[(m)->m_type]); \
      }
```

usr/src/sys/sys/mbuf.h

m_free frees just one mbuf, while m_freem frees all the mbufs in the chain. It frequently happens you want to put some data at the end of a buffer to leave all the possible room for headers. The M_ALIGN and MH_ALIGN macros conveniently set the m_data pointer of an mbuf to place an object of size len at the end of the mbuf :

usr/src/sys/sys/mbuf.h

```
#define   M_ALIGN(m, len) \
      { (m)->m_data += (MLEN - (len)) &~ (sizeof(long) - 1); }
/*
 * As above, for mbufs allocated with m_gethdr/MGETHDR
 * or initialized by M_COPY_PKTHDR.
 */
#define   MH_ALIGN(m, len) \
      { (m)->m_data += (MHLEN - (len)) &~ (sizeof(long) - 1); }

```

sys/sys/mbuf.h

The M_PREPEND macro instead is used to prepend len bytes at the head of the current data in the mbuf. If there is not enough space, a new mbuf is linked in front of the mbuf chain :

usr/src/sys/sys/mbuf.h

```
#define   M_PREPEND(m, plen, how) { \
      if (M_LEADINGSPACE(m) >= (plen)) { \
           (m)->m_data -= (plen); \
           (m)->m_len += (plen); \
      } else \
           (m) = m_prepend((m), (plen), (how)); \
      if ((m) && (m)->m_flags & M_PKTHDR) \
           (m)->m_pkthdr.len += (plen); \
}
```

sys/sys/mbuf.h

m_adj is a function that is used to trim len bytes from the head or the tail of the data :

sys/kern/uipc_mbuf.c

```
void
m_adj(mp, req_len)
      struct mbuf *mp;
      int req_len;
```

sys/kern/uipc_mbuf.c

m_cat is used instead to concatenate two mbufs together :

sys/kern/uipc_mbuf.c

```
void
m_cat(m, n)
      register struct mbuf *m, *n;
{
      while (m->m_next)
            m = m->m_next;
      while (n) {
            if (m->m_flags & M_EXT ||
                m->m_data + m->m_len + n->m_len >= &m->m_dat[MLEN]) {
                  /* just join the two chains */
                  m->m_next = n;
                  return;
            }
            /* splat the data from one into the other */
            bcopy(mtod(n, caddr_t), mtod(m, caddr_t) + m->m_len,
                (u_int)n->m_len);
            m->m_len += n->m_len;
            n = m_free(n);
      }
}
```

sys/kern/uipc_mbuf.c

There are different functions to copy mbufs m_copy, m_copydata, m_copyback, m_copym. m_copym creates a new mbuf chain and copies len bytes starting at offset `off0` from the old mbuf chain, m_copy is the same as the m_copym except that it calls m_copym with a forth argument of nowait. If the len argument is M_COPYALL, then all the remaining data in the mbuff after the offset is copied.

usr/src/sys/sys/mbuf.h

```
/* compatiblity with 4.3 */
#define  m_copy(m, o, l)  m_copym((m), (o), (l), M_DONTWAIT)
```

[usr/src/sys/kern/uipc_mbuf.c]

```
struct mbuf *
m_copym(m, off0, len, wait)
      register struct mbuf *m;
      int off0, wait;
      register int len;
{
      register struct mbuf *n, **np;
      register int off = off0;
      struct mbuf *top;
      int copyhdr = 0;

      if (off < 0 || len < 0)
            panic("m_copym");
      if (off == 0 && m->m_flags & M_PKTHDR)
            copyhdr = 1;
      while (off > 0) {
            if (m == 0)
                  panic("m_copym");
            if (off < m->m_len)
                  break;
            off -= m->m_len;
            m = m->m_next;
```

```
275         }
276         np = &top;
277         top = 0;
278         while (len > 0) {
279              if (m == 0) {
280                   if (len != M_COPYALL)
281                        panic("m_copym");
282                   break;
283              }
284              MGET(n, wait, m->m_type);
285              *np = n;
286              if (n == 0)
287                   goto nospace;
288              if (copyhdr) {
289                   M_COPY_PKTHDR(n, m);
290                   if (len == M_COPYALL)
291                        n->m_pkthdr.len -= off0;
292                   else
293                        n->m_pkthdr.len = len;
294                   copyhdr = 0;
295              }
296              n->m_len = min(len, m->m_len - off);
297              if (m->m_flags & M_EXT) {
298                   n->m_data = m->m_data + off;
299                   mclrefcnt[mtocl(m->m_ext.ext_buf)]++;
300                   n->m_ext = m->m_ext;
301                   n->m_flags |= M_EXT;
302              } else
303                   bcopy(mtod(m, caddr_t)+off, mtod(n, caddr_t),
304                       (unsigned)n->m_len);
305              if (len != M_COPYALL)
306                   len -= n->m_len;
307              off = 0;
308              m = m->m_next;
309              np = &n->m_next;
310         }
311         if (top == 0)
312              MCFail++;
313         return (top);
314    nospace:
315         m_freem(top);
316         MCFail++;
317         return (0);
318    }
```

[usr/src/sys/kern/uipc_mbuf.c]

264-265 If the offset off is less than zero or the len requested is negative, then the kernel panics.

266-267 If a packet header mbuf is being copied since its beginning (from offset 0), then a flag is set to force the copy of the packet header information (copyhdr).

268-275 The code runs through the mbuf chain skipping mbufs until offset bytes has being skipped. If this is not possible because the end of the chain is reached then it panics with the m_copym message.

278-310 This while loop tries to copy len bytes from the current position in the old mbuf chain to a chain of newly allocated mbufs. If the argument len is M_COPYALL, then it copies up to the end of the old mbuf chain. The pointer to the first mbuf allocated is kept in top. For the first mbuf allocated if the copyhdr is set, then the packet header info are copied (the len is decreased by the offset that is skipped). The data inside the mbuf is then copied, but if the mbuf had an external cluster, then the data is not copied, only the pointers are copied and the refcount for the pagecluster is incremented.

311- The new mbuf chain is then returned.

`m_copydata` copies data from an mbuf chain to a linear buffer starting at offset `off` bytes and m_copyback copies from a linear buffer to an mbuf chain starting `off` bytes from the beginning :

[usr/src/sys/kern/uipc_mbuf.c]

```
324  void
325  m_copydata(m, off, len, cp)
326       register struct mbuf *m;
327       register int off;
328       register int len;
329       caddr_t cp;
330  {
331       register unsigned count;
332
333       if (off < 0 || len < 0)
334            panic("m_copydata");
335       while (off > 0) {
336            if (m == 0)
337                 panic("m_copydata");
338            if (off < m->m_len)
339                 break;
340            off -= m->m_len;
341            m = m->m_next;
342       }
343       while (len > 0) {
344            if (m == 0)
345                 panic("m_copydata");
346            count = min(m->m_len - off, len);
347            bcopy(mtod(m, caddr_t) + off, cp, count);
348            len -= count;
349            cp += count;
350            off = 0;
351            m = m->m_next;
352       }
353  }
```

[usr/src/sys/kern/uipc_mbuf.c]

333-334 If offset or len is negative something is wrong and the kernel will panic. 335-342 All the mbufs in the chain prior to `off` offset bytes are skipped, if it happens that there are not enough bytes in the chain then the kernel will panic

342-352 The code runs through the mbuf chain and copies up to `len` bytes from the mbuf chain to the destionation buffer pointed to by `cp`, if the data finish before all the requested bytes have been copied then the kernel will panic.

[usr/src/sys/net/rtsock.c]

```
void
m_copyback(m0, off, len, cp)
     struct    mbuf *m0;
     register int off;
```

```
381         register int len;
382         caddr_t cp;
383    {
384         register int mlen;
385         register struct mbuf *m = m0, *n;
386         int totlen = 0;
387
388         if (m0 == 0)
389              return;
390         while (off > (mlen = m->m_len)) {
391              off -= mlen;
392              totlen += mlen;
393              if (m->m_next == 0) {
394                   n = m_getclr(M_DONTWAIT, m->m_type);
395                   if (n == 0)
396                        goto out;
397                   n->m_len = min(MLEN, len + off);
398                   m->m_next = n;
399              }
400              m = m->m_next;
401         }
402         while (len > 0) {
403              mlen = min (m->m_len - off, len);
404              bcopy(cp, off + mtod(m, caddr_t), (unsigned)mlen);
405              cp += mlen;
406              len -= mlen;
407              mlen += off;
408              off = 0;
409              totlen += mlen;
410              if (len == 0)
411                   break;
412              if (m->m_next == 0) {
413                   n = m_get(M_DONTWAIT, m->m_type);
414                   if (n == 0)
415                        break;
416                   n->m_len = min(MLEN, len);
417                   m->m_next = n;
418              }
419              m = m->m_next;
420         }
421    out: if (((m = m0)->m_flags & M_PKTHDR) && (m->m_pkthdr.len < totlen))
422              m->m_pkthdr.len = totlen;
423    }
```

[usr/src/sys/net/rtsock.c]

388-389 If the mbuf chain is empty then the function returns

390-401 It skips `off` bytes along the mbuf chain, eventually allocating new zeroed mbufs that are appended to the end of the chain

402-420 `len` bytes are now copied from the linear buffer pointed by `cp` to the mbuf chain, allocating all eventually needed mbufs. 421-422 If the original first mbuf was a packet header, then the total packet length is adjusted in the packet header according to the new size.

The `M_COPY_PKTHDR` macro copies the information used by packet header mbufs (pkthdr.len and pkthdr.rcvif), sets the flag M_COPYFLAGS and the m_data pointer to point just immediately after the packet header info.

[usr/src/sys/sys/uipc_mbuf.h]

```
#define    M_COPY_PKTHDR(to, from) { \
        (to)->m_pkthdr = (from)->m_pkthdr; \
        (to)->m_flags = (from)->m_flags & M_COPYFLAGS; \
        (to)->m_data = (to)->m_pktdat; \
}
```

[usr/src/sys/sys/uipc_mbuf.h]

`m_pullup` rearranges an mbuf chain so that the first len bytes are stored contiguously inside the first mbuf of the chain, it is used to keep the headers contiguous, to allow the parsing of them with common pointer arithmetic.

usr/src/sys/kern/uipc_mbuf.c

```
struct mbuf *
m_pullup(n, len)
        register struct mbuf *n;
        int len;
{
        register struct mbuf *m;
        register int count;
        int space;

        /*
         * If first mbuf has no cluster, and has room for len bytes
         * without shifting current data, pullup into it,
         * otherwise allocate a new mbuf to prepend to the chain.
         */
        if ((n->m_flags & M_EXT) == 0 &&
            n->m_data + len < &n->m_dat[MLEN] && n->m_next) {
                if (n->m_len >= len)
                        return (n);
                m = n;
                n = n->m_next;
                len -= m->m_len;
        } else {
                if (len > MHLEN)
                        goto bad;
                MGET(m, M_DONTWAIT, n->m_type);
                if (m == 0)
                        goto bad;
                m->m_len = 0;
                if (n->m_flags & M_PKTHDR) {
                        M_COPY_PKTHDR(m, n);
                        n->m_flags &= ~M_PKTHDR;
                }
        }
        space = &m->m_dat[MLEN] - (m->m_data + m->m_len);
        do {
                count = min(min(max(len, max_protohdr), space), n->m_len);
                bcopy(mtod(n, caddr_t), mtod(m, caddr_t) + m->m_len,
                  (unsigned)count);
                len -= count;
```

```
504                 m->m_len += count;
505                 n->m_len -= count;
506                 space -= count;
507                 if (n->m_len)
508                      n->m_data += count;
509                 else
510                      n = m_free(n);
511            } while (len > 0 && n);
512            if (len > 0) {
513                 (void) m_free(m);
514                 goto bad;
515            }
516            m->m_next = n;
517            return (m);
518       bad:
519            m_freem(n);
520            MPFail++;
521            return (0);
522       }
```

usr/src/sys/kern/uipc_mbuf.c

479-485 If the mbuf is not using a pagecluster, and there is enough room to keep all the `len` bytes inside it, then if the data in the first mbuf is already more than the data requested the mbuf pointer is returned without any further action. Otherwise `m->m_len` bytes are already in the first mbuf, and only `len-m->m_len` bytes need to be copied at the tail of the current data in the mbuf.

486-488 If the request was for more bytes than a packet header mbuf can keep (MHLEN = 100) then it tries to free the mbuf and returns an error.

489-491 It allocates a new mbuf of the same type, if the allocation is not successful then it returns an error.

492-497 The size of the data in the newly allocated mbuf is set to 0. If it was a packet header then the header data is copied from the old mbuf to this new one (pkthdr.len and pkthdr.rcvif), and then sets the m_data pointer to point just after the packet header information. Finally the M_PKTDHR flag is reset in the old mbuf, that will not be anymore a packet header.

The situation at this point is that we have allocated a new mbuf to prepend to the chain or not, in any case m points to what should be the first mbuf in the chain and n points to the second.

498 The space available at the end of the data in the `mbuf` is the difference between the two pointers, the one that points to the last byte of the mbuf (`&m->m_dat[MLEN]`) and the one that points to the last byte of the stored data (`m->m_data+m->m_len`).

499- It runs through the mbuf chain, copying until enough bytes have been copied, and eventually freeing the mbufs that have been completely exhausted. (m_free returns a pointer to the second mbuf in the chain if it exists)

512- If it was not possible to copy the requested number of bytes then the first mbuf is freed, and an error is returned. 516- This is the usual exit of the function. It links the first mbuf with the chain of other mbufs, and the complete chain is then returned. The dtom and mtod macros are frequently used :

usr/src/sys/sys/mbuf.h

```
#define   mtod(m,t)  ((t)((m)->m_data))
#define   dtom(x)         ((struct mbuf *)((long)(x) & ~(MSIZE-1)))
#define   mtocl(x)   (((u_long)(x) - (u_long)mbutl) >> MCLSHIFT)
#define   cltom(x)   ((caddr_t)((u_long)mbutl + ((u_long)(x) << MCLSHIFT)))
```

usr/src/sys/sys/mbuf.h

the mtod macro casts the m_data pointer to the correct type, the dtom macro gets the mbuf pointer from the

the m_data pointer truncating it to the previous 128-bytes(MSIZE) aligned address.

4.3. BSD Sharing of buffers

BSD allows only the sharing of external pages(page clusters). This is done keeping a reference count for each page cluster, the `mclrefcnt` array. The m_copy and m_free functions use this counter if the mbuf has an associated page cluster. m_copy instead of copying the data just increments the mclrefcnt counter and m_free puts the page cluster back in the free list only if the reference count becomes zero.

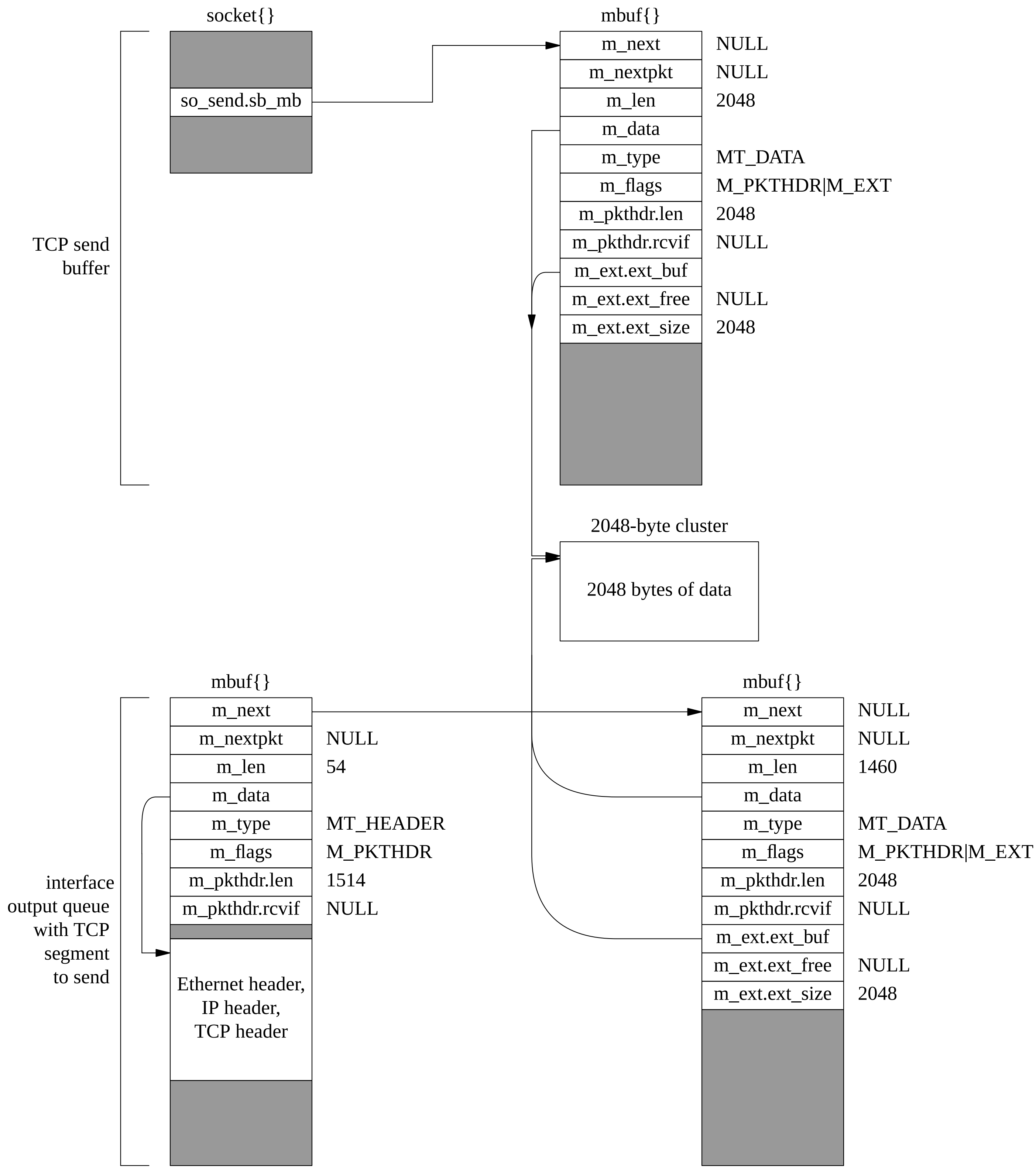


Figure 3 Sharing of a pagecluster

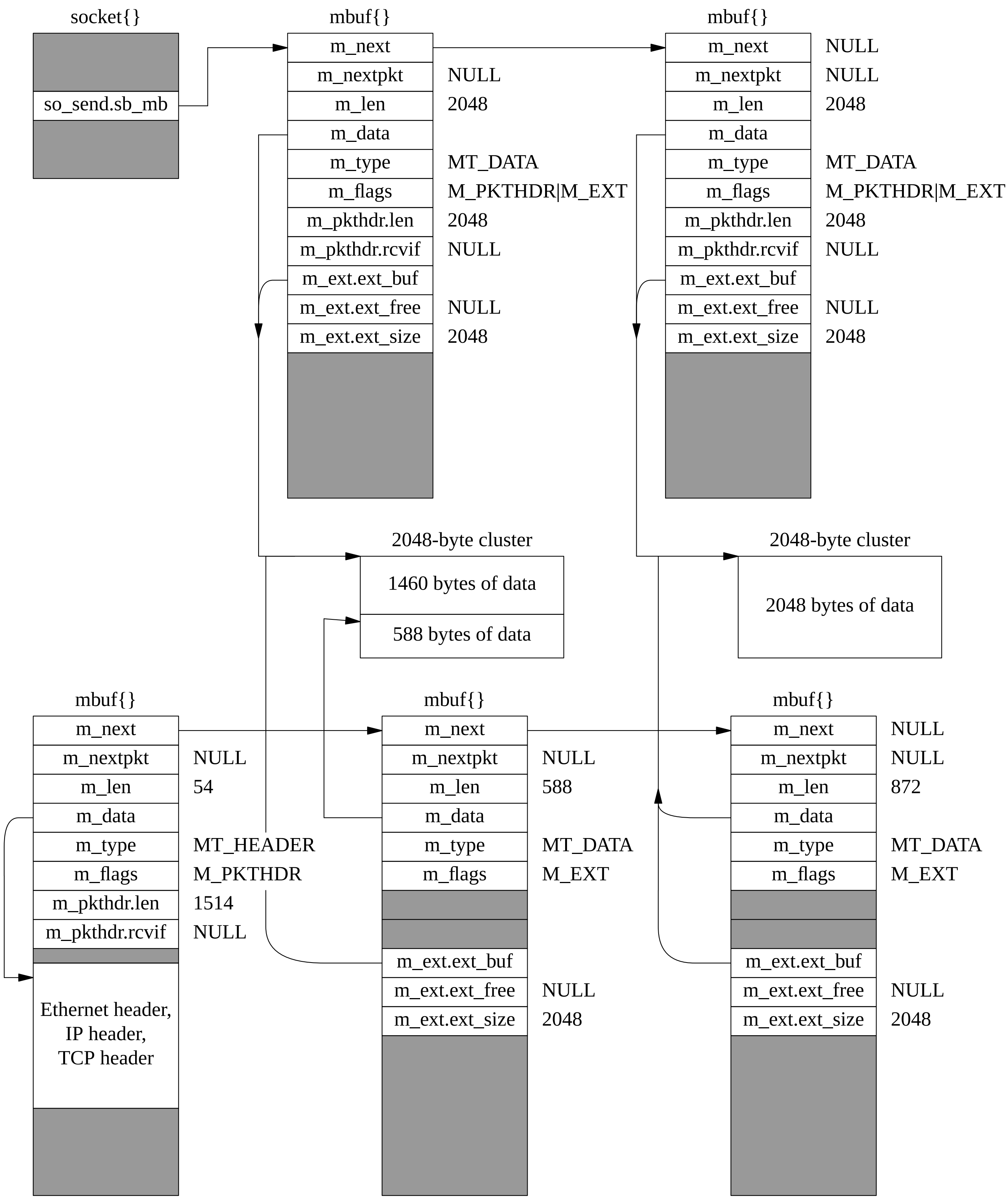


Figure 4 Partial sharing of a pageclusters

4.4. Queues of packets and mbuf chains

As we have seen a packet on BSD is stored as a single mbuf or a chain of mbufs linked through the `m_next` pointer. Packets are linked on queues (for instance the device or socket queue) through the `m_nextpkt` pointer of the only or the first mbuf of the chain.

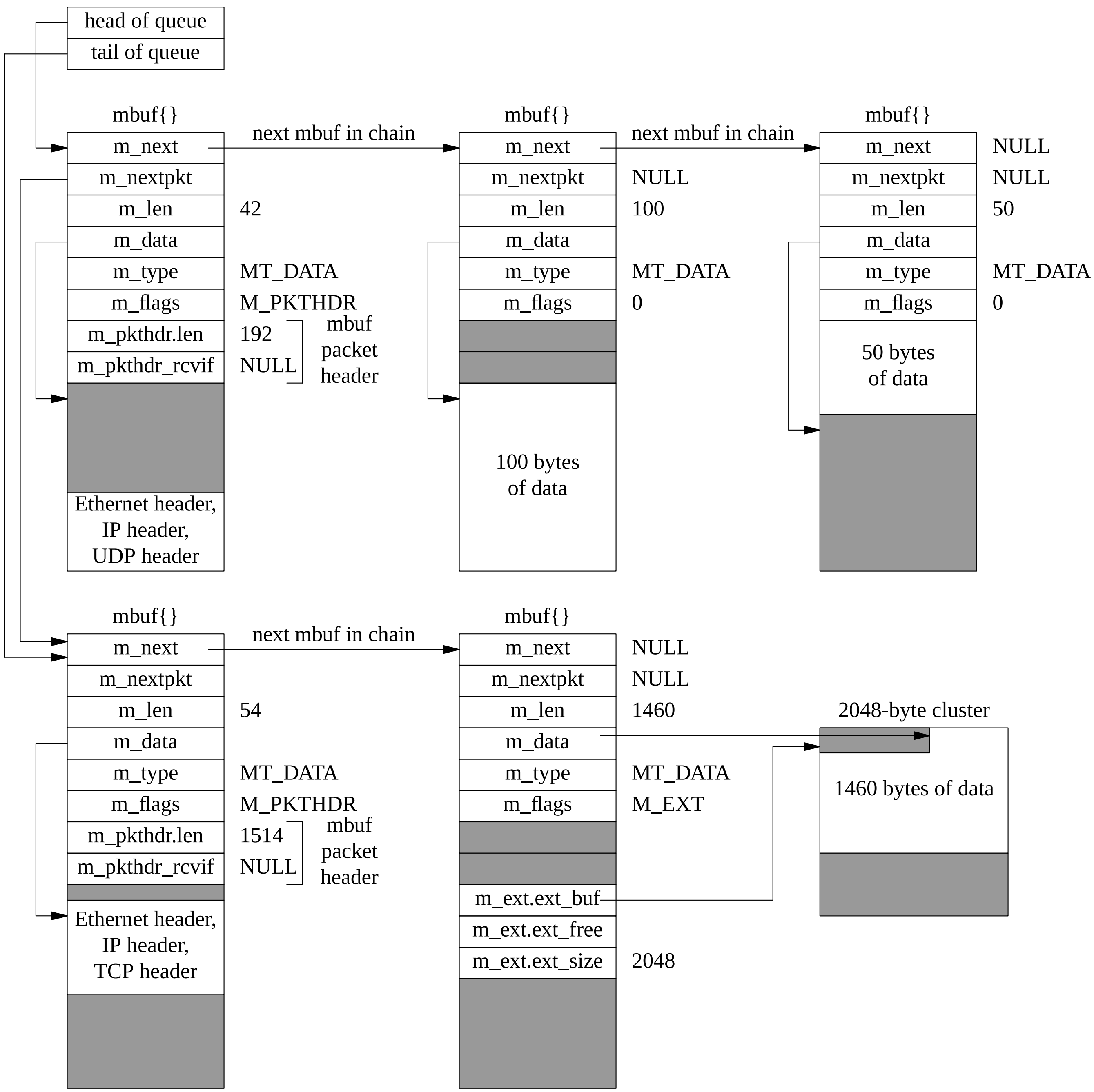


Figure 5 A UDP and a TCP packets enqueued, possibly at the device queue

4.5. TCP packet creation

The code allocates a header mbuf for the ip and tcp header using the `MGETHDR` macro. The header is then created leaving in front of it a space sufficient for a maximum link header (this parameter is sysctl configurable with a lower bound of 16, sufficient for storing a 14 byte Ethernet header and keeping the IP header 16 byte aligned for efficiency reason).

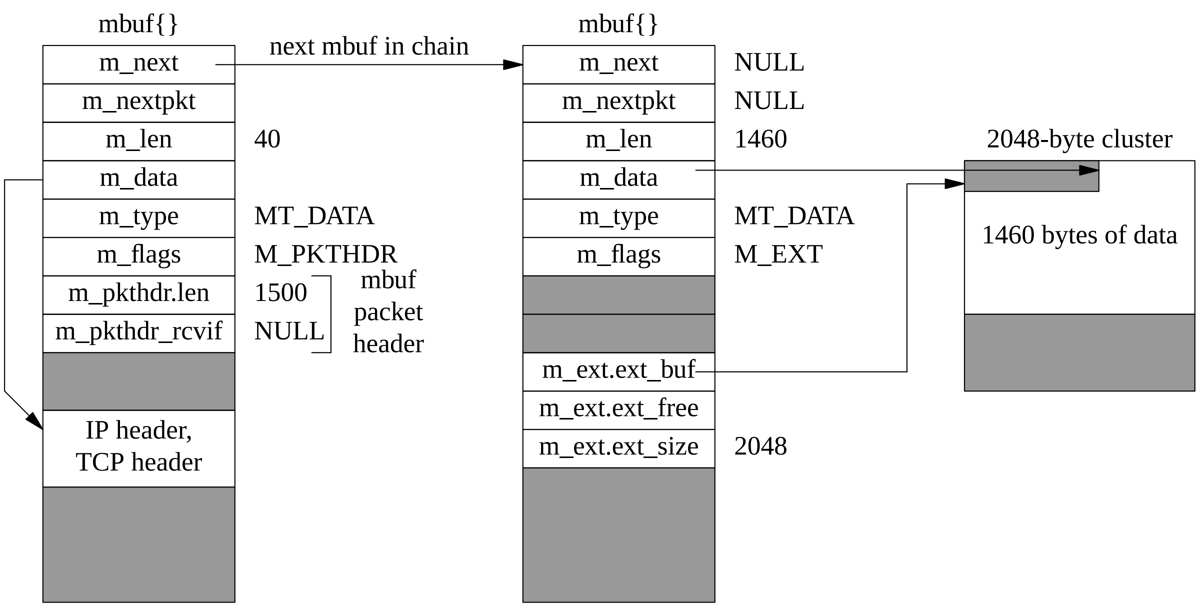


Figure 6 A TCP/IP packet with with headers in a prepended mbuf

If the data in the mbuf chain can fit inside it, then it is copied inside this header mbuf. Otherwise the header mbuf is prepended to the mbuf chain created by m_copy (we know that this function doesnt copy external pageclusters, just copies the pointer to them).

[sys/netinet/tcp_output.c]

```
        MGETHDR(m, M_DONTWAIT, MT_HEADER);
        if (m == NULL) {
             error = ENOBUFS;
             goto out;
        }
        m->m_data += max_linkhdr;
        m->m_len = hdrlen;
        if (len <= MHLEN - hdrlen - max_linkhdr) {
             m_copydata(so->so_snd.sb_mb, off, (int) len,
                 mtod(m, caddr_t) + hdrlen);
             m->m_len += len;
        } else {
             m->m_next = m_copy(so->so_snd.sb_mb, off, (int) len);
             if (m->m_next == 0) {
                  (void) m_free(m);
                  error = ENOBUFS;
                  goto out;
             }
        }
```

[sys/netinet/tcp_output.c]

4.6. UDP packet creation

The addition of IP and UDP headers is done prepending an mbuf to the mbuf chain where the udp and ip headers are placed at the end of its data region to allow for the addition of whatever header is eventually needed.

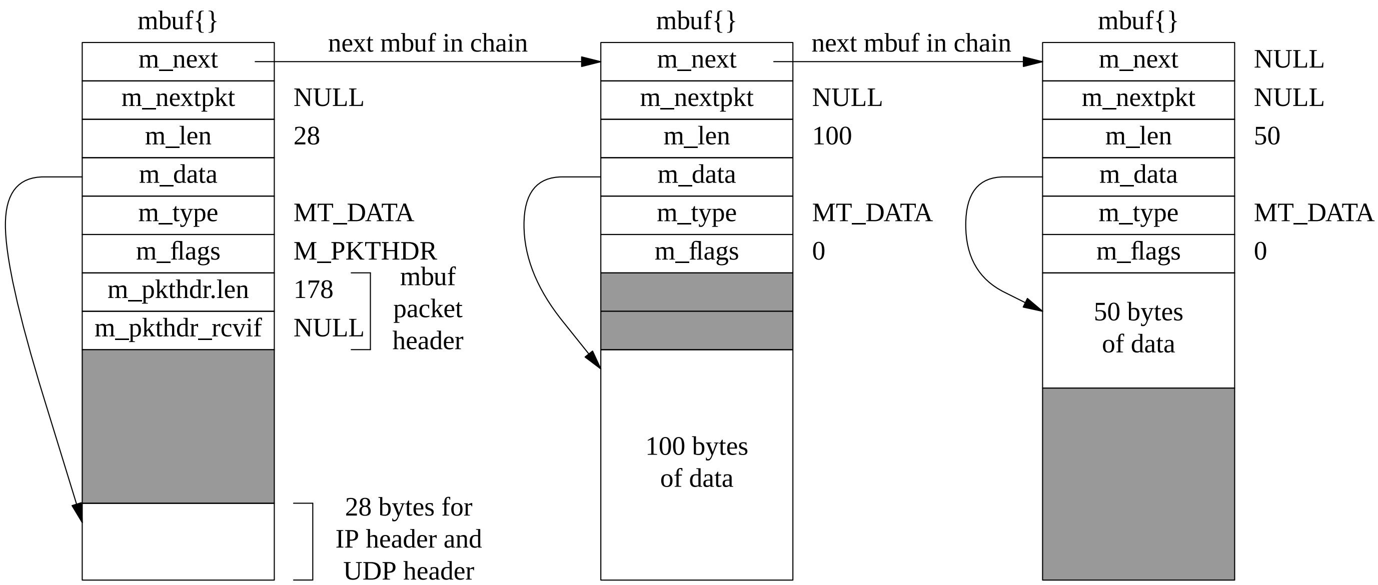


Figure 7 A UDP/IP packet with headers

[sys/netinet/udp_usrreq.c]

```
        /*
         * Calculate data length and get a mbuf
         * for UDP and IP headers.
         */
        M_PREPEND(m, sizeof(struct udpiphdr), M_DONTWAIT);
        if (m == 0) {
             error = ENOBUFS;
             goto release;
        }

        /*
         * Fill in mbuf with extended UDP header
         * and addresses and length put into network format.
         */
        ui = mtod(m, struct udpiphdr *);
        ui->ui_next = ui->ui_prev = 0;
        ui->ui_x1 = 0;
        ui->ui_pr = IPPROTO_UDP;
        ui->ui_len = htons((u_short)len + sizeof (struct udphdr));
        ui->ui_src = inp->inp_laddr;
        ui->ui_dst = inp->inp_faddr;
        ui->ui_sport = inp->inp_lport;
```

```
        ui->ui_dport = inp->inp_fport;
        ui->ui_ulen = ui->ui_len;

        /*
         * Stuff checksum and output datagram.
         */
        ui->ui_sum = 0;
        if (udpcksum) {
            if ((ui->ui_sum = in_cksum(m, sizeof (struct udpiphdr) + len)) == 0)
             ui->ui_sum = 0xffff;
        }
        ((struct ip *)ui)->ip_len = sizeof (struct udpiphdr) + len;
        ((struct ip *)ui)->ip_ttl = inp->inp_ip.ip_ttl;     /* XXX */
        ((struct ip *)ui)->ip_tos = inp->inp_ip.ip_tos;     /* XXX */
        udpstat.udps_opackets++;
        error = ip_output(m, inp->inp_options, &inp->inp_route,
            inp->inp_socket->so_options & (SO_DONTROUTE | SO_BROADCAST),
            inp->inp_moptions);
```

[sys/netinet/udp_usrreq.c]

4.7. Ethernet header addition

The addition of the ethernet header happens in the `ether_output` function, where it uses the M_PREPEND macro to verify that there are 14 bytes available in front of the packet.

[sys/net/if_ethersubr.c]

```
        M_PREPEND(m, sizeof (struct ether_header), M_DONTWAIT);
        if (m == 0)
             senderr(ENOBUFS);
        eh = mtod(m, struct ether_header *);
        type = htons((u_short)type);
        bcopy((caddr_t)&type,(caddr_t)&eh->ether_type,
             sizeof(eh->ether_type));
        bcopy((caddr_t)edst, (caddr_t)eh->ether_dhost, sizeof (edst));
        bcopy((caddr_t)ac->ac_enaddr, (caddr_t)eh->ether_shost,
            sizeof(eh->ether_shost));
```

[sys/net/if_ethersubr.c]

This funtion is called inside [sys/netinet/ip_output.c] ip_output() as the if_output method for an ethernet network device.

[sys/netinet/ip_output.c]

```
sendit:
        /*
         * If small enough for interface, can just send directly.
         */
        if ((u_short)ip->ip_len <= ifp->if_mtu) {
             ip->ip_len = htons((u_short)ip->ip_len);
             ip->ip_off = htons((u_short)ip->ip_off);
             ip->ip_sum = 0;
             ip->ip_sum = in_cksum(m, hlen);
             error = (*ifp->if_output)(ifp, m,
                        (struct sockaddr *)dst, ro->ro_rt);
```

```
                goto done;
        }
```

[sys/netinet/ip_output.c]

The linearization of the mbuf chains happens in BSD for example in the `leput` (Lance ethernet put) function for an ethernet interface. This function in the BSD code copies all the data from the mbuf chain contiguously to the hardware buffer.

[sys/hp300/dev/if_le.c]

```
/*
 * Routine to copy from mbuf chain to transmit
 * buffer in board local memory.
 */
leput(lebuf, m)
        register char *lebuf;
        register struct mbuf *m;
{
        register struct mbuf *mp;
        register int len, tlen = 0;

        for (mp = m; mp; mp = mp->m_next) {
                len = mp->m_len;
                if (len == 0)
                        continue;
                tlen += len;
                bcopy(mtod(mp, char *), lebuf, len);
                lebuf += len;
        }
        m_freem(m);
        if (tlen < LEMINSIZE) {
                bzero(lebuf, LEMINSIZE - tlen);
                tlen = LEMINSIZE;
        }
        return(tlen);
}
```

[sys/hp300/dev/if_le.c]

4.8. IP fragmentation/defragmentation in BSD

IP fragments are recognized by having the MF bit set to 1 or the IP offset field different from 0 in the IP header. BSD stores such datagrams in a queue specific for the IP source and destination addresses. The IP source and destination addresses are stored in the head of the queue.

[sys/netinet/ip_var.h]

```
struct ipq {
        struct    ipq *next,*prev;     /* to other reass headers */
        u_char    ipq_ttl;        /* time for reass q to live */
        u_char    ipq_p;               /* protocol of this fragment */
        u_short   ipq_id;              /* sequence id for reassembly */
        struct    ipasfrag *ipq_next,*ipq_prev;
                            /* to ip headers of fragments */
        struct    in_addr ipq_src,ipq_dst;
```

```
62    };
```

[sys/netinet/ip_var.h]

The IP address fields inside the IP packets are re-used to store backward and forward pointers between IP fragments to keep them linked in a doubly linked list. The mbuf structure can easily be retrieved from the m->data pointer by simply truncating it to a multiple of the mbuf size (MSIZE=128), as it is done by the dtom macro :

[sys/mbuf.h]

```
62    #define    dtom(x)        ((struct mbuf *)((long)(x) & ~(MSIZE-1)))
```

[sys/mbuf.h]

There is a problem when the data is stored in an external page as in an mbuf cluster. In that case knowing the address of the IP header would not be sufficient to retrieve the mbuf header using the dtom function (this function because of this is deprecated).

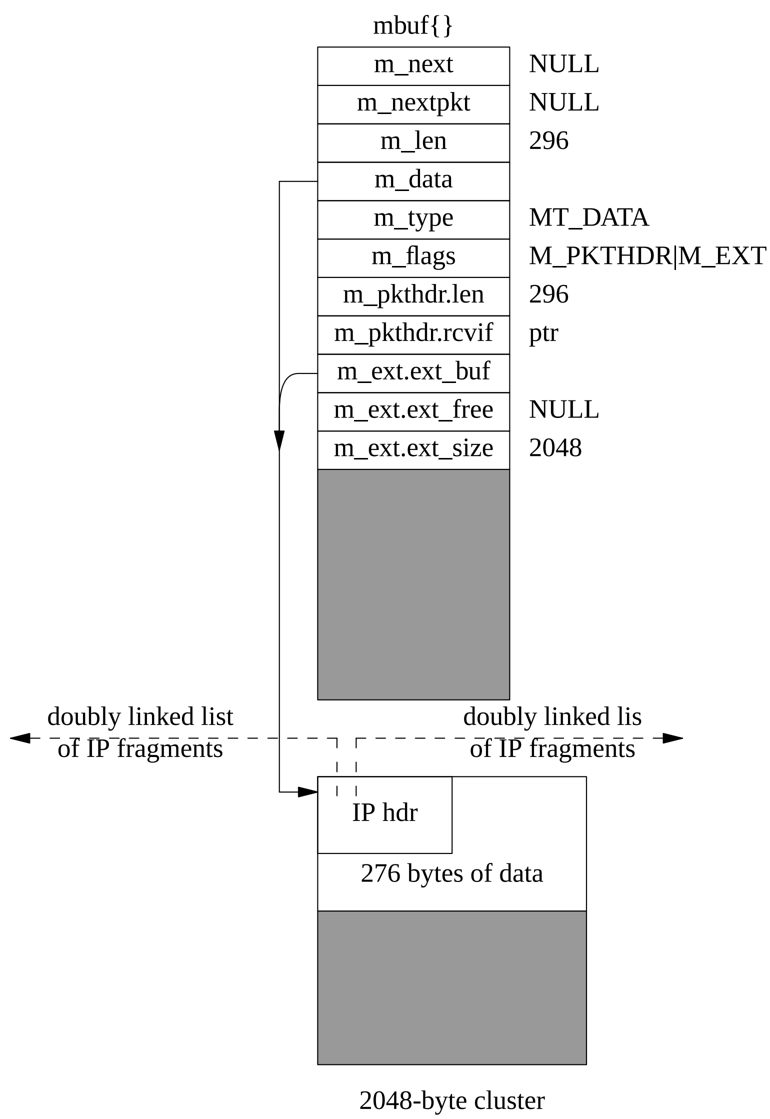


Figure 8 mbuf_five.pic

For this reason, in such a case the `m_pullup` function is called to eventually copy the IP header in a newly allocated mbuf that will be prepended to the one being processed.

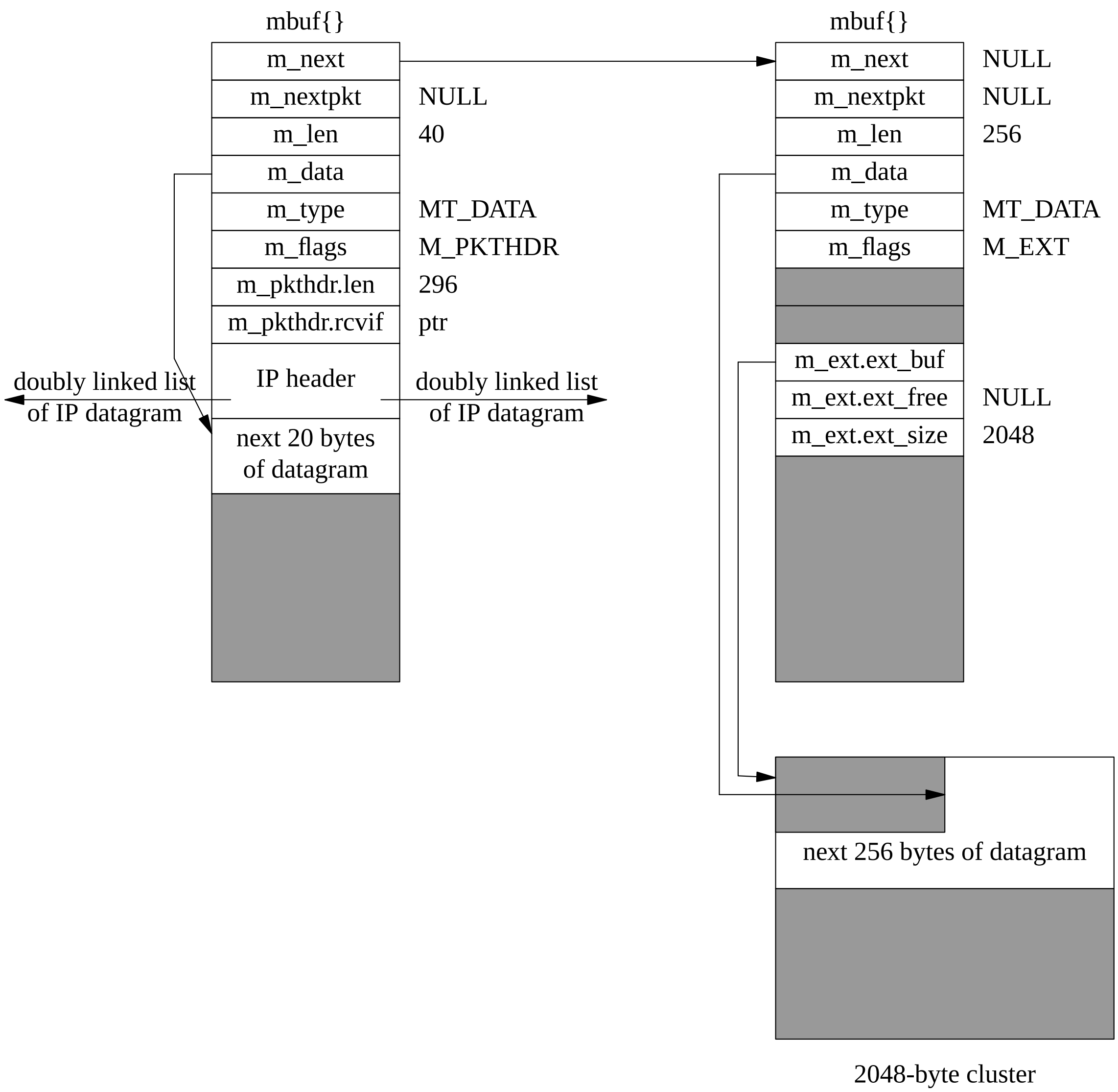


Figure 9 mbuf_six.pic

It is the ip_reass function in the ip_input.c file that takes care of managing IP fragments. An mbuf (or mbuf chain) with an IP fragment is stored on an ip fragment queue after having decreased its size (m_len) by the amount of the IP header length and adjusted its `m_data` pointer to point after the IP header. When ip_reass detects the queue of fragments is complete, then it goes through the links stored in the ip headers of the packets, concatenates the mbufs throught their m_next pointer(this reduces an eventual list of mbuf chains to a simple single mbuf chain), restores the IP source and destination in the IP header of the first mbuf, adjusts the m_data of the first mbuf to expose again the IP header and its m_len to comprise the IP header length. Then the packet len is computed as the sum of the lengths of the fragments and stored in the m_pkthdr.len of the first mbuf. Then it returns the mbuf chain.

[sys/netinet/ip_input.c]

```
void
```

```
ipintr()
{
     register struct ip *ip;
     register struct mbuf *m;
     register struct ipq *fp;
     register struct in_ifaddr *ia;
     int hlen, s;

next:
     /*
      * Get next datagram off input queue and get IP header
      * in first mbuf.
      */
     s = splimp();
     IF_DEQUEUE(&ipintrq, m);
     splx(s);
     if (m == 0)
          return;
#ifdef    DIAGNOSTIC
     if ((m->m_flags & M_PKTHDR) == 0)
          panic("ipintr no HDR");
#endif
     /*
      * If no IP addresses have been set yet but the interfaces
      * are receiving, can't do anything with incoming packets yet.
      */
     if (in_ifaddr == NULL)
          goto bad;
     ipstat.ips_total++;
     if (m->m_len < sizeof (struct ip) &&
         (m = m_pullup(m, sizeof (struct ip))) == 0) {
          ipstat.ips_toosmall++;
          goto next;
     }
     ip = mtod(m, struct ip *);
     if (ip->ip_v != IPVERSION) {
          ipstat.ips_badvers++;
          goto bad;
     }
     hlen = ip->ip_hl << 2;
     if (hlen < sizeof(struct ip)) {/* minimum header length */
          ipstat.ips_badhlen++;
          goto bad;
     }
     if (hlen > m->m_len) {
          if ((m = m_pullup(m, hlen)) == 0) {
               ipstat.ips_badhlen++;
               goto next;
          }
          ip = mtod(m, struct ip *);
     }
     if (ip->ip_sum = in_cksum(m, hlen)) {
          ipstat.ips_badsum++;
          goto bad;
```

```
        }

        /*
         * Convert fields to host representation.
         */
        NTOHS(ip->ip_len);
        if (ip->ip_len < hlen) {
             ipstat.ips_badlen++;
             goto bad;
        }
        NTOHS(ip->ip_id);
        NTOHS(ip->ip_off);

        /*
         * Check that the amount of data in the buffers
         * is as at least much as the IP header would have us expect.
         * Trim mbufs if longer than we expect.
         * Drop packet if shorter than we expect.
         */
        if (m->m_pkthdr.len < ip->ip_len) {
             ipstat.ips_tooshort++;
             goto bad;
        }
        if (m->m_pkthdr.len > ip->ip_len) {
             if (m->m_len == m->m_pkthdr.len) {
                  m->m_len = ip->ip_len;
                  m->m_pkthdr.len = ip->ip_len;
             } else
                  m_adj(m, ip->ip_len - m->m_pkthdr.len);
        }

        /*
         * Process options and, if not destined for us,
         * ship it on.  ip_dooptions returns 1 when an
         * error was detected (causing an icmp message
         * to be sent and the original packet to be freed).
         */
        ip_nhops = 0;        /* for source routed packets */
        if (hlen > sizeof (struct ip) && ip_dooptions(m))
             goto next;

        /*
         * Check our list of addresses, to see if the packet is for us.
         */
        for (ia = in_ifaddr; ia; ia = ia->ia_next) {
#define   satosin(sa)    ((struct sockaddr_in *)(sa))

             if (IA_SIN(ia)->sin_addr.s_addr == ip->ip_dst.s_addr)
                  goto ours;
             if (
#ifdef    DIRECTED_BROADCAST
                 ia->ia_ifp == m->m_pkthdr.rcvif &&
#endif
                 (ia->ia_ifp->if_flags & IFF_BROADCAST)) {
```

```
                        u_long t;

                        if (satosin(&ia->ia_broadaddr)->sin_addr.s_addr ==
                            ip->ip_dst.s_addr)
                                goto ours;
                        if (ip->ip_dst.s_addr == ia->ia_netbroadcast.s_addr)
                                goto ours;
                        /*
                         * Look for all-0's host part (old broadcast addr),
                         * either for subnet or net.
                         */
                        t = ntohl(ip->ip_dst.s_addr);
                        if (t == ia->ia_subnet)
                                goto ours;
                        if (t == ia->ia_net)
                                goto ours;
                }
        }
        if (IN_MULTICAST(ntohl(ip->ip_dst.s_addr))) {
                struct in_multi *inm;
#ifdef MROUTING
                extern struct socket *ip_mrouter;

                if (ip_mrouter) {
                        /*
                         * If we are acting as a multicast router, all
                         * incoming multicast packets are passed to the
                         * kernel-level multicast forwarding function.
                         * The packet is returned (relatively) intact; if
                         * ip_mforward() returns a non-zero value, the packet
                         * must be discarded, else it may be accepted below.
                         *
                         * (The IP ident field is put in the same byte order
                         * as expected when ip_mforward() is called from
                         * ip_output().)
                         */
                        ip->ip_id = htons(ip->ip_id);
                        if (ip_mforward(m, m->m_pkthdr.rcvif) != 0) {
                                ipstat.ips_cantforward++;
                                m_freem(m);
                                goto next;
                        }
                        ip->ip_id = ntohs(ip->ip_id);

                        /*
                         * The process-level routing demon needs to receive
                         * all multicast IGMP packets, whether or not this
                         * host belongs to their destination groups.
                         */
                        if (ip->ip_p == IPPROTO_IGMP)
                                goto ours;
                        ipstat.ips_forward++;
                }
#endif
```

```
		/*
		 * See if we belong to the destination multicast group on the
		 * arrival interface.
		 */
		IN_LOOKUP_MULTI(ip->ip_dst, m->m_pkthdr.rcvif, inm);
		if (inm == NULL) {
			ipstat.ips_cantforward++;
			m_freem(m);
			goto next;
		}
		goto ours;
	}
	if (ip->ip_dst.s_addr == (u_long)INADDR_BROADCAST)
		goto ours;
	if (ip->ip_dst.s_addr == INADDR_ANY)
		goto ours;

	/*
	 * Not for us; forward if possible and desirable.
	 */
	if (ipforwarding == 0) {
		ipstat.ips_cantforward++;
		m_freem(m);
	} else
		ip_forward(m, 0);
	goto next;

ours:
	/*
	 * If offset or IP_MF are set, must reassemble.
	 * Otherwise, nothing need be done.
	 * (We could look in the reassembly queue to see
	 * if the packet was previously fragmented,
	 * but it's not worth the time; just let them time out.)
	 */
	if (ip->ip_off &~ IP_DF) {
		if (m->m_flags & M_EXT) {	/* XXX */
			if ((m = m_pullup(m, sizeof (struct ip))) == 0) {
				ipstat.ips_toosmall++;
				goto next;
			}
			ip = mtod(m, struct ip *);
		}
		/*
		 * Look for queue of fragments
		 * of this datagram.
		 */
		for (fp = ipq.next; fp != &ipq; fp = fp->next)
			if (ip->ip_id == fp->ipq_id &&
			    ip->ip_src.s_addr == fp->ipq_src.s_addr &&
			    ip->ip_dst.s_addr == fp->ipq_dst.s_addr &&
			    ip->ip_p == fp->ipq_p)
				goto found;
		fp = 0;
```

```
found:

        /*
         * Adjust ip_len to not reflect header,
         * set ip_mff if more fragments are expected,
         * convert offset of this to bytes.
         */
        ip->ip_len -= hlen;
        ((struct ipasfrag *)ip)->ipf_mff &= ~1;
        if (ip->ip_off & IP_MF)
            ((struct ipasfrag *)ip)->ipf_mff |= 1;
        ip->ip_off <<= 3;

        /*
         * If datagram marked as having more fragments
         * or if this is not the first fragment,
         * attempt reassembly; if it succeeds, proceed.
         */
        if (((struct ipasfrag *)ip)->ipf_mff & 1 || ip->ip_off) {
            ipstat.ips_fragments++;
            ip = ip_reass((struct ipasfrag *)ip, fp);
            if (ip == 0)
                goto next;
            ipstat.ips_reassembled++;
            m = dtom(ip);
        } else
            if (fp)
                ip_freef(fp);
    } else
        ip->ip_len -= hlen;

    /*
     * Switch out to protocol's input routine.
     */
    ipstat.ips_delivered++;
    (*inetsw[ip_protox[ip->ip_p]].pr_input)(m, hlen);
    goto next;
bad:
    m_freem(m);
    goto next;
}
```

```
struct ip *
ip_reass(ip, fp)
    register struct ipasfrag *ip;
    register struct ipq *fp;
{
    register struct mbuf *m = dtom(ip);
    register struct ipasfrag *q;
    struct mbuf *t;
    int hlen = ip->ip_hl << 2;
    int i, next;

    /*
```

```
         * Presence of header sizes in mbufs
         * would confuse code below.
         */
        m->m_data += hlen;
        m->m_len -= hlen;

        /*
         * If first fragment to arrive, create a reassembly queue.
         */
        if (fp == 0) {
                if ((t = m_get(M_DONTWAIT, MT_FTABLE)) == NULL)
                        goto dropfrag;
                fp = mtod(t, struct ipq *);
                insque(fp, &ipq);
                fp->ipq_ttl = IPFRAGTTL;
                fp->ipq_p = ip->ip_p;
                fp->ipq_id = ip->ip_id;
                fp->ipq_next = fp->ipq_prev = (struct ipasfrag *)fp;
                fp->ipq_src = ((struct ip *)ip)->ip_src;
                fp->ipq_dst = ((struct ip *)ip)->ip_dst;
                q = (struct ipasfrag *)fp;
                goto insert;
        }

        /*
         * Find a segment which begins after this one does.
         */
        for (q = fp->ipq_next; q != (struct ipasfrag *)fp; q = q->ipf_next)
                if (q->ip_off > ip->ip_off)
                        break;

        /*
         * If there is a preceding segment, it may provide some of
         * our data already.  If so, drop the data from the incoming
         * segment.  If it provides all of our data, drop us.
         */
        if (q->ipf_prev != (struct ipasfrag *)fp) {
                i = q->ipf_prev->ip_off + q->ipf_prev->ip_len - ip->ip_off;
                if (i > 0) {
                        if (i >= ip->ip_len)
                                goto dropfrag;
                        m_adj(dtom(ip), i);
                        ip->ip_off += i;
                        ip->ip_len -= i;
                }
        }

        /*
         * While we overlap succeeding segments trim them or,
         * if they are completely covered, dequeue them.
         */
        while (q != (struct ipasfrag *)fp && ip->ip_off + ip->ip_len > q->ip_off) {
                i = (ip->ip_off + ip->ip_len) - q->ip_off;
```

```
			if (i < q->ip_len) {
				q->ip_len -= i;
				q->ip_off += i;
				m_adj(dtom(q), i);
				break;
			}
			q = q->ipf_next;
			m_freem(dtom(q->ipf_prev));
			ip_deq(q->ipf_prev);
		}

	insert:
		/*
		 * Stick new segment in its place;
		 * check for complete reassembly.
		 */
		ip_enq(ip, q->ipf_prev);
		next = 0;
		for (q = fp->ipq_next; q != (struct ipasfrag *)fp; q = q->ipf_next) {
			if (q->ip_off != next)
				return (0);
			next += q->ip_len;
		}
		if (q->ipf_prev->ipf_mff & 1)
			return (0);

		/*
		 * Reassembly is complete; concatenate fragments.
		 */
		q = fp->ipq_next;
		m = dtom(q);
		t = m->m_next;
		m->m_next = 0;
		m_cat(m, t);
		q = q->ipf_next;
		while (q != (struct ipasfrag *)fp) {
			t = dtom(q);
			q = q->ipf_next;
			m_cat(m, t);
		}

		/*
		 * Create header for new ip packet by
		 * modifying header of first packet;
		 * dequeue and discard fragment reassembly header.
		 * Make header visible.
		 */
		ip = fp->ipq_next;
		ip->ip_len = next;
		ip->ipf_mff &= ~1;
		((struct ip *)ip)->ip_src = fp->ipq_src;
		((struct ip *)ip)->ip_dst = fp->ipq_dst;
		remque(fp);
		(void) m_free(dtom(fp));
```

```
        m = dtom(ip);
        m->m_len += (ip->ip_hl << 2);
        m->m_data -= (ip->ip_hl << 2);
        /* some debugging cruft by sklower, below, will go away soon */
        if (m->m_flags & M_PKTHDR) { /* XXX this should be done elsewhere */
             register int plen = 0;
             for (t = m; m; m = m->m_next)
                  plen += m->m_len;
             t->m_pkthdr.len = plen;
        }
        return ((struct ip *)ip);

   dropfrag:
        ipstat.ips_fragdropped++;
        m_freem(m);
        return (0);
   }
```

[sys/netinet/ip_input.c]

5. Unix SVR4 STREAMS

STREAMS is a general programming model to develop communication services on Unix. It defines the programming interface between the various components of the programming model: a stream head, zero or more modules and a device driver. In this model a user process interacts with a STREAMS queue head, this one can interact directly or through a stack of modules with a device driver.

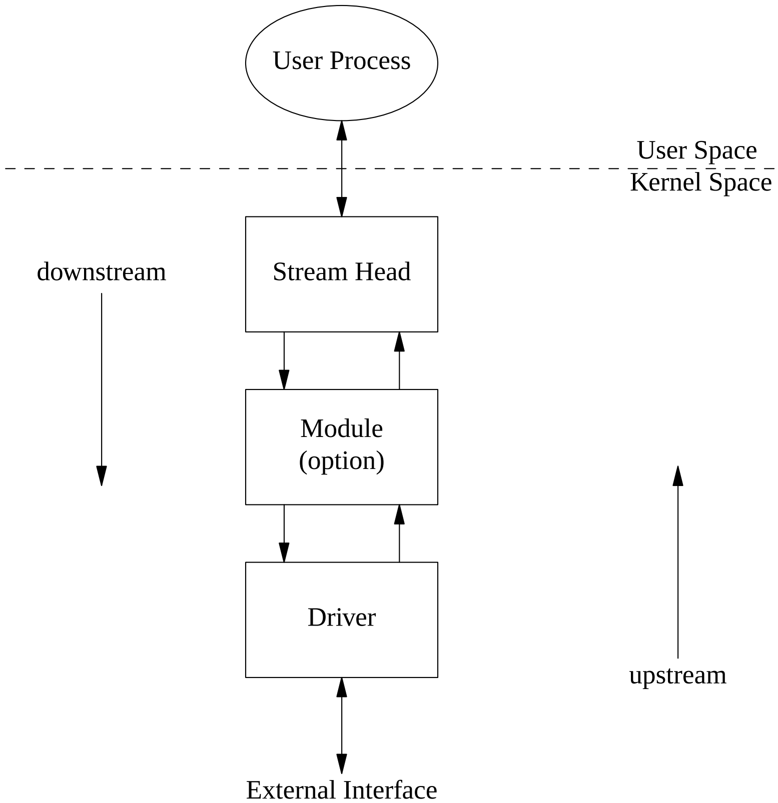


Figure 10 stream.pic

All the communications between STREAMS components are done through `messages.` A `message` in STREAMS is described by one or more message blocks (msgb) linked through the `b_cont` pointer. Messages are kept on queues in a doubly linked list through the `b_next` , `b_prev` pointers of the first message block(msgb) of the message.

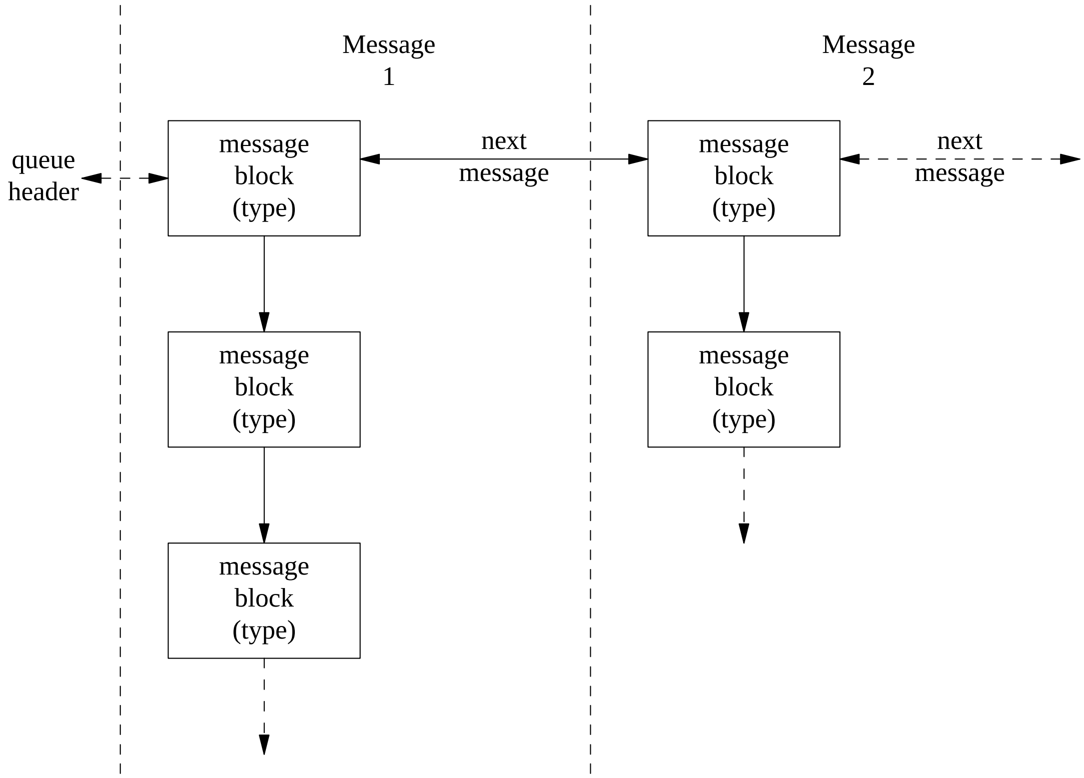


Figure 11 stream_two.pic

```
struct msgb {
  struct msgb    *b_next;         /*next msg on queue*/
  struct msgb    *b_prev;         /*previous msg on queue*/
  struct msgb    *b_cont;         /*next msg block of message*/
  unsigned char  *b_rptr;         /*1st unread byte in bufr*/
  unsigned char  *b_wptr;         /*1st unwritten byte in bufr*/
  struct datab   *b_datap;        /*data block*/
  unsigned char  b_band;          /*message priority*/
  unsigned short b_flag;          /*see below - Message flags*/
};
typedef struct msgb mblk_t;
```

The data, kept in a `data buffer` is described by the data block(`datab`) that is accessed through the `b_datap` pointer in the message block.

```
struct datab {
   unsigned char   *db_base;      /* first byte of buffer */
   unsigned char   *db_lim;       /* last byte+1 of buffer */
   unsigned char   db_ref;        /* msg count ptg to this blk */
   unsigned char   db_type;       /* msg type */
};
typedef struct datab dblk_t;
```

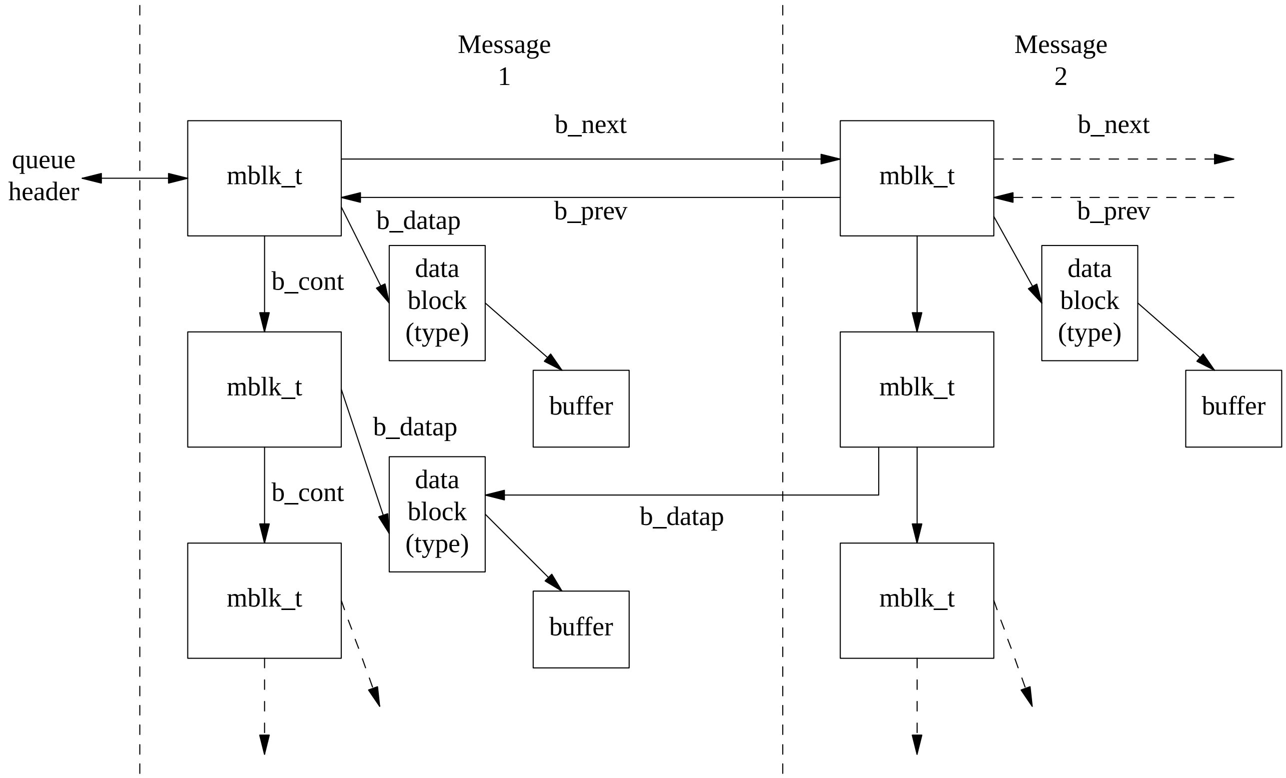


Figure 12 stream_one.pic

A mechanism to avoid unnecessary copies is provided through the sharing of the data block, where a referenc counter `db_ref` is kept and incremented for each new entity that references the same data block. This is called duplication on STREAMS documentation, and is performed through the use of the `dupmsg` or `dupb` function.

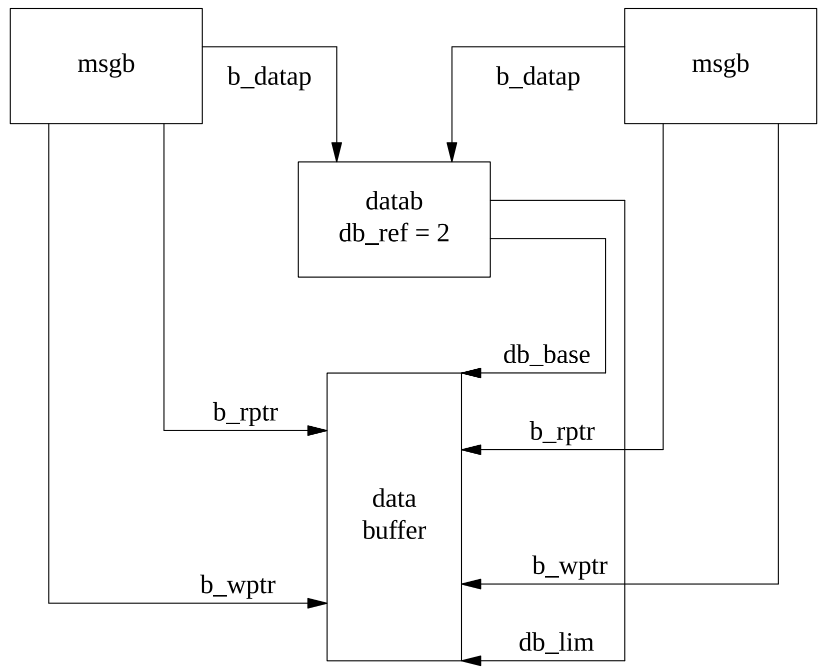


Figure 13 A shared data block

With Unix STREAMS a module can create a new message block that shares the data block and data buffer of an existing one. The read and write pointers (b_rptr, b_wptr) are kept inside the message block and so two entities can share the data block and the associated data buffer but have a different viewport on the data. For instance a tcp module and an ip module can share a data block and buffer even if the tcp module will not consider the IP header.

5.1. STREAMS functions

Message blocks (mblk) are allocated through the :

```
mblk_t *allocb(size_t size,uint_t pri);
```

where size is the number of data bytes available in the message block, and pri is the priority of the request. This call allocates all the necessary structures : the msgb, the datab and a conveniently sized data buffer and it appropriately initializes all the fields. This example sends an error code of 1 byte on a stream :

```
send_error(q,err)
  queue_t *q;
  unsigned char err;
{
  mblk_t *bp;

  if ((bp = allocb(1, BPRI_HI)) == NULL) /* allocate msg. block */
       return(0);

   bp->b_datap->db_type = M_ERROR;    /* set msg type to M_ERROR */
   *bp->b_wptr++ = err;               /* increment write pointer */

   if (!(q->q_flag & QREADR))     /* if not read queue     */
        q = RD(q);                    /*    get read queue     */
   putnext(q,bp);                     /* send message upstream */
   return(1);
 }
```

A message block can be allocated for an existing data buffer. In this case the esballoc (extended streams buffer allocation) function is used :

```
mblk_t  *esballoc(uchar  *base,  size_t  size,  uint_t  pri,frtn_t *fr_rtnp);
```

that allocates only the msgb and datab, and sets up the fields to point to the user supplied `base` buffer (it is used for instance with devices having dual-ported memory to avoid copying the data to a kernel buffer). The caller can in this case specify a message specific free function that will be called to deallocate the data buffer when it will be released. The allocation of memory for a message block in STREAMS can be cumbersome. Three different areas need to be allocated : a msgb, a datab and a data buffer. In SVR4 memory allocation is done in a smart way through the use of 128 bytes mdbblocks.

```
struct mdbblock {
    mblk_t m_mblock;
    void   (*m_func)(void);
    dblk_t d_dblock;
    char   databuf[FASTBUF];
}
```

These areas of memory can keep a msgb and a datab, and eventually the remaining space can be used for a small data buffer. Therefore the allocation of a msgb for a small buffer size reduces to only one memory allocation.

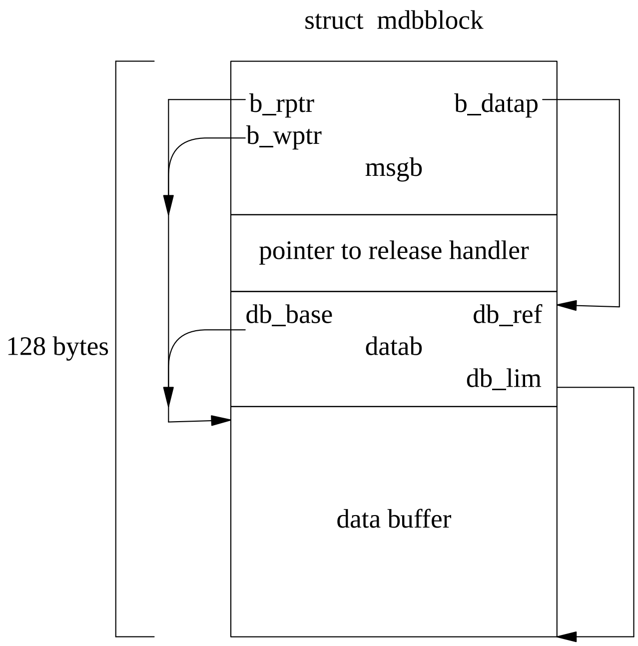


Figure 14 An mddblock structure just after allocation

In case the size requested is more than the free space available in an mdbblock then an external buffer of proper size is allocate with the standard kmem_alloc() function. LiS (Linux STREAMS) is an implementation of STREAMS for Linux that follows the SVR4 memory allocation design. In LiS the size of the mdbblock internal data buffer(FASTBUF) on Intel x86 is 78 bytes.
Message blocks can be linked through their b_cont field to form a chain using the function :

```
void linkb(mblk_t* mp,mblk_t* bp);
```

This function concatenates the 2 messages putting the bp message at the tail of the mp message.
Message blocks can be copied with copyb :

```
struct msgb *bp = copyb(struct msgb *mp);
```

this will allocate a new msgb and datab and data buffer and will copy over the data between the rptr and wptr in the old msgb. If it is successfull it returns a pointer to the new msgb otherwise it returns NULL. This function will not follow the b_cont pointer, and will copy a single msgb. The copymsg function instead copies all the msgb in a message following the b_cont links :

```
mblk_t* copymsg(mblk_t* bp);
```

Example 1: : Using copyb

For each message in the list, test to see if the downstream queue is full with the canputnext(9F) function (line 21). If it is not full, use copyb to copy a header message block, and dupmsg(9F) to duplicate the data to be retransmitted. If either operation fails, reschedule a timeout at the next valid interval.

Update the new header block with the correct destination address (line 34), link the message to it (line 35), and send it downstream (line 36). At the end of the list, reschedule this routine.

```
 1  struct retrans {
 2        mblk_t            *r_mp;
 3        int               r_address;
 4        queue_t           *r_outq;
 5        struct retrans    *r_next;
 6  };
 7
 8  struct protoheader {
       ...
 9     int                  h_address;
       ...
10  };
11
12  mblk_t *header;
13
14  void
15  retransmit(struct retrans *ret)
16  {
17       mblk_t *bp, *mp;
18       struct protoheader *php;
19
20       while (ret) {
21         if (!canputnext(ret->r_outq)) {   /* no room */
22                ret = ret->r_next;
23                continue;
24         }
25         bp = copyb(header);               /* copy header msg. block */
26         if (bp == NULL)
27               break;
28         mp = dupmsg(ret->r_mp);           /* duplicate data */
29         if (mp == NULL) {                 /* if unsuccessful */
30              freeb(bp);                   /* free the block */
31              break;
32         }
33         php = (struct protoheader *)bp->b_rptr;
34         php->h_address = ret->r_address;   /* new header */
35         bp->bp_cont = mp;                  /* link the message */
36         putnext(ret->r_outq, bp);          /* send downstream */
37         ret = ret->r_next;
38       }
```

```
        /* reschedule */
        (void) timeout(retransmit, (caddr_t)ret, RETRANS_TIME);
}
```

If its not necessary to modify the data in the buffer, the dupb function can be called to allocate a new msgb that points to the same datab, and so shares the data buffer :

```
mblk_t* dupb(mblk_t bp);
mblk_t *dupmsg(mblk_t *mp);
```

The dupb function doesn't follow the b_cont link and does its job only on one msgb. The dupmsg follows the b_cont link and so duplicates all the msgb of a chain. Message blocks can be freed with freeb :

```
void  freeb(mblk_t *bp);
void freemsg(mblk_t* bp);
```

The `freeb` function decrements the reference counter of the datab, if it becomes zero it really gives back the memory for the data buffer and the datab structure, then in any case deallocates the msgb area. The freeb function doesn't follow the b_cont link, while the freemsg function can be used to deallocate all the parts of a message and this is done following the b_cont link.
A message can be trimmed at the head or the tail using the adjmsg() function :

```
int adjmsg(mblk_t* mp, int len)
```

where if len is positive len bytes are trimmed from the beginning and if negative len bytes are trimmed from the tail (this can be used by the protocol layers to strip headers or trailers). The general functions to get or put a message on a queue are :

```
int putq(queue_t q,mblk_t* bp);
mblk_t*  getq(queue_t q);
int insq(queue_t* q,mblk_t* emp,mblk_t* mp);
```

putq puts a message on the specified queue based on its priority (db_type/b_band), getq gets the next available message and insq inserts the mp message immediately before emp in the queue q.
The memory allocation can't sleep waiting for resources, so it can return with a NULL, meaning that there is no memory available. In this case it is possible to register a callback function that will be invoked as soon as a buffer of the requested size will be available.

```
int bufcall(uint size,int pri, void (*func)(), long arg);
```

The STREAMS framework will invoke (*func)() at the time it decides memory is available again.
Each STREAMS stream head, module or driver has to define a `put` procedure (pointed to by the qi_putp pointer in the qinit structure):

```
int put(queue_t* q, mblk_t* mp);
```

for each of the 2 queues that process incoming messages and eventually forward them. All stream heads and eventually modules or drivers can also define a `service` procedure (pointed to by the qi_srvp pointer in the qinit structure) :

```
int service(queue* q);
```

to process all deferred messages, when the STREAMS scheduler will detect that a blocking condition is resolved and will invoke it.
In the STREAMS framework TCP/IP has been provided through a message based interface called TPI(Transport Provider Interface) and through the procedural TLI (Transport Layer Interface) APIs.

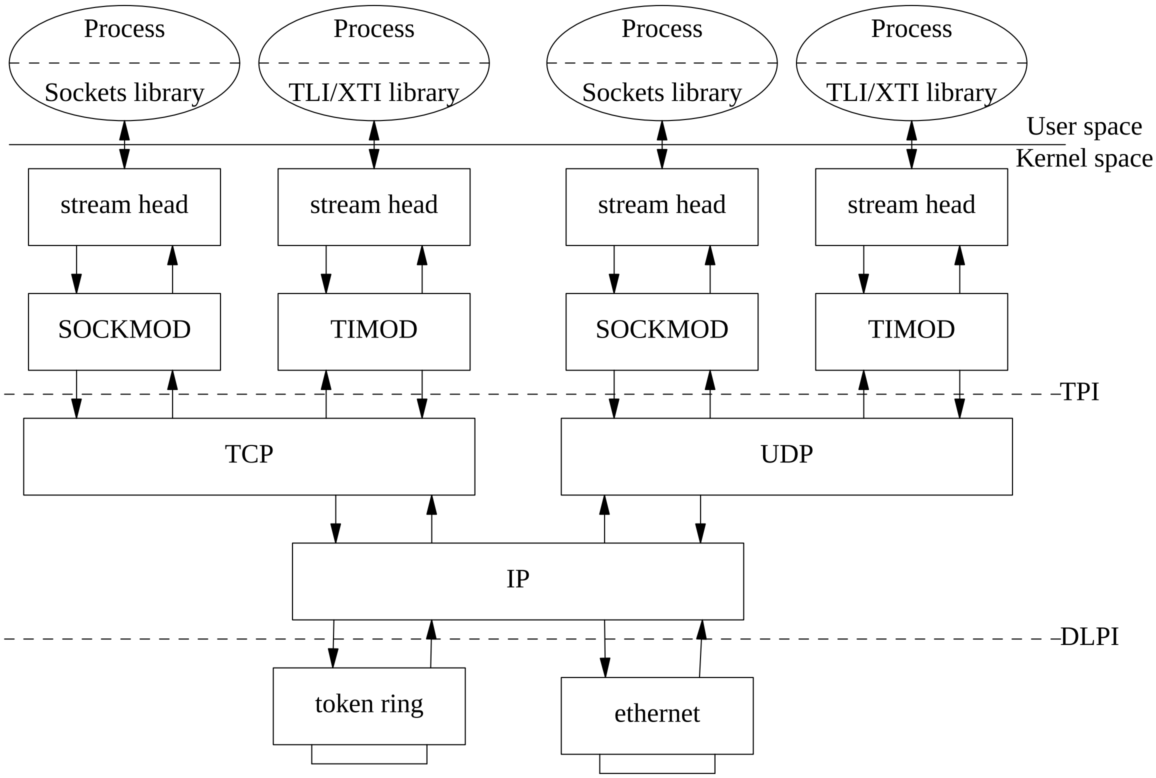


Figure 15 stream_seven.pic

As the socket interface came long before TPI/TLI and was widely diffused it has been provided on top of the STREAMS TCP, UDP and IP modules through a STREAMS sockmod module and a library.

STREAMS entities that can queue messages have a service procedure defined. There are two functions that can check if there is room in a queue. If the queue doesn't have a service function then they will search for one along the stream in the same direction :

```
int bcanput(queue_t* q,unsigned char pri);
int canput(queue_t* q);
```

bcanput checks if there is room for one message at priority pri, canput is equivalent to bcanput(q,0).
An operation that is frequently performed is that of sending a message to the next module (or driver or stream head) in a stream. This is done invoking the putp function registered by that module in the qinit structure. This operation can be conveniently boiled down in a putnext macro :

```
#define putnext(q, mp) ((*(q)->q_next->q_qinfo->qi_putp)((q)->next,(mp)))
```

Many operations on headers and trailers are done through some pointer arithmetic (for instance the source address field is at a certain offset from the network header). These operations on pointers can be done only if that area of memory is contiguous. Therefore a special utility function is used to copy in a contiguous data buffer a specified number of bytes from a chain of message blocks.

```
int pullupmsg(struct msgb* mp,int len);
```

This function returns a chain of msgb in which the first msgb contains len bytes of data in its contiguous data buffer. If len is -1 then the utility concatenates all the blocks of the same type at the beginning of the message. In the LiS implementation the first msgb is reused while a new datab and data buffer are allocated for the first block.
STREAMS queues are always allocated in pairs, one is the read side and the other the write side, the first

passes messages downstream (towards the driver), the second passes messages upstream (toward the stream head). Inside the queue structure there is a pointer q_other that connects each other, and makes it simple to get the read or write side or the other side of the queue pair.

```
taken from the LiS implementation for Linux :
typedef
struct queue
{
  EXPORT
        struct  qinit   *q_qinfo;       /* procs and limits for queue [I]*/
        struct  msgb    *q_first;       /* first data block [Z]*/
        struct  msgb    *q_last;        /* last data block [Z]*/
        struct  queue   *q_next;        /* next Q of stream [Z]*/
        struct  queue   *q_link;        /* to next Q for the scan list [Z]*/
        void            *q_ptr;         /* module private data for free */
        ulong           q_count;        /* number of bytes on Q [Z]*/
        volatile ulong  q_flag;         /* queue state [Z]*/
  SHARE
        long            q_minpsz;       /* min packet size accepted [I]*/
        long            q_maxpsz;       /* max packet size accepted [I]*/
        ulong           q_hiwat;        /* queue high water mark [I]*/
        ulong           q_lowat;        /* queue low water mark [I]*/
  PRIVATE
        struct qband    *q_bandp;       /* separate flow information */
        struct queue    *q_other;       /* for RD()/WR()/OTHER() */
        void            *q_str;         /* pointer to stream's stdata */
        struct  queue   *q_scnxt;       /* next q in the scan list */
        ulong            q_magic;       /* magic number */
        lis_semaphore_t  q_lock;        /* for put, srv, open, close */
        lis_atomic_t     q_lock_nest;   /* for nested entries */
        void            *q_taskp;       /* owner of the q_lock */
        lis_spin_lock_t  q_isr_lock;    /* for ISR protection */
        lis_semaphore_t *q_wakeup_sem ; /* helps sync closes */
} queue_t;
```

Special calls are defined to perform these operations :

```
queue_t RD(queue_t* q);
queue_t WR(queue_t* q);
queue_t OTHERQ(queue_t* q);
```

RD returns a pointer to the read side, WR returns a pointer to the write side and OTHERQ returns a pointer to the other side of the queue pair whatever it is. Read or write side is easily recognized because of the QREADR flag set or unset in the q_flag field of the queue structure respectively. The qinit structure pointed to by the q_qinfo pointer in the queue structure, is a table of pointers to functions defined to be used with the queue and a structure with info about the module and another with some statistical data. The only required procedure is the put procedure, if the module implements queueing, then at least the service procedure should also be defined.

```
typedef
struct  qinit {
#if defined(USE_VOID_PUT_PROC)
        void    (*qi_putp)(queue_t*, mblk_t*);  /* put procedure */
        void    (*qi_srvp)(queue_t*);           /* service procedure */
#else
        int     (*qi_putp)(queue_t*, mblk_t*);  /* put procedure */
        int     (*qi_srvp)(queue_t*);           /* service procedure */
#endif
        int     (*qi_qopen)(queue_t *, dev_t *, int, int, cred_t *);   /* open procedure */
        int     (*qi_qclose)(queue_t *, int,cred_t *);   /* close procedure */
        int     (*qi_qadmin)(void);         /* debugging */
        struct lis_module_info *qi_minfo;   /* module information structure */
        struct module_stat *qi_mstat;   /* module statistics structure */
} qinit_t;
```

Each module should define 2 such qinit structures, one for the read side and one for the write side, pointers to them are kept inside a table called streamtab :

```
typedef struct streamtab {
  SHARE
        struct qinit *st_rdinit; /* read queue */
        struct qinit *st_wrinit; /* write queue */
        struct qinit *st_muxrinit; /* mux read queue */
        struct qinit *st_muxwinit; /* mux write queue */
} streamtab_t;
```

The user defines the read and write qinit structures, and creates the streamtab structure that links the 2 together.

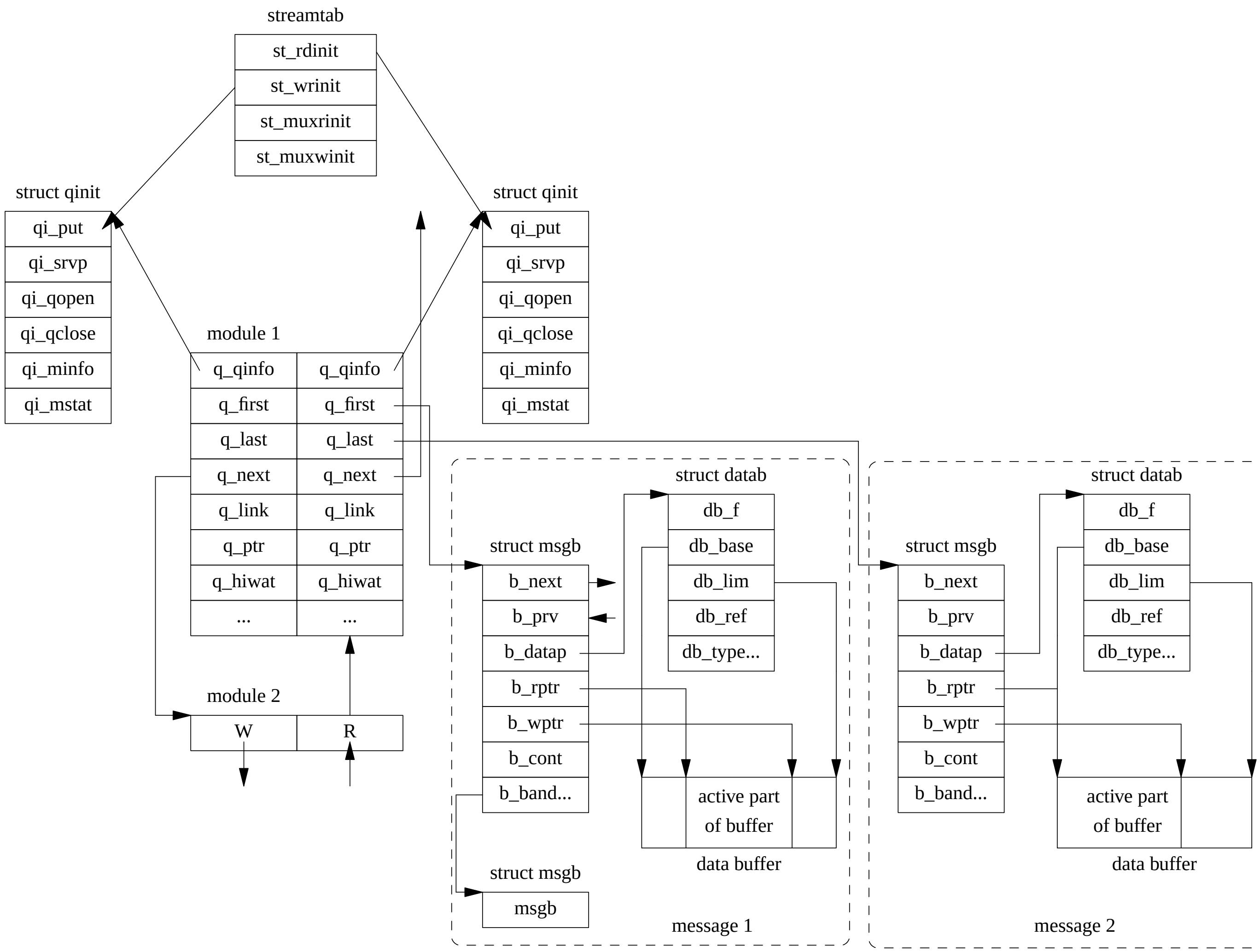


Figure 16 streamtab structure linking the read and write side of a queue

A module is pushed on the top of a stream with the I_PUSH ioctl call or calling stropen. The STREAMS then calls qattach with the streamtab as an argument. qattach allocates a pair of queues with allocq, links the queues onto the stream and calls the module's qopen routine.

The standard read and write calls can't differentiate between message types and priorities, they simply read and write unqualified streams of bytes. Therefore the putmsg,getmsg,putpmsg,getpmsg functions are added (the putpmsg and getpmsg functions simply have one additional argument that specifies a priority for the message). The put functions accept a control and a data buffer, and will build a message with a msgb of type M_PROTO or M_DATA or one with a msgb of type M_PROTO and one of type M_DATA in a chain. The get functions will perform the reverse.

```
int putmsg(int fd, void *ctlptr, void *dataptr, int flags)
int getmsg(int fd, void *ctlptr, void *dataptr, int *flagsp)
int putpmsg(int fd, void *ctlptr, void *dataptr, int band, int flags) ;
int getpmsg(int fd, void *ctlptr, void *dataptr, int *bandp, int *flagsp) ;
```

5.2. Avoiding unecessary copies sharing or cloning network buffers

Sometimes it is necessary that a protocol layer or a module, a device driver needs to keep a copy of a network buffer. For instance a reliable protocol can need to keep a copy for retransmission (TCP). Or a packet capture driver can need it to pass it to another reader. In many cases who requires the copy doesn't need to alter it. A mechanism of lazy copy or COW or sharing of network buffers therefore could provide great benefits. Linux has two different mechanisms to avoid unnecessary copies of network buffers : one in which everything is shared (the header and the data, this is obtained calling skb_get for instance, and denoted by the skb->users field in the header being different from 1), and the other in which only the data is shared (denoted by incrementing the shared_info field dataref), also called cloning, where two different headers, point to the same data part. It is important to stress that in the Linux cloning the pointers inside the headers can be modified without the necessity to copy the data (adding/stripping headers).

6. Linux skbuffs

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 version 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.

6.1. Linux memory management

There are different levels of memory management on Linux. There is a base level that allocates contiguous pages of memory, this is the physical page allocator, and an upper allocator for smaller memory areas that is organized to take advantage of object reuse.

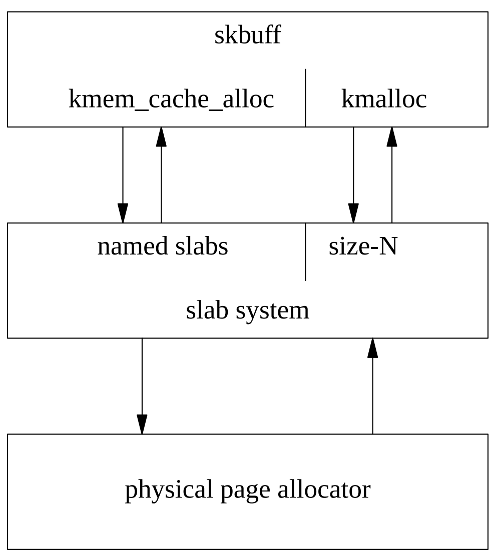


Figure 17 skb_mem.pic

6.1.1. Physical page allocation

The physical page allocator in Linux is patterned after the binary buddy system [1] This allocator keeps pages in a power of 2 list, this power of 2 is called order and in the current implementation the maximum of it (MAX_ORDER) is 10 and so there are lists for chunks of physical pages from 2^0=1(4 KB) to 2^9=512(2

MB). If a block of the desired size is not available a higher order block is split and the 2 halves become buddies. When eventually running through the list you find a buddy free, you look if the other is free too and eventually you join them and link the resulting higher order block to the higher order list. This system has a coarse granularity and it is not suitable for allocation of small memory areas, also because this kind of allocation can be expensive in terms of the required time. At the very end all memory allocation is anyway based on this method.

6.1.2. Small areas allocation

This allocator is essentially based on the slab allocator proposed by Bonwick in 1994 [Bonwick 94], and first implemented on Sun Solaris. It is based on the concept of object based allocation. This system relies on the physical page allocator. Named slabs are created and given a set of contiguous pages for the allocation of specific structures, so that the time required for their complete initialization is avoided. Anyway this same system is used also for the allocation of generic small memory areas, this is done providing generic slabs in power of 2 sizes.

6.1.2.1. Named slabs

These are used for the object based allocation. At their creation a constructor and destructor function are specified. For instance tcp_open_request, tcp_bind_bucket, tcp_tw_bucket, sock, skbuff_head_cache are named slabs.
Named slabs are created with the function:

```
kmem_cache_t* kmem_cache_create(const char* name,size_t size,size_t offset,
           unsigned long flags,
           void (*ctor) (void*,kmem_cache_t*,unsigned long),
           void (*dtor) (void*,kmem_cache_t*,unsigned long))
```

Then the allocation of an object from a slab is done with :

```
void* kmem_cache_alloc(kmem_cache_t* cachep, int flags )
```

and its deallocation with :

```
void kmem_cache_free(kmem_cache_t* cachep, void* objp)
```

6.1.2.2. Size-n generic slabs

Two power of 2 lists of generic slabs ranging in size from 2^5=32 bytes to 2^{17}=128KB of memory chunks are kept. One is suitable for DMA (in x86 architecture this memory should be below 16 MB) and one is general. They are named size-32(DMA), ..., size-131072(DMA) and size-32, ..., size-131072 respectively.
Allocation from these generic slabs happens when the generic kernel allocator

```
void * kmalloc(size_t size,int flags)
```

function is invoked.
Info about slabs can be obtained from /proc/slabinfo where the columns are : slab name, active objects, number of objects, object size.

6.1.3. Skbuff use of Linux memory management

The skbuff system uses in two ways the slab allocator.[2] The skbuff heads are allocated from a named slab called `skbuff_head_cache,` created when the `skb_init` function is called at network initialization time.

[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]

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

[net/core/skbuff.c]

```
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;

     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;

     atomic_set(&(skb_shinfo(skb)->dataref), 1);
     skb_shinfo(skb)->nr_frags  = 0;
     skb_shinfo(skb)->tso_size = 0;
     skb_shinfo(skb)->tso_segs = 0;
     skb_shinfo(skb)->frag_list = NULL;
out:
     return skb;
nodata:
     kmem_cache_free(skbuff_head_cache, skb);
     skb = NULL;
     goto out;
}
```

[net/core/skbuff.c]

7. Fundamental data structures

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

7.1. 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. Its size is 160 bytes. 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]

```
static __inline__ void skb_bond(struct sk_buff *skb)
{
     struct net_device *dev = skb->dev;

     if (dev->master) {
          skb->real_dev = skb->dev;
          skb->dev = dev->master;
     }
}
```

[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 buffer (`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 `len - data_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]

```
#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]

```
                goto nodata;

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

[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]

```
 *   @nf_bridge: Saved data about a bridged frame - see br_netfilter.c
 *      @private: Data which is private to the HIPPI implementation
 *   @tc_index: Traffic control index
 */

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
#ifdef CONFIG_BRIDGE_NETFILTER
        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,
```

[include/linux/skbuff.h]

7.2. 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. The frag_list pointer is only used to connect the 1st fragment to the second, the other skbuff are linked through the standard next pointer. Skbuffs fragmented in a list are used for the reassembly of ip fragments. 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]

```

struct sk_buff;
```

[include/linux/skbuff.h]

[include/linux/skbuff.h]

```
/* 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 int   nr_frags;
        unsigned short  tso_size;
        unsigned short  tso_segs;
```

[include/linux/skbuff.h]

The `shared_info` structure that we said is placed at the `skb->end` is usually accessed using the macro `skb_shinfo`

[include/linux/skbuff.h]

```
#define skb_shinfo(SKB)        ((struct skb_shared_info *)((SKB)->end))
```

[include/linux/skbuff.h]

for instance :
`skb_shinfo(skb)->frag_list = 0;`

```
skb_shinfo(skb)->dataref
```

8. Skbuff organization

The network buffers in Linux are formed by 2 different entities : a fixed size header, and a variable size data area. Until recently the data area of the skbuff was unique and physically contiguous (aka linear). 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 network packet. In version 2.4 of the stable branch of the kernel fragmented skbuffs were introduced in two forms and for two completely different reasons. To keep the information necessary to describe this more complicate skbuff organizations a small area placed immediately after the skbuff data area (skb->end) was allocated and reserved. Placing this info there could allow the sharing of the information between multiple clones and so the structure that keeps the information is called `skb_shared_info.` The possibility that an skbuff can have data in an array of associated unmapped pages was added to allow the efficient use of interfaces able to perform gather and scatter in hardware. To efficiently manage ip fragments, a chain of skbuffs that could be passed along the network stack like a single skbuff was introduced.

8.1. Linux sharing and cloning

Linux has the most flexible way of providing mechanisms to avoid unnecessary copies. With *sharing* the complete skbuff (headers and data area) is shared between two entities, with *cloning* two different headers point to the same data area.

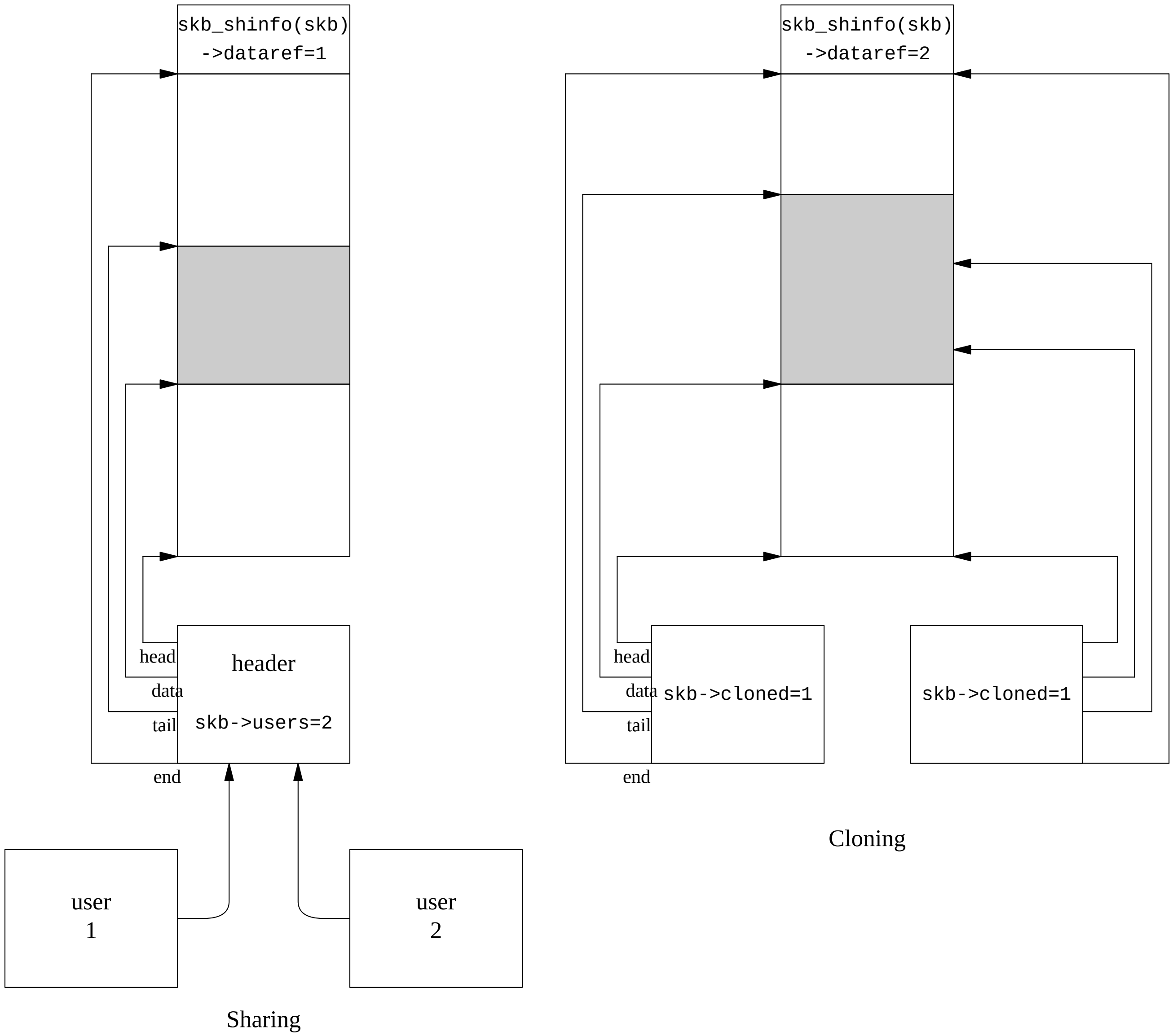


Figure 18 Sharing and cloning skbuffs

Sharing is the least expensive, but doesnt allow any change. Cloning requires copying the header, but then allows the flexibility to modify the header, without copying the data (for instance the pointers skb->data and skb->tail can be changed to strip away headers). Sharing is denoted by atomically incrementing the skb->users counter for each user holding a ref to the skbuff, cloning is denoted by setting the skb->cloned flag in the header and incrementing the skb_shinfo(skb)->dataref field in the shared_info struct.

An skbuff can be shared calling the `skb_get` function over an existing skbuff. This will atomically increment the skb->users counter and return a pointer to the skbuff. For instance :

8.2. Linear skbuffs

This was the original and only way in which skbuffs were used. In this case the variable length data area was formed by a physically contiguous area of memory. Network operations that resulted in adding/stripping headers or trailers or data, were performed changing the pointers to the data area (`skb->data,` skb->tail) inside the header. If for any reason the data area could not be expanded in place to add headers, trailers or data, a new linear skbuff is created and the data is copied over. In the case of a linear skbuff, the only info in the skb_shared_info area that could be used is that related to the tso (TCP Segmentation Offloading) and the counter that indicates how many entities share the data area of the skbuff (clones).

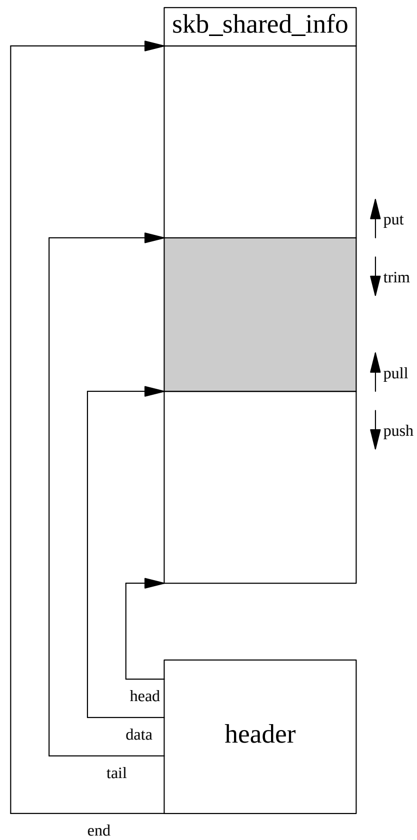


Figure 19 skbuff_three.pic

8.3. Nonlinear skbuffs

For non linear skbuffs the infor

8.3.1. Paged skbuffs

In this case some unmapped pages of memory are associated with the skbuff. The number of these pages is stored in the `nr_frags` field of the skb_shared_info. And an array of page pointers, of size to allow at least 64 KB of data (it was just 6 pages in kernel 2.4) is used. This kind of fragmentation was introduced to make use of the scatter/gather capability of modern interfaces which can be given a vector of buffers to deal with by themselves. An important difference with BSD should be stressed : the Linux code allows in principle to have an skbuff using the linear part, and being fragmented in both the possible ways at the same time. On BSD if an `mbuf` uses an external page, then no data is put in its linear part. A hook to allocate paged skbuff is provided by the function `sock_alloc_send_pskb.` This function would allocated a paged skbuff having at least `data_len` bytes of space in its external pages.

[net/core/sock.c]

```
        finish_wait(sk->sk_sleep, &wait);
        return timeo;
}


/*
 *      Generic send/receive buffer handlers
 */

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;
        unsigned int 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);
        while (1) {
                err = sock_error(sk);
                if (err != 0)
                        goto failure;

                err = -EPIPE;
                if (sk->sk_shutdown & SEND_SHUTDOWN)
                        goto failure;

                if (atomic_read(&sk->sk_wmem_alloc) < sk->sk_sndbuf) {
                        skb = alloc_skb(header_len, sk->sk_allocation);
                        if (skb) {
                                int npages;
                                int i;

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

                                npages = (data_len + (PAGE_SIZE - 1)) >> PAGE_SHIFT;
                                skb->truesize += data_len;
                                skb_shinfo(skb)->nr_frags = npages;
                                for (i = 0; i < npages; i++) {
                                        struct page *page;
                                        skb_frag_t *frag;

                                        page = alloc_pages(sk->sk_allocation, 0);
                                        if (!page) {
                                                err = -ENOBUFS;
```

```
                        skb_shinfo(skb)->nr_frags = i;
                        kfree_skb(skb);
                        goto failure;
                    }

                    frag = &skb_shinfo(skb)->frags[i];
                    frag->page = page;
                    frag->page_offset = 0;
                    frag->size = (data_len >= PAGE_SIZE ?
                                  PAGE_SIZE :
                                  data_len);
                    data_len -= PAGE_SIZE;
                }

                /* Full success... */
                break;
            }
            err = -ENOBUFS;
            goto failure;
        }
        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);
    }

```

[net/core/sock.c]

But in fact this function is never called directly by the current kernel but only through the `sock_alloc_send_skb` function that doesnt request any space in external pages.

[net/core/sock.c]

```
    return skb;

interrupted:
    err = sock_intr_errno(timeo);
failure:
```

[net/core/sock.c]

In the inet_stream_ops table there is a .sendpage method, this is initialized to tcp_sendpage when the table is defined in af_inet.c. Tcp_sendpage if the scatter/gather support is not listed in the route capabilities, will call sock_no_sendpage, otherwise it calls do_tcp_sendpages. This seems to be the only place that gives rise to a paged skbuff. [tcp_alloc_pskb and tcp_alloc_skb only differ because the 1st can add to the truesize of the allocated skbuff the size of the memory that will be allocated in external pages, but it doesnt allocate them at all. The second arg of tcp_alloc_pskb is size=linear part of skb, while the 3d arg is mem=paged part of skb] This paged skbuff is requested to have a paged part of at least tp->mss_cache bytes and a linear part of 0 bytes (the place for TCP headers is automatically added. So the resulting pskb will have the headers in the linear part of the skb and the data in the external pages.

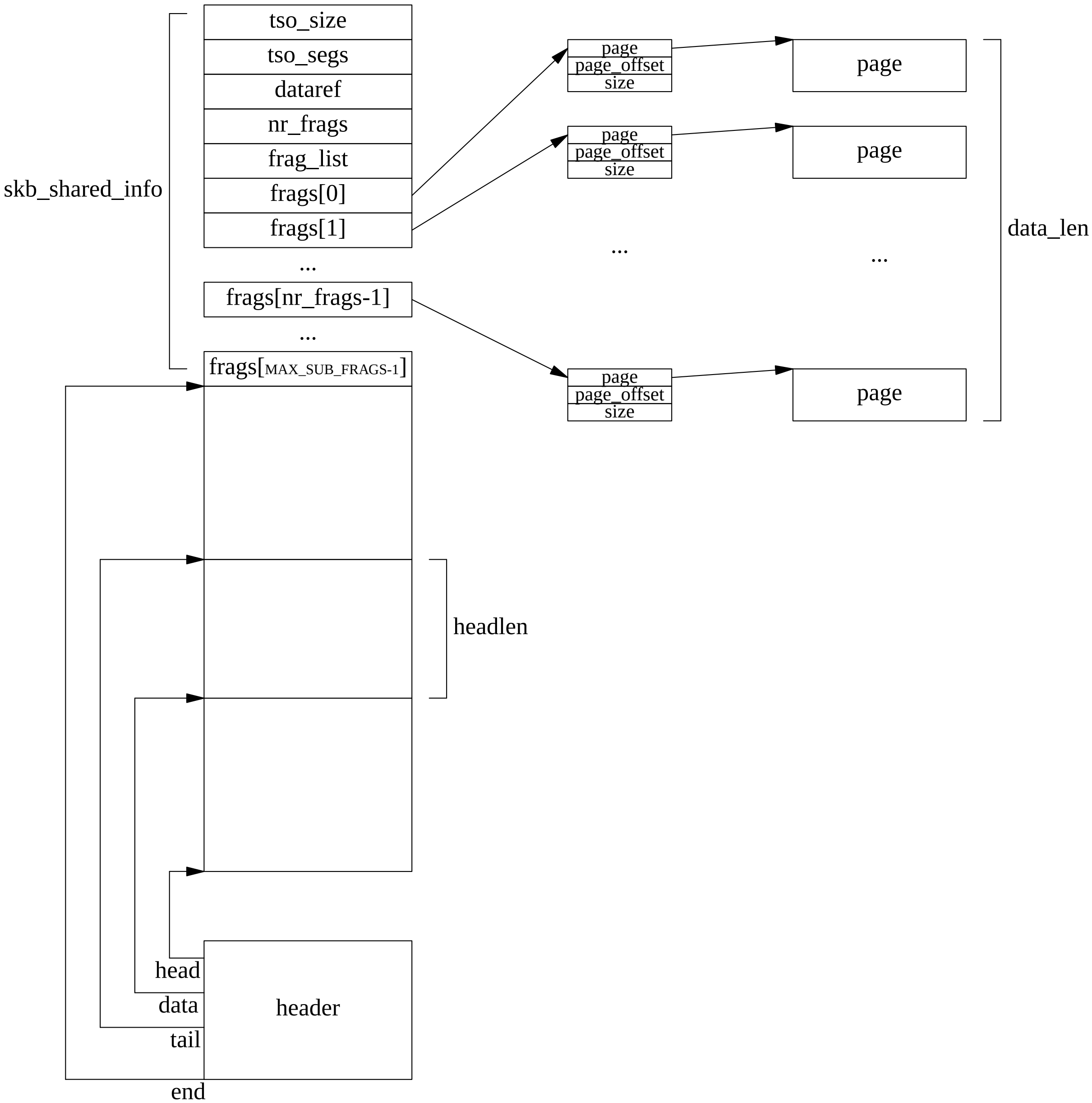


Figure 20 Paged skbuff (use of nr_frags and frags[])

8.3.2. Skbuff chains

Skbuff chains can be treated like a single skbuff and passed around the network stack. This organization is revealed by the `frag_list` pointer of the first skbuff. This pointer points to the second skbuff. The following skbuffs are simply linked through their next pointer. In this way to transform a list of skbuffs, in an skbuff chain it is only necessary to copy the next pointer of the first skbuff over the frag_list pointer. This is what is done inside the ip_local_deliver->ip_defrag->ip_frag_reasm function which transforms a queue of IP fragments stored in an IP fragment queue into an skbuff chain that can be passed around the network stack unitarily. Fragments that are to be forwarded are instead dispatched long before and without any delay in the input process from the ip_rcv_finish function to the ip_forward function.

[net/ipv4/ip_fragment.c]

```
        skb_shinfo(head)->frag_list = head->next;
```

[net/ipv4/ip_fragment.c]

In the recent addition (kernel version 2.6) of the SCTP protocol this kind of fragmented skbuff arises when reassembling the messages of an SCTP stream.

[net/sctp/ulpqueue.c]

```
                skb_shinfo(f_frag)->frag_list = pos;
```

[net/sctp/ulpqueue.c]

(SCTP is a transport protocol offering reliable, multiple stream, multihoming transport service that is TCP friendly and without head-of-line blocking RFC 3286)

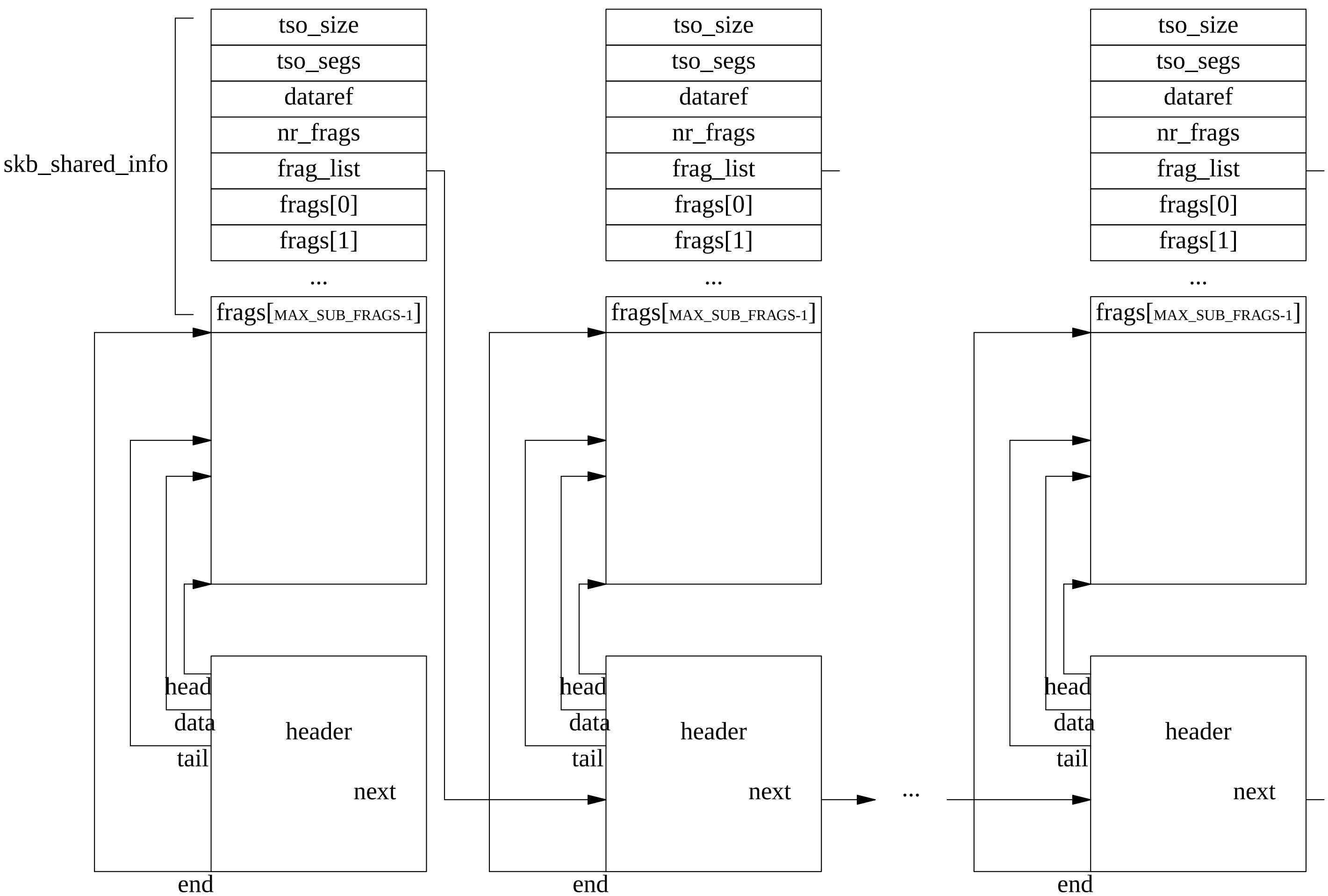


Figure 21 Skbuff chain (use of frag_list pointer)

9. Queues of skbuffs

During output skbuffs are queued first on the socket, and then when the output interface is determined the skbuffs are moved to the device queue. In input the skbuffs are queued on the device queue and then when the owner socket is found are moved to the owner socket queue. These queues are doubly linked lists of skbuffs, formed using the next and prev pointers in the skbuff header. Also IP fragments are stored in such a kind of queues, just that fragments queues are kept in offset order and so insertions can happen also in the middle of the queue.

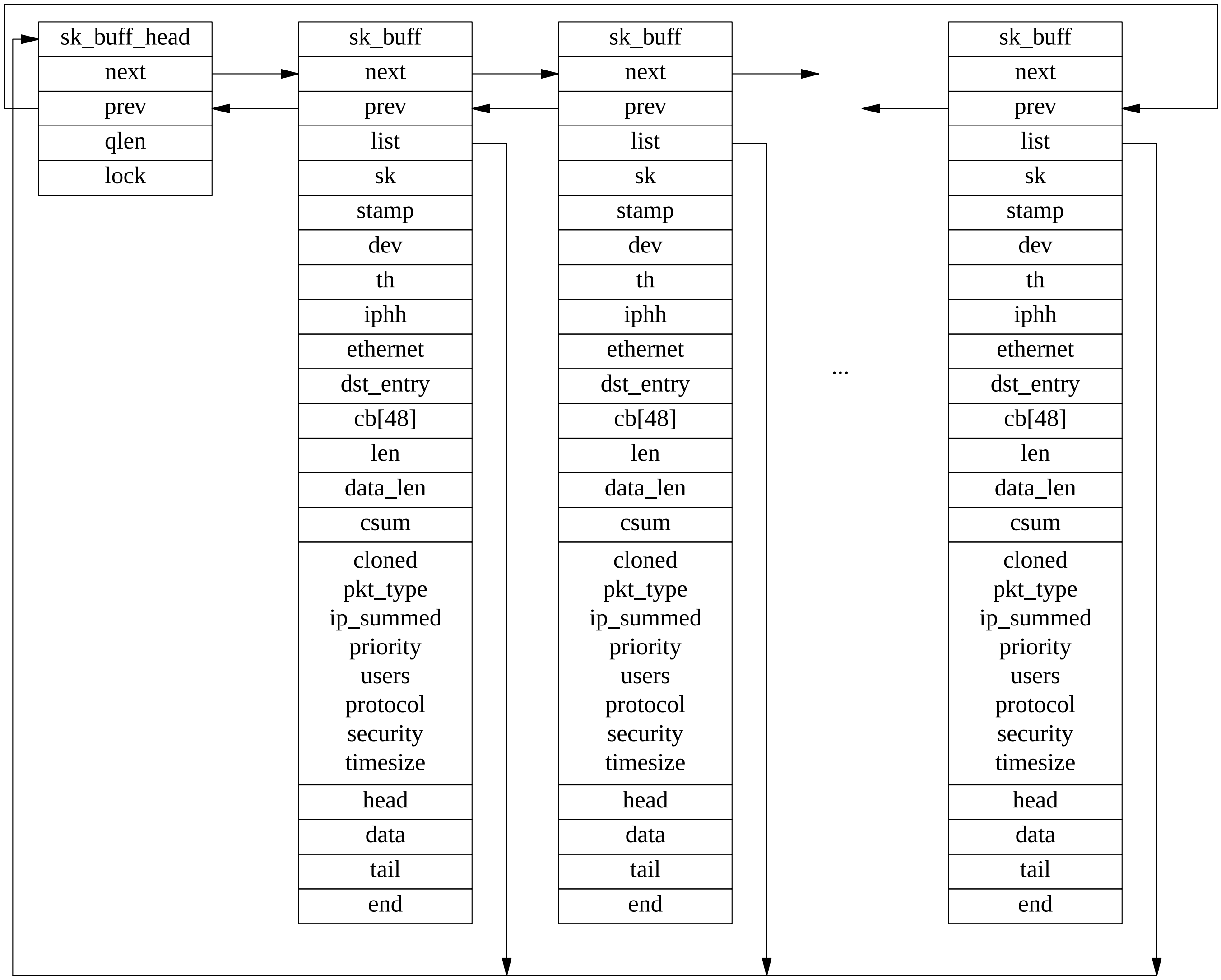


Figure 22 skb_buff.pic

9.1. Functions to manage skbuff queues

9.1.1. skb_queue_len

This function returns the number of skbuffs queued on the list that you pass as an argument. The socket write and read queues for instance (`sk->sk_write_queue` , `sk->sk_receive_queue`) or the device queues `dev->qdisc->q` or the TCP queue of out of order segments `tp->out_of_order_queue` or the `tp->ucopy.prequeue` used to hold skbuffs to pass data to the user in big chunks for efficiency.

File : `[net/sctp/ulpqueue.c]`

[include/linux/skbuff.h]

```
static inline __u32 skb_queue_len(const struct sk_buff_head *list_)
{
     return list_->qlen;
}
```

[include/linux/skbuff.h]

9.1.2. skb_queue_head_init

This function initializes a queue of skbuffs : it initializes the spinlock structure inside the sk_buff_head and sets the `prev` and `next` pointers to the list head and sets the qlen field to zero.

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]

9.1.3. 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, using its spinlock structure and so it is safe to call it 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]

```
 *   @newsk: buffer to queue
 *
 *   Queue a buffer at the start of a list. This function takes no locks
 *   and you must therefore hold required locks before calling it.
 *
 *   A buffer cannot be placed on two lists at the same time.
 */
static inline void __skb_queue_head(struct sk_buff_head *list,
                         struct sk_buff *newsk)
{
     struct sk_buff *prev, *next;

     newsk->list = list;
```

[include/linux/skbuff.h]

[include/linux/skbuff.h]

```
 *
```

```
 *   Queue a buffer at the start of the list. This function takes the
 *   list lock and can be used safely with other locking &sk_buff functions
 *   safely.
 *
 *   A buffer cannot be placed on two lists at the same time.
 */
static inline void skb_queue_head(struct sk_buff_head *list,
                       struct sk_buff *newsk)
```

[include/linux/skbuff.h]

9.1.4. skb_queue_tail and __skb_queue_tail

These two functions queue an skbuff at the tail of an skbuff queue. The `skb_queue_tail` function establishes a spin lock on the queue and so it is safe to call it with interrupts enabled, 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)
{
     unsigned long flags;

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

[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]

9.1.5. skb_entail

File : `[include/linux/skbuff.h]`
This function is part of the TCP code because it is used only by TCP. It enqueues an skbuff at the tail of an skbuff queue using the `__skb_queue_tail` function, but it also does some housekeeping for the TCP protocol. It updates the per socket counters of bytes queued and forwarded. It stores for example some tcp status information on the general 48-bytes control buffer in the skbuff. This information stored on the control buffer comprises TCP start and end sequence, ack flag, sacked.

[net/ipv4/tcp.c]

```
static inline void skb_entail(struct sock *sk, struct tcp_opt *tp,
                      struct sk_buff *skb)
{
     skb->csum = 0;
     TCP_SKB_CB(skb)->seq = tp->write_seq;
     TCP_SKB_CB(skb)->end_seq = tp->write_seq;
     TCP_SKB_CB(skb)->flags = TCPCB_FLAG_ACK;
     TCP_SKB_CB(skb)->sacked = 0;
     __skb_queue_tail(&sk->sk_write_queue, skb);
     tcp_charge_skb(sk, skb);
     if (!tp->send_head)
          tp->send_head = skb;
     else if (tp->nonagle&TCP_NAGLE_PUSH)
          tp->nonagle &= ~TCP_NAGLE_PUSH;
}
```

[net/ipv4/tcp.c]

9.1.6. skb_dequeue and __skb_dequeue

These two functions dequeue an skbuff buffer from the head of an skbuff queue. The `skb_dequeue` function establishes a spin lock on the queue and therefore it is safe to call it 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 : `[net/ipv4/tcp.c]`

[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]

9.1.7. skb_insert and __skb_insert

These functions insert an skbuff in a queue of skbuffs before a given skbuff. The `__skb_insert` function fixes the pointers to do the queue operations, while the `skb_insert` function wraps it with code to set and unset a spinlock on the list.

[include/linux/skbuff.h]

```
static inline void __skb_insert(struct sk_buff *newsk,
                    struct sk_buff *prev, struct sk_buff *next,
                    struct sk_buff_head *list)
{
        newsk->next = next;
        newsk->prev = prev;
        next->prev  = prev->next = newsk;
        newsk->list = list;
        list->qlen++;
}
```

[include/linux/skbuff.h]

9.1.8. skb_append and __skb_append

These function use __skb_insert to append an skbuff on a queue of skbuffs after a given skbuff.

[include/linux/skbuff.h]

```
static inline void __skb_append(struct sk_buff *old, struct sk_buff *newsk)
{
        __skb_insert(newsk, old, old->next, old->list);
}
```

[include/linux/skbuff.h]

9.1.9. skb_unlink and __skb_unlink

These function remove an skbuff from a list. As usual the `skb_unlink` function wraps the __skb_unlink function with code to set and unset a spinlock on the list.

[include/linux/skbuff.h]

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

        list->qlen--;
        next    = skb->next;
        prev    = skb->prev;
        skb->next  = skb->prev = NULL;
        skb->list  = NULL;
        next->prev = prev;
        prev->next = next;
}
```

[include/linux/skbuff.h]

9.1.10. skb_dequeue_tail and __skb_dequeue_tail

These function dequeue the last skbuff on a list.

[include/linux/skbuff.h]

```
static inline struct sk_buff *__skb_dequeue_tail(struct sk_buff_head *list)
{
        struct sk_buff *skb = skb_peek_tail(list);
        if (skb)
                __skb_unlink(skb, list);
        return skb;
}
```

[include/linux/skbuff.h]

10. Skbuff Functions

Many functions have similar names. Two often used conventions on their names are that `funct` is the standard function, and the `__funct` is the streamlined function `funct` without some consistency checks, while `pfunct` is the function to be applied if the skbuff is nonlinear (from a form of fragmented skbuff : the paged skbuff).

The following functions distinguish between the three kind of skbuffs : linear, paged skbuff, skbuff chain.

10.1. Support functions

10.1.1. 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]

10.1.2. SKB_PAGE_ASSERT

File : `[include/linux/skbuff.h]`
This macro will raise a bug if the skbuff is a paged skbuff. 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]

10.1.3. SKB_FRAG_ASSERT

File : `[include/linux/skbuff.h]`
This macro will raise a bug if the skbuff is an skbuff chain. We have already discussed that if the skbuff is fragmented in a list of skbuffs then 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]

10.1.4. 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]

10.1.5. skb_fill_page_desc

File : `[include/linux/skbuff.h]`
This function add a new page to a paged skbuff, filling the relevant information in the shared_info structure of the skbuff.

[include/linux/skbuff.h]

```
static inline void skb_fill_page_desc(struct sk_buff *skb, int i, struct page *page, int off, int size)
{
     skb_frag_t *frag = &skb_shinfo(skb)->frags[i];
     frag->page = page;
     frag->page_offset = off;
     frag->size = size;
     skb_shinfo(skb)->nr_frags = i+1;
}
```

[include/linux/skbuff.h]

10.1.6. pskb_may_pull

File : `[include/linux/skbuff.h]`
This function is used for instance with fragmented skbuffs resulting from IP fragments reassembly. It checks if there are at least len bytes in the skbuff, otherwise it returns false (0). Then if there are less than len bytes in the first skbuff of the list, the skbuff is linearized calling __pskb_pull_tail, in such a way that the required bytes are in the first skbuff, and the pull operation for stripping a header can be performed simply changing the data pointer.

[include/linux/skbuff.h]

```
static inline int pskb_may_pull(struct sk_buff *skb, unsigned int len)
{
     if (likely(len <= skb_headlen(skb)))
          return 1;
     if (unlikely(len > skb->len))
          return 0;
     return __pskb_pull_tail(skb, len-skb_headlen(skb)) != NULL;
}
```

[include/linux/skbuff.h]

10.2. Initialization

10.2.1. skb_init

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

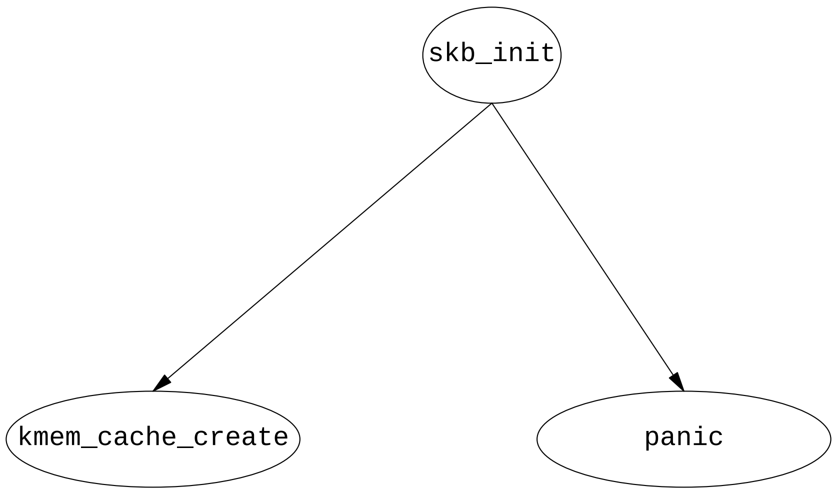


Figure 23 skb_init.pic

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 specify any specific constructor or destructor for them.

[include/net/socket.h]

```
 *   Tune the memory allocator for a new MTU size.
 */
void skb_add_mtu(int mtu)
{
     /* Must match allocation in alloc_skb */
     mtu = SKB_DATA_ALIGN(mtu) + sizeof(struct skb_shared_info);

     kmem_add_cache_size(mtu);
}
#endif
```

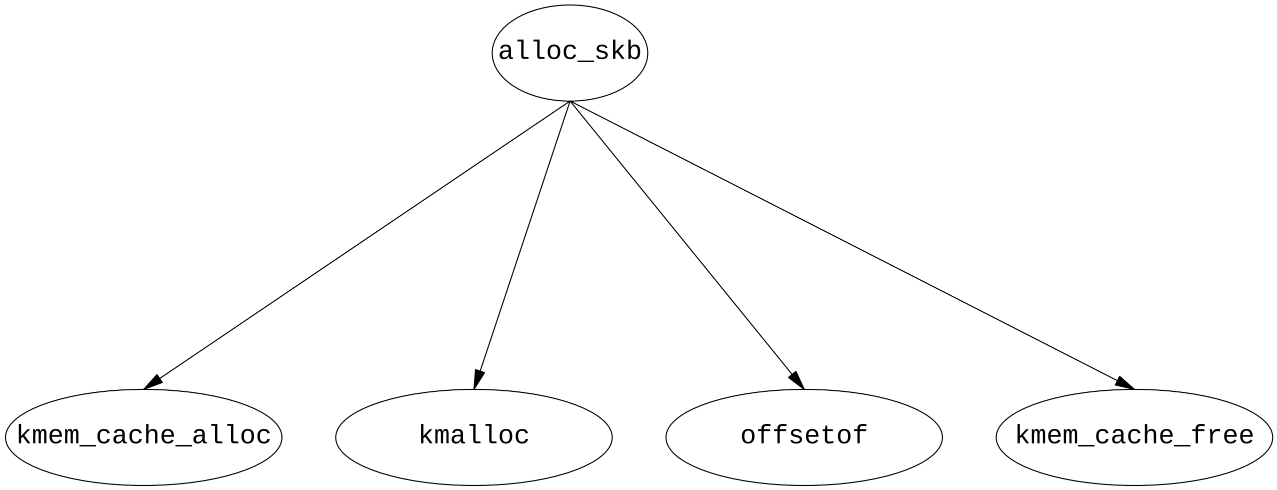
[include/net/socket.h]

10.3. Allocation of skbuffs

10.3.1. alloc_skb, tcp_alloc_skb

File : `[include/net/tcp.h]`
These functions allocate a network buffer for output. The alloc_skb 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 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]

```
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 no I/O is performed over its data, thus the __GFP_DMA flag is reset 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` area that stores the additional information needed by fragmented skbuffs and a reference counter needed by the sharing mechanism. The SKB_DATA_ALIGN macro adds enough bytes to the requested size so that the shared_info area can be aligned with a level 1 cache line (on P4 for example the X86_L1_CACHE_SHIFT is 7 and L1_CACHE_BYTES is then 2^7=128 bytes and so the size is rounded up to the next 128 multiple).

include/linux/cache.h

```
#ifndef SMP_CACHE_BYTES
#define SMP_CACHE_BYTES L1_CACHE_BYTES
#endif
```

include/linux/cache.h

include/linux/skbuff.h

```
#define SKB_DATA_ALIGN(X) (((X) + (SMP_CACHE_BYTES - 1)) & \
                      ~(SMP_CACHE_BYTES - 1))
```

include/linux/skbuff.h

net/core/skbuff.c

```
        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

```
        skb_shinfo(skb)->frag_list = NULL;
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));
```

net/core/skbuff.c

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

net/core/skbuff.c

```
        atomic_set(&skb->users, 1);
        skb->head = data;
        skb->data = data;
        skb->tail = data;
        skb->end  = data + size;
```

net/core/skbuff.c

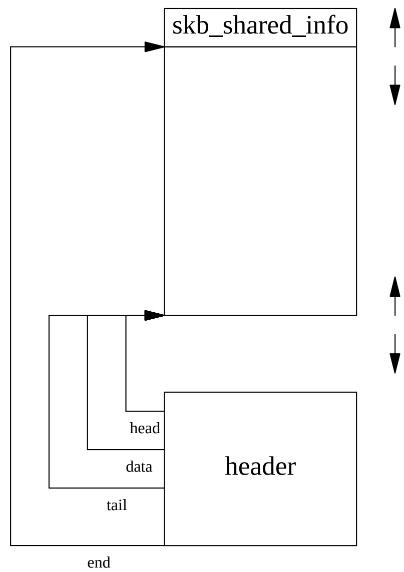


Figure 24 skbuff header and data area immediately after allocation

The reference count is set to 1 as the header currently allocated is the only one referencing the data area

net/core/skbuff.c

```
        atomic_set(&(skb_shinfo(skb)->dataref), 1);
        skb_shinfo(skb)->nr_frags  = 0;
```

net/core/skbuff.c

and the counter for the external pages is set to 0 which indicates this is not a paged 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

The tso fields (for the TCP Segmentation Offloading support) are set to 0 because the network buffer is allocated for a generic interface and eventually only later in the process of sending the segment over an interface they can be set differently if the specific interface supports TSO . This initial allocation creates a simple skbuff, and not an skbuff chain and this is indicated setting the fragment list to NULL.

Figure 25 Immediately after alloc_skb all the room in an skbuff is in the tail

No space is reserved as headroom with alloc_skb. In case a headroom is needed it has to be reserved separately.

The tcp code instead uses its own skbuff allocator `tcp_alloc_skb` to request additional `MAX_TCP_HEADER` bytes to accomodate a headroom sufficient for the TCP/IP header (usually this is 128+ MAX_HEADER 32= MAX_TCP_HEADER = 160 bytes).This function allocates only a linear skbuff. The tcp_alloc_skb function is simply a 1-line wrapper around the more general tcp_alloc_pskb function that can allocate paged skbuffs when the mem argument is different from 0 (in this case the size argument is only the size of the linear part of the skbuff and mem is the size of the paged part).

net/core/skbuff.c

```
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);
}
```

net/core/skbuff.c

The tcp_alloc functions reserve as headroom with skb_reserve the additional space allocated for a maximum TCP header (MAX_TCP_HEADER = 128 usually).

Figure 26 tcp_alloc_skb allocates additional space sufficient for headers, and reserves it as headroom

Another allocation of skbuffs in the output process that requires some size adjustment is in the IP fragmentation case. In this situation the IP datagram has to be split and additional bytes for the IP header and the link layer header are requested for each of the chunks following the first one (the link layer size requested is rounded up to allow an efficient alignment of the IP header) :

[net/ipv4/ip_output.c]

```
		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;
```

```
              }

              /*
               *   Set up data on packet
               */

              ip_copy_metadata(skb2, skb);
              skb_reserve(skb2, LL_RESERVED_SPACE(rt->u.dst.dev));
```

[net/ipv4/ip_output.c]

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

Figure 27 the IP fragmentation code allocates additional space for the current IP header and the link header, and reserves as headroom the space for the link header

10.3.2. sock_alloc_send_pskb,sock_alloc_send_skb

These functions allocate an skbuff for a socket. There is no receive equivalent because when a packet arrives, the socket is not known immediately. sock_alloc_send_skb can allocate only linear skbuffs, and so it is simply a 1-line wrapper around the sock_alloc_send_pskb that can allocate also paged skbuffs. header_len is the size to be allocated in the linear part of the skbuff, while data_len as usual is the size to be allocated using external pages. The function sock_alloc_send_pskb checks if the memory limit allows the allocation. Then it allocates a linear skbuff for header_len bytes. If paged memory is requested (data_len != 0), then it computes the number of pages necessary, allocates the pages, and initializes the pointers to them in the skbuff shared_info area. It then sets the owner of the skbuff to the socket and returns the skbuff. This function can wait until memory is available : when it starts it sets a timeout on the socket, if memory is not available it sleeps on memory until it is available or the timeout expires and it reports an error.

[net/core/sock.c]

```
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;
      unsigned int 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);
      while (1) {
            err = sock_error(sk);
```

```
        if (err != 0)
            goto failure;

        err = -EPIPE;
        if (sk->sk_shutdown & SEND_SHUTDOWN)
            goto failure;

        if (atomic_read(&sk->sk_wmem_alloc) < sk->sk_sndbuf) {
            skb = alloc_skb(header_len, sk->sk_allocation);
            if (skb) {
                int npages;
                int i;

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

                npages = (data_len + (PAGE_SIZE - 1)) >> PAGE_SHIFT;
                skb->truesize += data_len;
                skb_shinfo(skb)->nr_frags = npages;
                for (i = 0; i < npages; i++) {
                    struct page *page;
                    skb_frag_t *frag;

                    page = alloc_pages(sk->sk_allocation, 0);
                    if (!page) {
                        err = -ENOBUFS;
                        skb_shinfo(skb)->nr_frags = i;
                        kfree_skb(skb);
                        goto failure;
                    }

                    frag = &skb_shinfo(skb)->frags[i];
                    frag->page = page;
                    frag->page_offset = 0;
                    frag->size = (data_len >= PAGE_SIZE ?
                                PAGE_SIZE :
                                data_len);
                    data_len -= PAGE_SIZE;
                }

                /* Full success... */
                break;
            }
            err = -ENOBUFS;
            goto failure;
        }
        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);
        }

        skb_set_owner_w(skb, sk);
        return skb;

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

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);
}
```

[net/core/sock.c]

10.3.3. sock_wmalloc,sock_rmalloc

sk_sndbuf and sk_rcvbuf are two socket variables initially set to the global variables syctl configurable syctl_wmem_default,sysctl_rmem_default. Their values can later be tuned, for instance when a tcp socket enters the established state. These 2 functions allocate an skbuff of the requested size checking, unless the argument force is set, that the memory for reading or writing owned by the socket is less than the memory allowed.

[net/core/sock.c]

```
struct sk_buff *sock_wmalloc(struct sock *sk, unsigned long size, int force, int priority)
{
        if (force || atomic_read(&sk->sk_wmem_alloc) < sk->sk_sndbuf) {
                struct sk_buff * skb = alloc_skb(size, priority);
                if (skb) {
                        skb_set_owner_w(skb, sk);
                        return skb;
                }
        }
        return NULL;
}

/*
 * Allocate a skb from the socket's receive buffer.
 */
struct sk_buff *sock_rmalloc(struct sock *sk, unsigned long size, int force, int priority)
{
        if (force || atomic_read(&sk->sk_rmem_alloc) < sk->sk_rcvbuf) {
                struct sk_buff *skb = alloc_skb(size, priority);
                if (skb) {
                        skb_set_owner_r(skb, sk);
                        return skb;
```

```
        }
    }
    return NULL;
}
```

[net/core/sock.c]

bt_skb_alloc [include/net/bluetooth/bluetooth.h] is a special skb allocation function, it simply allocates and reserves as headroom additional 8 bytes, and initializes to 0 a control variable kept in the skb control buffer.

dn_alloc_skb [net/decnet/dn_nsp_out.c] is a special allocation function for decnet. it allocates additional 64 bytes for header and reserves them as headroom

10.3.4. dev_alloc_skb and __dev_alloc_skb

These function are used to allocate an skbuff for an incoming packet at the device driver level. The dev_alloc_skb function is a 1-line wrapper that immediately calls the __dev_alloc_skb function. The only argument for dev_alloc_skb is the size in bytes of the requested skb. This function calls the double underlined function with the GFP_ATOMIC specification. This __dev_alloc_skb function doesn't use a parameterized link layer header size, it simply allocates 16 more bytes than those requested to leave space for an ethernet header (14 bytes) and keep the IP header 16 bytes aligned for efficiency reasons (cache). This additional space is reserved as headroom calling skb_reserve.

[include/linux/skbuff.h]

```
static inline struct sk_buff *__dev_alloc_skb(unsigned int length,
                                int gfp_mask)
{
    struct sk_buff *skb = alloc_skb(length + 16, gfp_mask);
    if (likely(skb))
        skb_reserve(skb, 16);
    return skb;
}

/**
 *  dev_alloc_skb - allocate an skbuff for sending
 *  @length: length to allocate
 *
 *  Allocate a new &sk_buff and assign it a usage count of one. The
 *  buffer has unspecified headroom built in. Users should allocate
 *  the headroom they think they need without accounting for the
 *  built in space. The built in space is used for optimisations.
 *
 *  %NULL is returned in there is no free memory. Although this function
 *  allocates memory it can be called from an interrupt.
 */
static inline struct sk_buff *dev_alloc_skb(unsigned int length)
{
    return __dev_alloc_skb(length, GFP_ATOMIC);
}
```

[include/linux/skbuff.h]

The __dev_alloc_skb function allocates the skb requesting 16 additional bytes of data space using the standard alloc_skb (this allocates an skb having a data area of the size requested plus the size of the shared_info struct). Then _dev_alloc_skb shifts the data and tail pointers of 16 bytes with skb_reserve.

Figure 28 these functions allocate and reserve as headroom 16 bytes for the link header

10.4. Data Pointer Manipulation

10.4.1. skb_reserve

File : `[include/linux/skbuff.h]`
When an skbuff is allocated all the free room is allocated in the tail, after the data and tail pointers position.

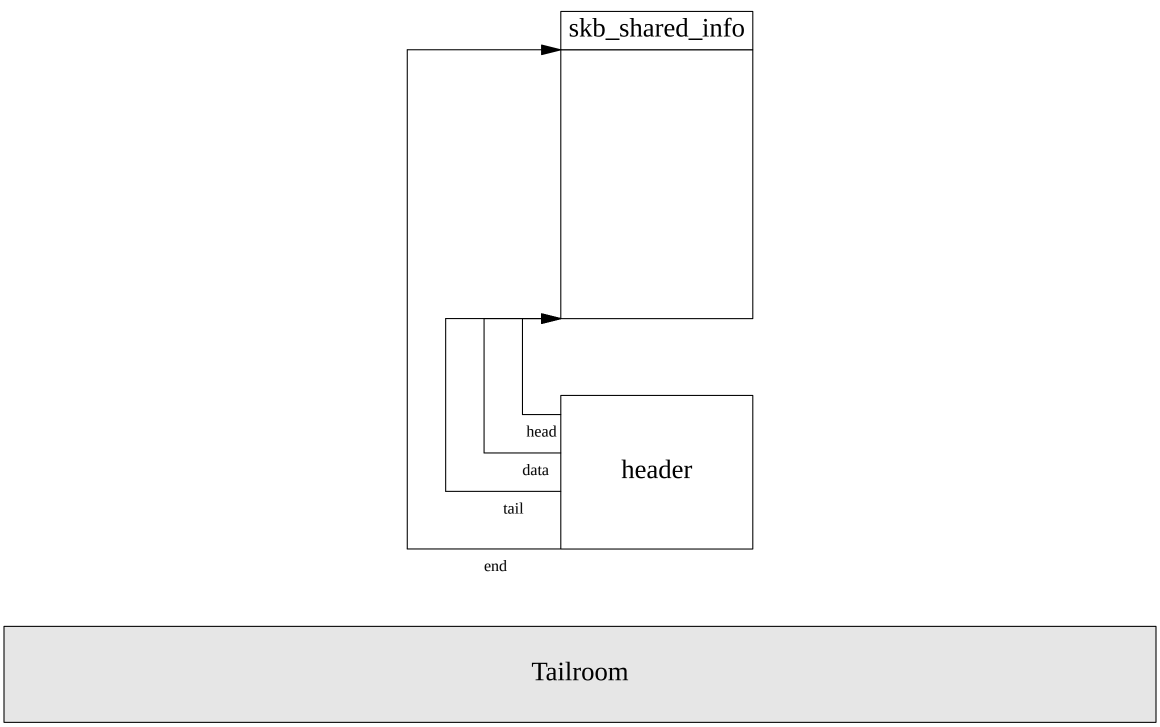


Figure 29 Skbuff before any headroom reservation

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.
```

[include/linux/skbuff.h]

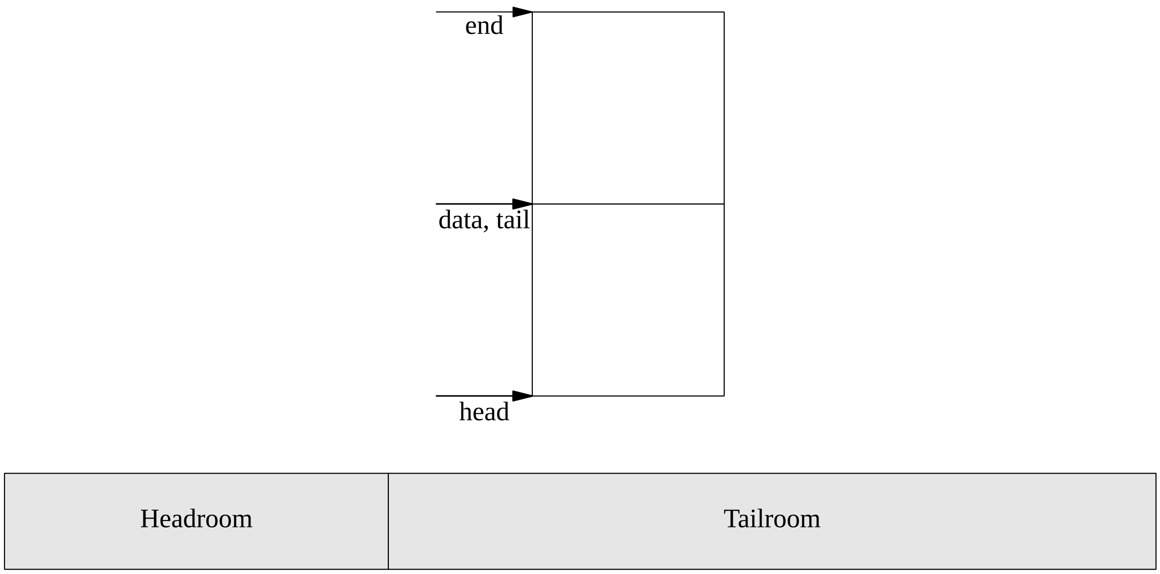
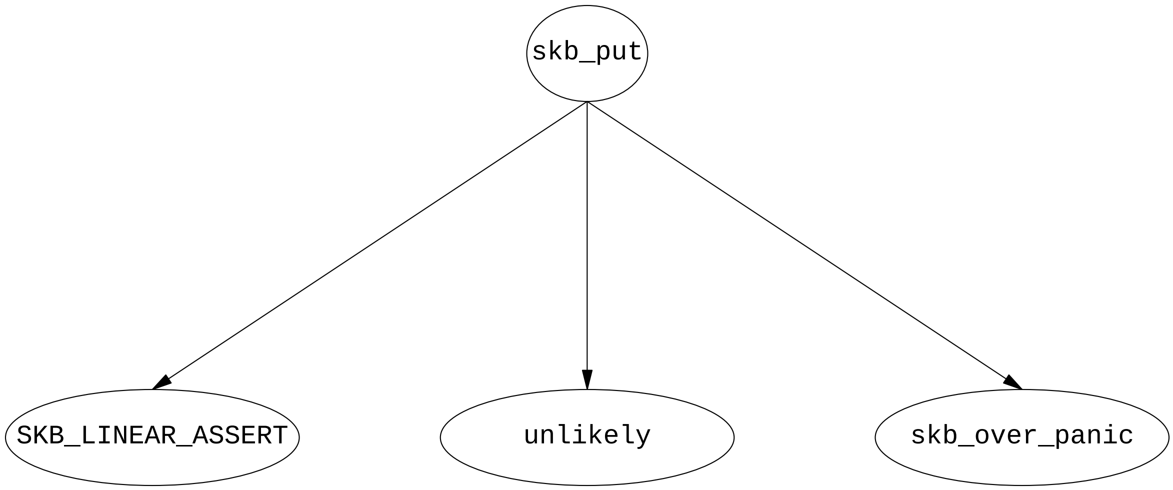
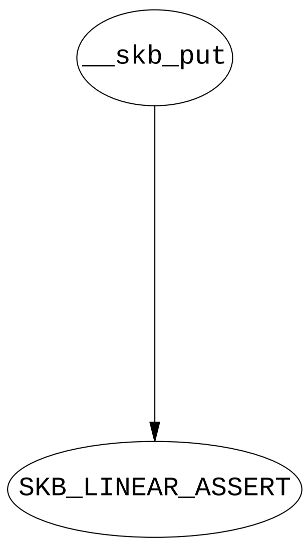


Figure 30 Skbuff after reserving some headroom

10.4.2. skb_put and __skb_put

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). After calling the skb_reserve() function to move the data pointer, skb_put/__skb_put is then usually called to move the tail pointer and prepare the space for copying over the data. It updates the len field of the skbuff header, it doesn't perform any copy. This operation can be applied only on linear skbuffs.





[include/linux/skbuff.h]

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

[include/linux/skbuff.h]

```
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;
}
```

[include/linux/skbuff.h]

```
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]

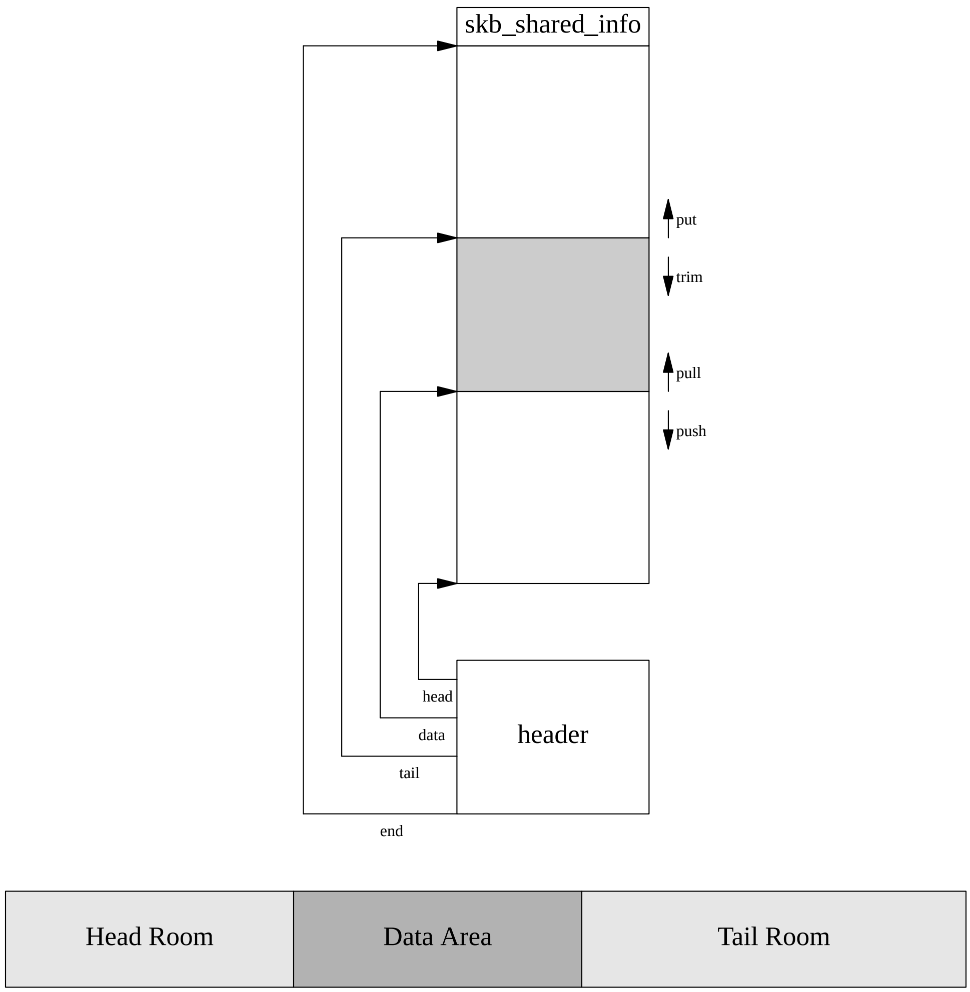


Figure 31 An skbuff after calling skb_put

10.4.3. skb_add_data

File : `[net/ipv4/tcp.c]`
This function is inside the tcp code because it is used only by TCP. It uses the skb_put function to make room (advancing the tail pointer) for copying `copy` bytes of data. Differently from skb_put, this function also copies the data with the csum_and_copy_from_user or copy_from_user function accordingly if the interface doesn't support or supports the checksum in hardware.

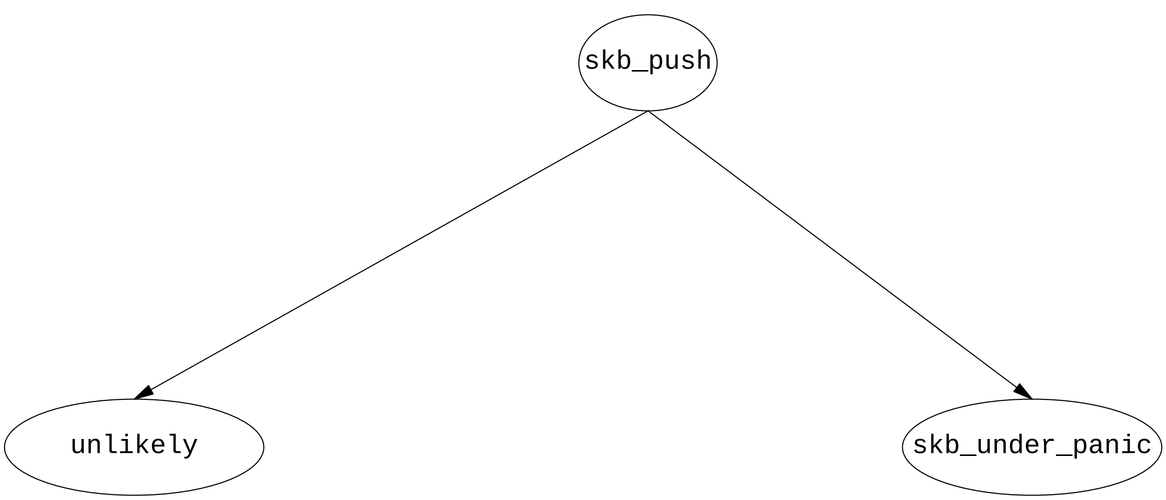
[net/ipv4/tcp.c]

```
static inline int skb_add_data(struct sk_buff *skb, char *from, int copy)
{
	int err = 0;
	unsigned int csum;
	int off = skb->len;

	if (skb->ip_summed == CHECKSUM_NONE) {
		csum = csum_and_copy_from_user(from, skb_put(skb, copy),
					       copy, 0, &err);
		if (!err) {
			skb->csum = csum_block_add(skb->csum, csum, off);
			return 0;
		}
	} else {
		if (!copy_from_user(skb_put(skb, copy), from, copy))
			return 0;
	}

	__skb_trim(skb, off);
	return -EFAULT;
}
```

[net/ipv4/tcp.c]

10.4.4. 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);
```

[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]

```
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]

```
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. The skb_push function simply doesn't perform the consistency check.

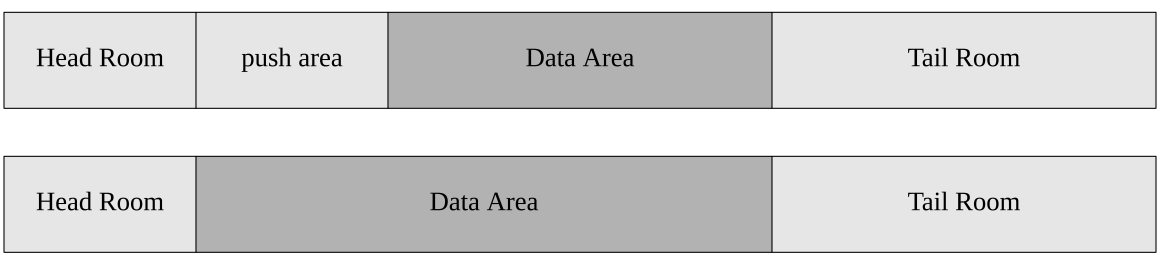
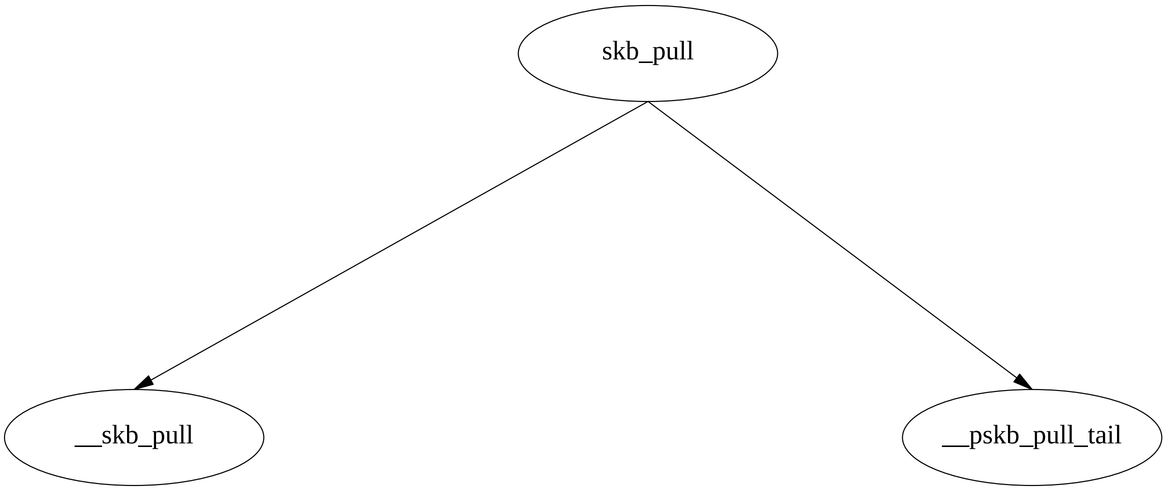


Figure 32 Skbuff before and after a push operation

10.4.5. skb_pull()

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



The skb_pull function is used to strip away the headers during input processing. 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 is performed inside `__skb_pull` to check that `skb->len` is greater than or equal to `skb->data_len.` If that is not true, this would mean that the beginning of the data would be inside the nonlinear part of the skbuff, and so it would need to be rearranged, for this reason a bug is raised.

[include/linux/skbuff.h]

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

[include/linux/skbuff.h]

```
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;
}
```

[include/linux/skbuff.h]

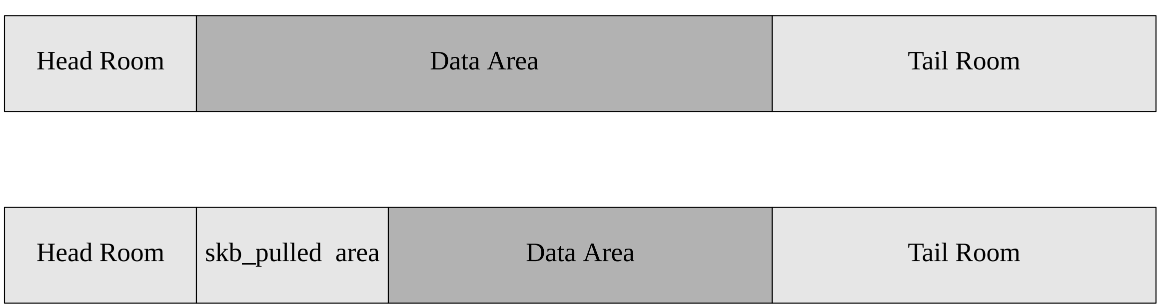
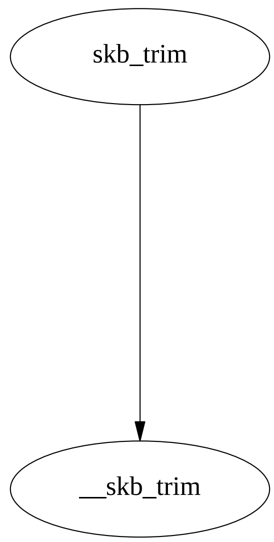


Figure 33 Skbuff before and after a pull operation

10.4.6. skb_trim, __skb_trim, __pskb_trim

File : `include/linux/skbuff.h`

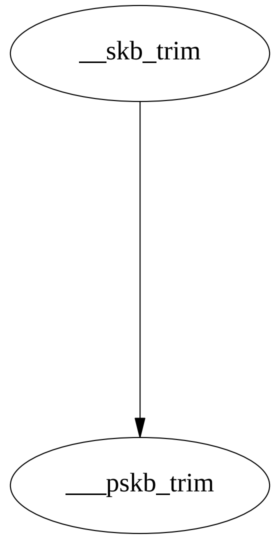


These functions trim the data area at its end. skb_trim is the highest level function. It is a wrapper function that makes a consistency check to see if the total data in the skbuff is sufficient to satisfy the request and then calls __skb_trim()

include/linux/skbuff.h

```
 *   skb_trim - remove end from a buffer
 *   @skb: buffer to alter
 *   @len: new length
 *
 *   Cut the length of a buffer down by removing data from the tail. If
 *   the buffer is already under the length specified it is not modified.
```

include/linux/skbuff.h

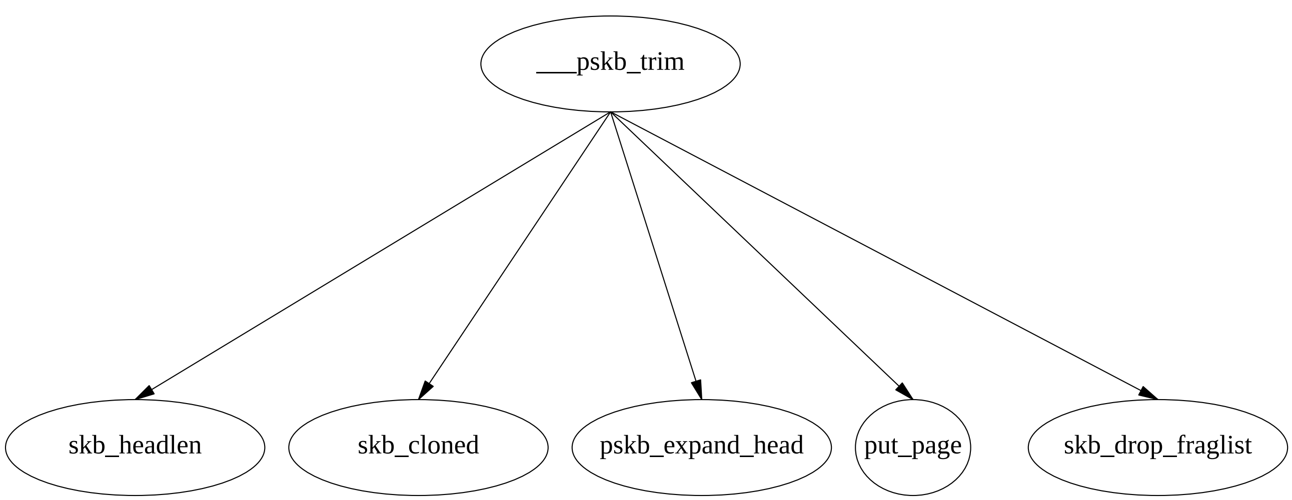


If the skbuff is linear (skb->data_len == 0) `__skb_trim` simply sets the total data length to len and trims the skb->tail pointer. Otherwise if the skbuff is nonlinear it calls the paged skbuff version `___pskb_trim()`.

include/linux/skbuff.h

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

extern int ___pskb_trim(struct sk_buff *skb, unsigned int len, int realloc);

static inline void __skb_trim(struct sk_buff *skb, unsigned int len)
{
```

include/linux/skbuff.h



This function is called to trim at the end a generic skbuff (be it linear or not). The len argument will be the new size of the data area. The operation is used for instance to remove any padding added by other network layers to an IP datagram after having determined the correct size from the IP header :

net/ipv4/ip_input.c

```
        if (skb->len > len) {
            __pskb_trim(skb, len);
            if (skb->ip_summed == CHECKSUM_HW)
                skb->ip_summed = CHECKSUM_NONE;
        }
```

net/ipv4/ip_input.c

or to trim back an skbuff to its initial size after having attempted unsuccessfully to add some data to it (see skb_add_data) .

An skbuff can have an array of external pages (`skb->frags[]`) and/or be an skbuff chain (`skb->fraglist`).

net/core/skbuff.c

```
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;
    }

    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

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.

include/linux/skbuff.h

```
                        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;
        }

        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;
```

include/linux/skbuff.h

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

10.4.7. tcp_trim_head, __pskb_trim_head

These functions are used only by TCP. They cut len bytes from the beginning of the data area. tcp_trim_head is the more general of these functions. If the skbuff is cloned (2 headers using the same data) then it is copied over to a newly allocated linear skbuff (pskb_expand_head). If there are len bytes or more in the linear part of the skbuff (for sure this happens if it is a linear skbuff) then this operation reduces to a pull operation (__skb_pull), otherwise the paged version of this function is called (__pskb_trim_head). __pskb_trim_head runs through the external pages of the paged skbuff and releases all the pages comprised in the first len bytes. Then it adjusts the number of pages left and the page_offset of the first page, and it collapses the data area in the linear part (skb->tail = skb->data).

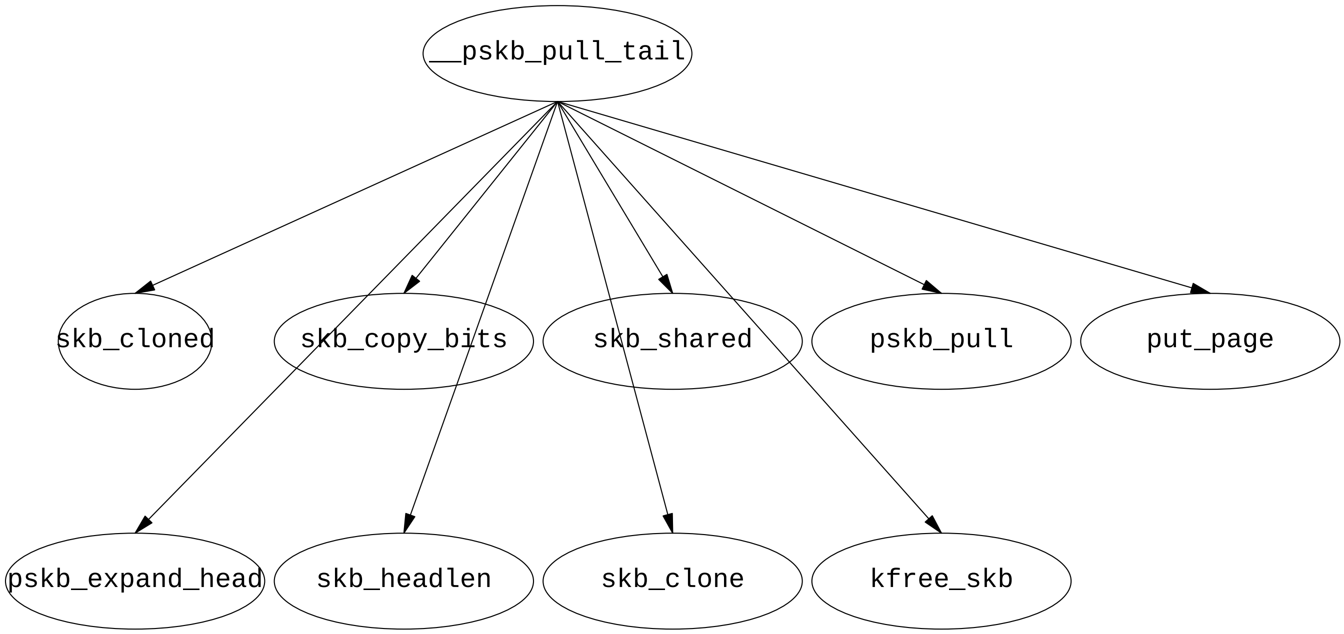
[net/ipv4/tcp_output.c]

```
unsigned char * __pskb_trim_head(struct sk_buff *skb, int len)
{
        int i, k, eat;

        eat = len;
        k = 0;
        for (i=0; i<skb_shinfo(skb)->nr_frags; i++) {
                if (skb_shinfo(skb)->frags[i].size <= eat) {
                        put_page(skb_shinfo(skb)->frags[i].page);
                        eat -= skb_shinfo(skb)->frags[i].size;
                } else {
                        skb_shinfo(skb)->frags[k] = skb_shinfo(skb)->frags[i];
                        if (eat) {
                                skb_shinfo(skb)->frags[k].page_offset += eat;
                                skb_shinfo(skb)->frags[k].size -= eat;
                                eat = 0;
                        }
                        k++;
                }
        }
        skb_shinfo(skb)->nr_frags = k;

        skb->tail = skb->data;
        skb->data_len -= len;
        skb->len = skb->data_len;
        return skb->tail;
}

static int tcp_trim_head(struct sock *sk, struct sk_buff *skb, u32 len)
{
        if (skb_cloned(skb) &&
              pskb_expand_head(skb, 0, 0, GFP_ATOMIC))
                return -ENOMEM;

        if (len <= skb_headlen(skb)) {
                __skb_pull(skb, len);
        } else {
                if (__pskb_trim_head(skb, len-skb_headlen(skb)) == NULL)
                        return -ENOMEM;
        }

        TCP_SKB_CB(skb)->seq += len;
        skb->ip_summed = CHECKSUM_HW;
        return 0;
}
```

[net/ipv4/tcp_output.c]

10.4.8. __pskb_pull_tail

File : [net/ipv4/tcp_output.c]



The purpose of this function is to put the headers of a packet in the linear part of an skbuff, this is required to parse the headers with the usual pointer arithmetic that is possible only on a contiguous data area. 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 (delta - (skb->end - skb->tail) > 0), 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. It allocates a new linear part also in the case the skbuff has been cloned since that data area needs to be changed. Then delta bytes are copied 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 it is sufficient to pull the array of pages, otherwise the frag_list needs to be scanned. If the array of pages associated with this skbuff is large enough then again it is sufficient to pull the array. Then going through the frag_list and eating (kfree_skb) all the complete skbuffs that can be eaten. 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.

[net/ipv4/tcp_output.c]

```
unsigned char *__pskb_pull_tail(struct sk_buff *skb, int delta)
{
     /* If skb has not enough free space at tail, get new one
      * plus 128 bytes for future expansions. If we have enough
      * room at tail, reallocate without expansion only if skb is cloned.
      */
     int i, k, eat = (skb->tail + delta) - skb->end;

     if (eat > 0 || skb_cloned(skb)) {
          if (pskb_expand_head(skb, 0, eat > 0 ? eat + 128 : 0,
```

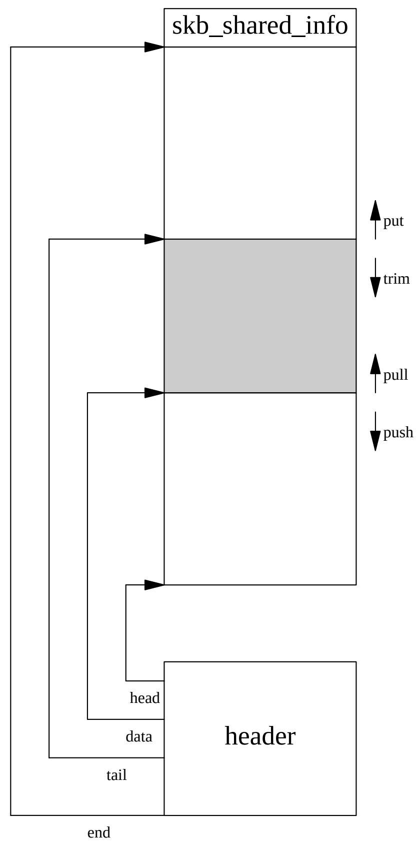
```
                    GFP_ATOMIC))
                return NULL;
    }

    if (skb_copy_bits(skb, skb_headlen(skb), skb->tail, delta))
        BUG();

    /* Optimization: no fragments, no reasons to preestimate
     * size of pulled pages. Superb.
     */
    if (!skb_shinfo(skb)->frag_list)
        goto pull_pages;

    /* Estimate size of pulled pages. */
    eat = delta;
    for (i = 0; i < skb_shinfo(skb)->nr_frags; i++) {
        if (skb_shinfo(skb)->frags[i].size >= eat)
            goto pull_pages;
        eat -= skb_shinfo(skb)->frags[i].size;
    }

    /* If we need update frag list, we are in troubles.
     * Certainly, it possible to add an offset to skb data,
     * but taking into account that pulling is expected to
     * be very rare operation, it is worth to fight against
     * further bloating skb head and crucify ourselves here instead.
     * Pure masohism, indeed. 8)8)
     */
    if (eat) {
        struct sk_buff *list = skb_shinfo(skb)->frag_list;
        struct sk_buff *clone = NULL;
        struct sk_buff *insp = NULL;

        do {
            if (!list)
                BUG();

            if (list->len <= eat) {
                /* Eaten as whole. */
                eat -= list->len;
                list = list->next;
                insp = list;
            } else {
                /* Eaten partially. */

                if (skb_shared(list)) {
                    /* Sucks! We need to fork list. :-( */
                    clone = skb_clone(list, GFP_ATOMIC);
                    if (!clone)
                        return NULL;
                    insp = list->next;
                    list = clone;
                } else {
                    /* This may be pulled without
```

```
                          * problems. */
                         insp = list;
                    }
                    if (!pskb_pull(list, eat)) {
                         if (clone)
                              kfree_skb(clone);
                         return NULL;
                    }
                    break;
               }
          } while (eat);

          /* Free pulled out fragments. */
          while ((list = skb_shinfo(skb)->frag_list) != insp) {
               skb_shinfo(skb)->frag_list = list->next;
               kfree_skb(list);
          }
          /* And insert new clone at head. */
          if (clone) {
               clone->next = list;
               skb_shinfo(skb)->frag_list = clone;
          }
     }
     /* Success! Now we may commit changes to skb data. */

pull_pages:
     eat = delta;
     k = 0;
     for (i = 0; i < skb_shinfo(skb)->nr_frags; i++) {
          if (skb_shinfo(skb)->frags[i].size <= eat) {
               put_page(skb_shinfo(skb)->frags[i].page);
               eat -= skb_shinfo(skb)->frags[i].size;
          } else {
               skb_shinfo(skb)->frags[k] = skb_shinfo(skb)->frags[i];
               if (eat) {
                    skb_shinfo(skb)->frags[k].page_offset += eat;
                    skb_shinfo(skb)->frags[k].size -= eat;
                    eat = 0;
               }
               k++;
          }
     }
     skb_shinfo(skb)->nr_frags = k;

     skb->tail     += delta;
     skb->data_len -= delta;

     return skb->tail;
}
```

[net/ipv4/tcp_output.c]

------------------ 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 :

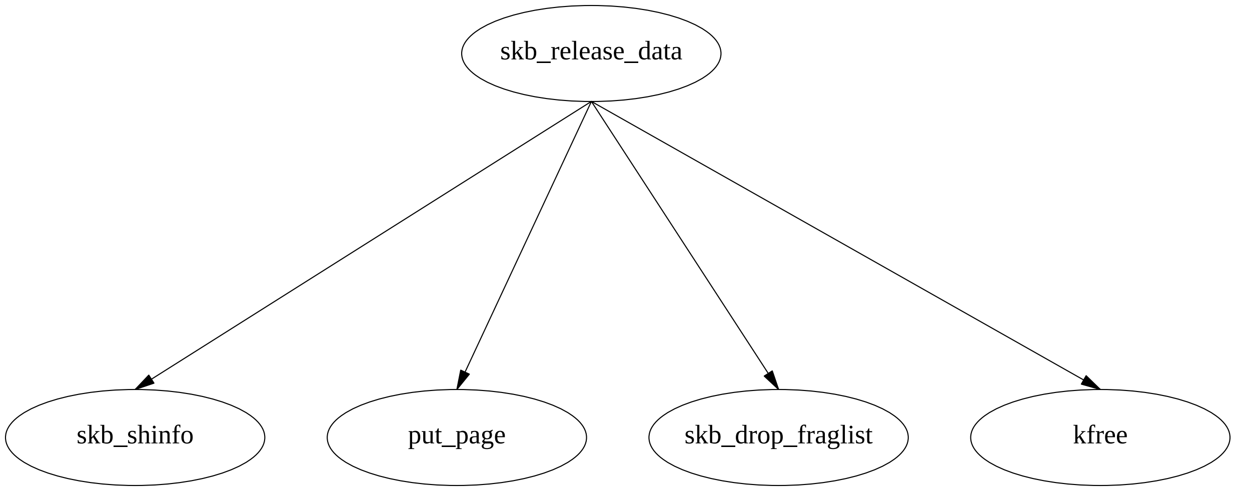
```
skb->data : push .. extends the used data area towards the beginning of
                    the buffer skb->head
            pull .. shrinks the beginning of the used data area
skb->tail : trim .. shrinks the end of the used data area
            put .. extends the end of the used data area towards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

10.5. Releasing skbuffs

10.5.1. 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]

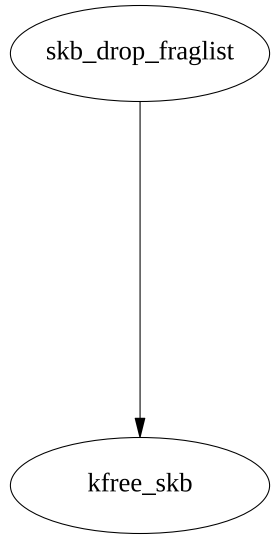
```

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]

The `skb->data_len` field is different from zero only on nonlinear skbuffs. In fact the function `skb_is_nonlinear()` returns that field. It represents the number of bytes in the nonlinear part of the skbuff. The function `skb_headlen()` returns the bytes in the linear part of the skbuff : `skb->len - skb->data_len`.

10.5.2. 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.

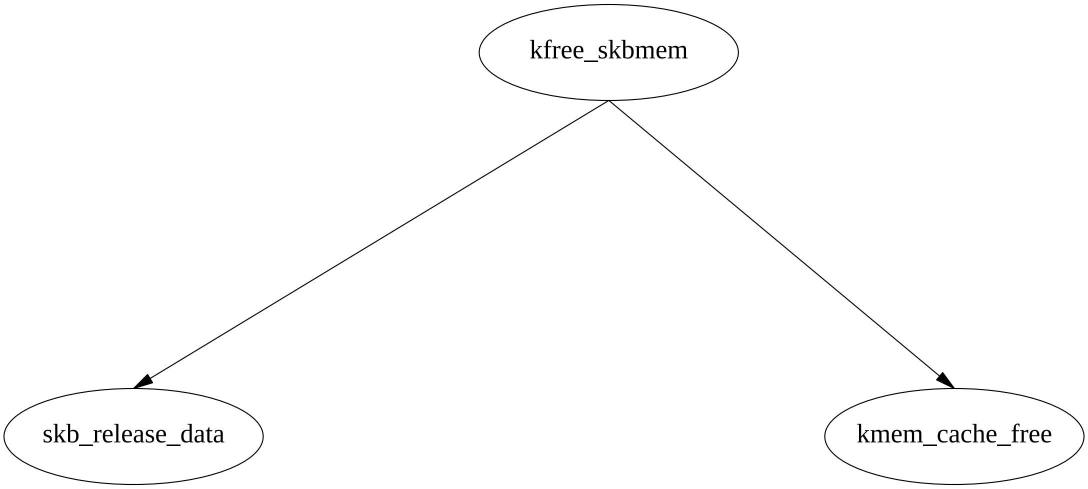
[net/core/skbuff.c]

```

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]

10.5.3. 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.

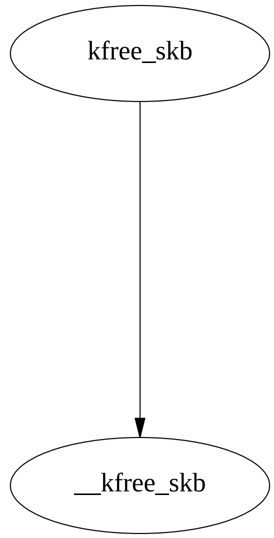
[net/core/skbuff.c]

```

/*
 *   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]

10.5.4. 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.

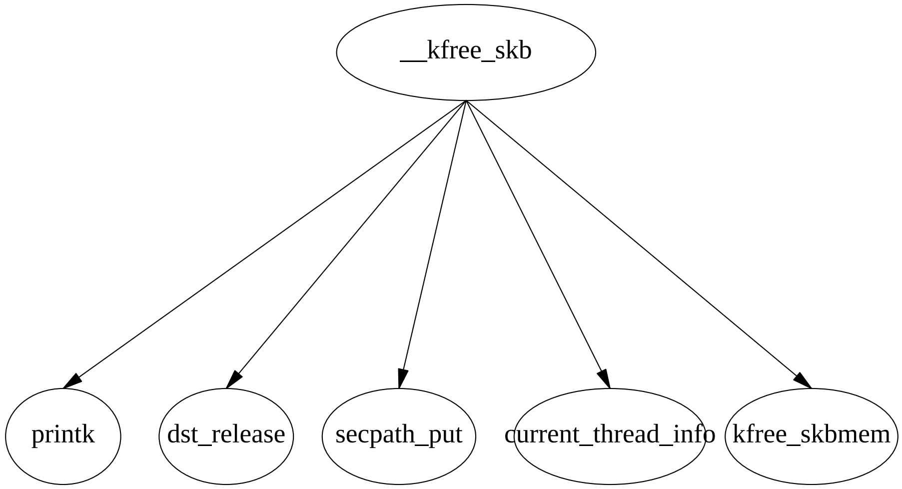
[include/linux/skbuff.h]

```
/*
 * If users == 1, we are the only owner and are can avoid redundant
 * atomic change.
 */

/**
 *   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)
```

[include/linux/skbuff.h]

10.5.5. __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);
#ifdef CONFIG_BRIDGE_NETFILTER
        nf_bridge_put(skb->nf_bridge);
#endif
#endif
        kfree_skbmem(skb);
}
```

[net/core/skbuff.c]

10.6. skb_split

File : `[net/core/skbuff.c]`
This function is in the [ipv4/tcp_output.c] file because it is used only by the TCP code in case it needs to fragment a segment (tcp_fragment).

[net/core/skbuff.c]

```
static void skb_split(struct sk_buff *skb, struct sk_buff *skb1, u32 len)
{
        int i;
        int pos = skb_headlen(skb);

        if (len < pos) {
                /* Split line is inside header. */
                memcpy(skb_put(skb1, pos-len), skb->data + len, pos-len);

                /* And move data appendix as is. */
                for (i = 0; i < skb_shinfo(skb)->nr_frags; i++)
                        skb_shinfo(skb1)->frags[i] = skb_shinfo(skb)->frags[i];

                skb_shinfo(skb1)->nr_frags = skb_shinfo(skb)->nr_frags;
                skb_shinfo(skb)->nr_frags = 0;

                skb1->data_len = skb->data_len;
                skb1->len += skb1->data_len;
                skb->data_len = 0;
                skb->len = len;
                skb->tail = skb->data+len;
        } else {
                int k = 0;
                int nfrags = skb_shinfo(skb)->nr_frags;

                /* Second chunk has no header, nothing to copy. */

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

```
            skb1->len = skb1->data_len = skb->len - len;
            skb->len = len;
            skb->data_len = len - pos;

            for (i=0; i<nfrags; i++) {
                 int size = skb_shinfo(skb)->frags[i].size;
                 if (pos + size > len) {
                      skb_shinfo(skb1)->frags[k] = skb_shinfo(skb)->frags[i];

                      if (pos < len) {
                           /* Split frag.
                            * We have to variants in this case:
                            * 1. Move all the frag to the second
                            *    part, if it is possible. F.e.
                            *    this approach is mandatory for TUX,
                            *    where splitting is expensive.
                            * 2. Split is accurately. We make this.
                            */
                           get_page(skb_shinfo(skb)->frags[i].page);
                           skb_shinfo(skb1)->frags[0].page_offset += (len-pos);
                           skb_shinfo(skb1)->frags[0].size -= (len-pos);
                           skb_shinfo(skb)->frags[i].size = len-pos;
                           skb_shinfo(skb)->nr_frags++;
                      }
                      k++;
                 } else {
                      skb_shinfo(skb)->nr_frags++;
                 }
                 pos += size;
            }
            skb_shinfo(skb1)->nr_frags = k;
       }
  }
```

[net/core/skbuff.c]

10.7. skb_checksum

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

There are 3 different functions to perform the checksum. One that simply performs the checksum, one that performs the checksum while copying the data (it is required by efficiency reasons) and one that takes advantage from the hardware support offered by the card. Partial checksums over contiguous data areas are performed using the `csum_partial` function, and then added together using the `csum_block_add` function. `skb_checksum` computes in this way the checksum for the skbuff also in the case the skbuff is fragmented in an array of pages or a list of skbuffs.

[net/core/skbuff.c]

```
unsigned int skb_checksum(const struct sk_buff *skb, int offset,
                  int len, unsigned int csum)
{
```

```
        int start = skb_headlen(skb);
        int i, copy = start - offset;
        int pos = 0;

        /* Checksum header. */
        if (copy > 0) {
                if (copy > len)
                        copy = len;
                csum = csum_partial(skb->data + offset, copy, csum);
                if ((len -= copy) == 0)
                        return csum;
                offset += copy;
                pos  = 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) {
                        unsigned int csum2;
                        u8 *vaddr;
                        skb_frag_t *frag = &skb_shinfo(skb)->frags[i];

                        if (copy > len)
                                copy = len;
                        vaddr = kmap_skb_frag(frag);
                        csum2 = csum_partial(vaddr + frag->page_offset +
                                             offset - start, copy, 0);
                        kunmap_skb_frag(vaddr);
                        csum = csum_block_add(csum, csum2, pos);
                        if (!(len -= copy))
                                return csum;
                        offset += copy;
                        pos    += 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) {
                                unsigned int csum2;
                                if (copy > len)
                                        copy = len;
```

```
                        csum2 = skb_checksum(list, offset - start,
                                             copy, 0);
                        csum = csum_block_add(csum, csum2, pos);
                        if ((len -= copy) == 0)
                             return csum;
                        offset += copy;
                        pos    += copy;
                   }
                   start = end;
              }
         }
         if (len)
              BUG();

         return csum;
    }
```

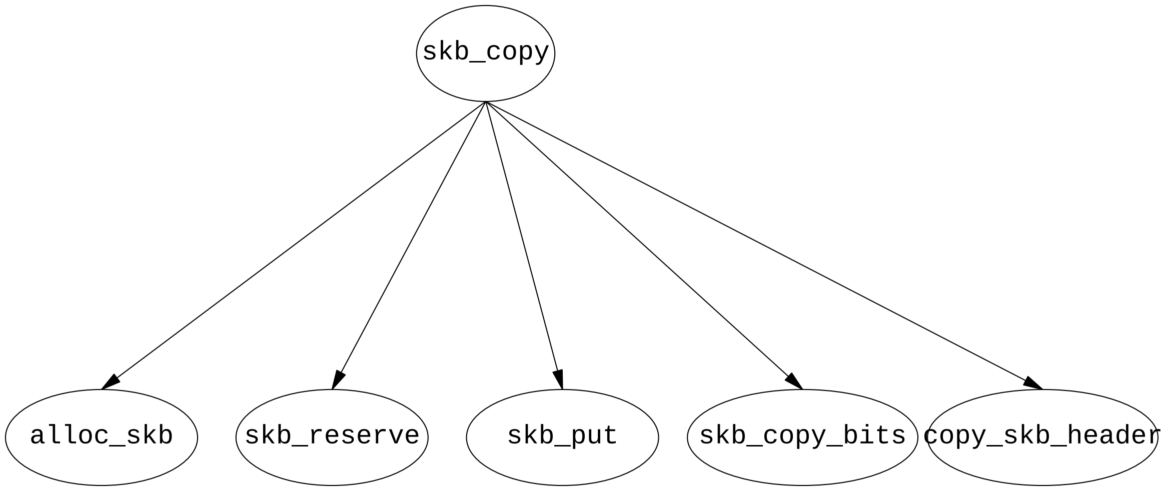
[net/core/skbuff.c]

`skb_copy_and_csum_bits` performs at the same time the function of the `skb_copy_bits` and the `skb_checksum` functions.
In case the device supports checksumming in hardware (CHECKSUM_HW flag), the `skb_copy_and_csum_dev` function is used.

10.8. Copying functions

10.8.1. skb_copy()

Makes a complete copy of an skbuff header and its data. It eventually converts a nonlinear skbuff to a linear one. The headroom of the old skbuff is computed and reserved also in the new one (the variable for it is inappropriately called headerlen). If the data doesn't need to be modified then the use of pskb_copy that uses the reference count mechanisms for the nonlinear part is recommended.



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. The data area size is equal to the size of the linear part of the old skbuff plus the size of data in the nonlinear parts (external pages and rest of the chain of skbuffs). If it is not possible to allocate such an skbuff it returns NULL. A headroom equal to the one in the old skbuff is reserved in the new skbuff calling skb_reserve. It sets the tail pointer in the new skbuff to accomodate for the total length of the old skbuff(calling skb_put). It copies the checksum and the ip_summed flag. Then it calls the skb_copy_bits function to copy everything from the very beginning of the old skbuff (from skb->head not skb->data) to the end of the data. This means that it copies also the headroom. The second argument of skb_copy_bits is the displacement from the standard skb->data pointer from where to begin the copy. Finally it copies the skbuff header, and returns a pointer to the new skbuff.

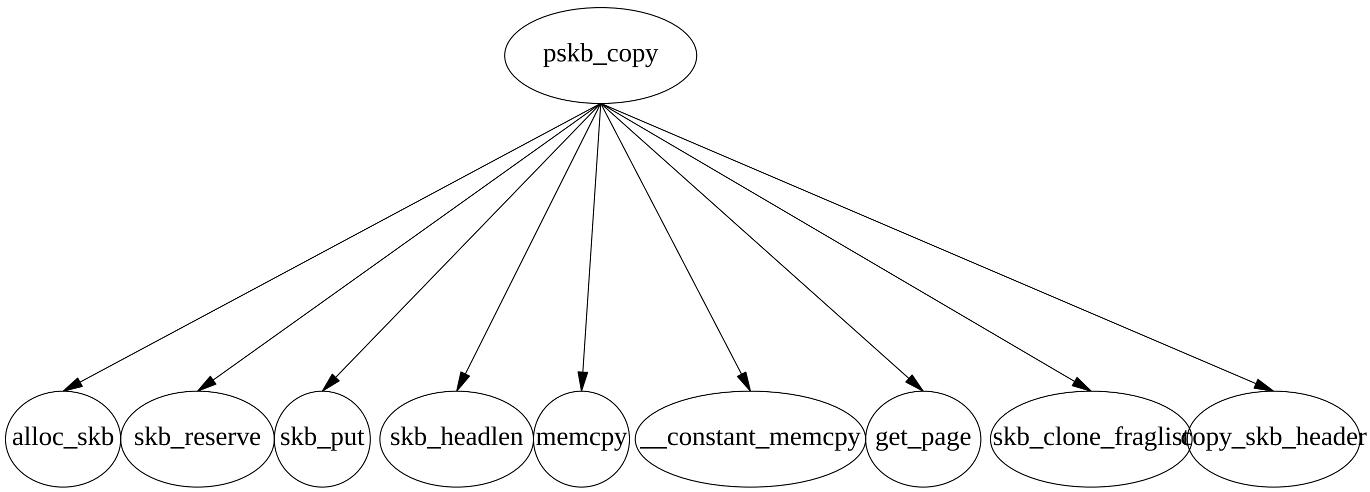
net/core/skbuff.c

```
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;
}
```

net/core/skbuff.c

10.8.2. pskb_copy()

File : `net/core/skbuff.c`



It allocates an skbuff having the same linear size of the old one (because of this it doesnt use skb->len, but skb->end - skb->head). It reserves the same headroom available in the old skbuff moving the data and tail pointers (headroom is the space between the beginning of the linear part of the skbuff and the skb->data pointer). Then it adjusts the tail pointer to accomodate for the data in the linear part of the skbuff (skb_headlen is the size of such data). It copies headlen bytes of data from the old skbuff to the new one. Copies checksum, ip_summed flag,data_len (the size of the data not in the linear part), len from old to new. Runs through the array of pages, if it exists, copies fragment descriptors from the old to the new skbuff. For each page it increments the usage count without copying it. Then if it is an skbuff chain, it copies the frag_list pointer, and it goes through the list of skbuffs and increases the usage count (skb_clone_fraglist increases the skb->users counter on each of the following skbuffs). Finally it copies the skb header (copy_skb_header). And it returns a pointer to the new skbuff head.

net/core/skbuff.c

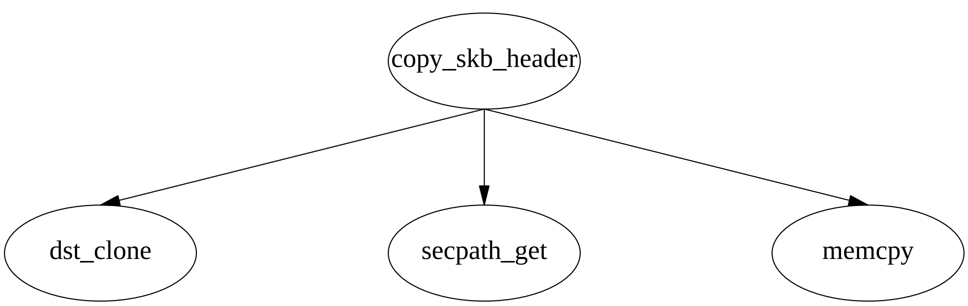
```
struct sk_buff *pskb_copy(struct sk_buff *skb, int gfp_mask)
{
     /*
      *   Allocate the copy buffer
      */
     struct sk_buff *n = alloc_skb(skb->end - skb->head, gfp_mask);

     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;
}
```

net/core/skbuff.c

10.8.3. 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 the standard pointers of the skbuff are already initialized to the new data area (head, data, tail, end). Most of the remaining fields are just copied over from the old skbuff. But the list pointer (pointer to queue on which the skbuff is linked) is set to NULL, as sk (pointer to the socket the skbuff belongs). As the forwarding mechanism uses a reference count, the dst field cannot be simply copied, but this is done through the dst_clone function that updates the reference counter atomically. The security mechanism also uses a reference count, and so instead of simply copy the sp pointer, it calls the secpath_get function that atomically increases the reference counter associated with the path used. It 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 copies over with memcpy the cb control buffer. The skbuff destructor is set to NULL. Two of the fields used by netfilter, that use the reference count mechanism, nfct (connection track) and nf_bridge, are copied using functions that atomically increase the reference counters : nf_conntrack_get and nf_bridge_get. Finally it atomically sets the number of users of the new skbuff to 1.

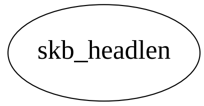
net/core/skbuff.c

```
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
#ifdef CONFIG_BRIDGE_NETFILTER
     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

10.8.4. skb_headlen

File : [include/linux/skbuff.h]

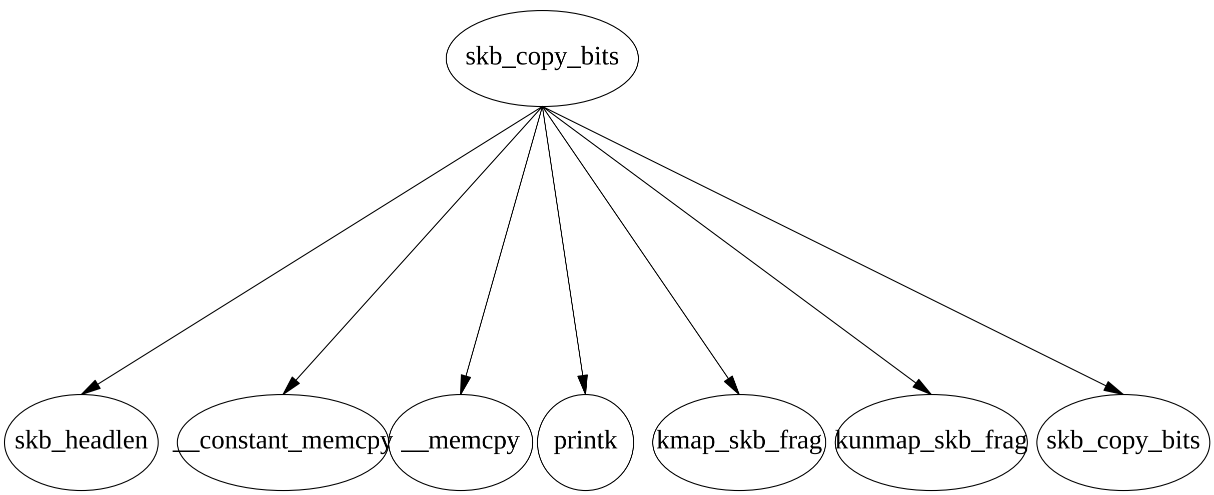


This function returns the number of data bytes in the linear part of the first skbuff. It is equal to (skb->tail - skb->data).

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

10.8.5. skb_copy_bits

File : [net/core/skbuff.c]



This function copies bytes of data from an skbuff to another area of memory and doing so it eventually linearize nonlinear skbuffs. The first argument is a pointer to an skbuff header from which data area the data should be copied, the third argument is a pointer to an area of memory where the data should be copied to. 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, it is the number of bytes to skip from data pointer position. But it can be also negative and in that case it represents the bytes to copy from the headroom in addition to those starting from the data pointer. The len argument is the number of bytes that should be copied. This function is able to treat paged skbuffs and chains of skbuffs. Initially it copies from the linear part of the first skbuff. It copies copy = headlen - offset bytes. It has code to copy all the external pages with memcpy. If a page is in high memory, then it is necessary to find a free slot in the low memory area dedicated to mapping, where to map it (kmap_skb_frag), the page can then be copied and unmapped again (kunmap_skb_frag). If it is an skbuff chain, then the following skbuffs are copied calling itself recursively for each skbuff in the chain (frag_list).

[net/core/skbuff.c]

```
 832    /* 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]

10.8.6. skb_copy_expand

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

It makes a copy of a skbuff and its data expanding the free data areas at the beginning and at the end of it. A new skbuff is allocated with `alloc_skb` of size equal to the size of the data in the old skbuff plus the required `newheadroom` and `newtailroom.` Then the pointers of the skbuff are displaced with `skb_reserve` of the requested new headroom, and the skbuff and its data are copied over with `skb_copy_bits` (if the skbuff was fragmented it is linearized). The TSO info are copied too.

[net/core/skbuff.c]

```
struct sk_buff *skb_copy_expand(const struct sk_buff *skb,
                                int newheadroom, int newtailroom, int gfp_mask)
{
        /*
         *   Allocate the copy buffer
         */
        struct sk_buff *n = alloc_skb(newheadroom + skb->len + newtailroom,
                                      gfp_mask);
        int head_copy_len, head_copy_off;

        if (!n)
                return NULL;

        skb_reserve(n, newheadroom);

        /* Set the tail pointer and length */
        skb_put(n, skb->len);

        head_copy_len = skb_headroom(skb);
```

```
        head_copy_off = 0;
        if (newheadroom <= head_copy_len)
             head_copy_len = newheadroom;
        else
             head_copy_off = newheadroom - head_copy_len;

        /* Copy the linear header and data. */
        if (skb_copy_bits(skb, -head_copy_len, n->head + head_copy_off,
                    skb->len + head_copy_len))
             BUG();

        copy_skb_header(n, skb);
        skb_shinfo(n)->tso_size = skb_shinfo(skb)->tso_size;
        skb_shinfo(n)->tso_segs = skb_shinfo(skb)->tso_segs;

        return n;
   }
```

[net/core/skbuff.c]

10.8.7. skb_copy_datagram, skb_copy_datagram_iovec

File : `[net/core/datagram.c]`
The first function is simply a call to the second after having set up an iovec with only one entry. The second copies a datagram to an existing iovec: if the skbuff is fragmented (in a list out of IP defragmentation or in an array of pages) the data is linearized (as it can be an iovec .. it can be also splitted !). This function is called in tcp_input.c and also in udp.c and raw.c to deliver the data to the user.

[net/core/datagram.c]

```
   /*
    *   Copy a datagram to a linear buffer.
    */
   int skb_copy_datagram(const struct sk_buff *skb, int offset, char *to, int size)
   {
        struct iovec iov = {
             .iov_base = to,
             .iov_len =size,
        };

        return skb_copy_datagram_iovec(skb, offset, &iov, size);
   }

   /**
    *   skb_copy_datagram_iovec - Copy a datagram to an iovec.
    *   @skb - buffer to copy
    *   @offset - offset in the buffer to start copying from
    *   @iovec - io vector to copy to
    *   @len - amount of data to copy from buffer to iovec
    *
    *   Note: the iovec is modified during the copy.
    */
   int skb_copy_datagram_iovec(const struct sk_buff *skb, int offset,
```

```
                    struct iovec *to, int len)
{
     int start = skb_headlen(skb);
     int i, copy = start - offset;

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

     /* Copy paged appendix. Hmm... why does this look so complicated? */
     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) {
               int err;
               u8  *vaddr;
               skb_frag_t *frag = &skb_shinfo(skb)->frags[i];
               struct page *page = frag->page;

               if (copy > len)
                    copy = len;
               vaddr = kmap(page);
               err = memcpy_toiovec(to, vaddr + frag->page_offset +
                              offset - start, copy);
               kunmap(page);
               if (err)
                    goto fault;
               if (!(len -= copy))
                    return 0;
               offset += 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_datagram_iovec(list,
                                           offset - start,
                                           to, copy))
                             goto fault;
                        if ((len -= copy) == 0)
                             return 0;
                        offset += copy;
                   }
                   start = end;
              }
         }
         if (!len)
              return 0;

    fault:
         return -EFAULT;
    }
```

[net/core/datagram.c]

10.9. Cloning and sharing

10.9.1. clone_fraglist()

File : `net/core/skbuff.c`

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);
    }
```

10.9.2. skb_shared, skb_clone, skb_share_check, skb_unshare

skb_shared atomically tests if the skbuff is shared, that is there is more than 1 user of the same header and data.

net/core/skbuff.c

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

net/core/skbuff.c

It is used to panic if a shared skbuff is passed to functions that do not expect it or in the case a user wants to modify the header, to check if cloning is necessary. skb_cloned checks if an skbuff has been cloned (multiple headers have pointers to the same data area).

net/core/skbuff.c

```
static inline int skb_cloned(const struct sk_buff *skb)
{
     return skb->cloned && atomic_read(&skb_shinfo(skb)->dataref) != 1;
}
```

net/core/skbuff.c

skb_share_check checks if an skbuff is shared, in that case it is cloned, the users counter is decremented on the old skbuff and a pointer to the new is returned. The new skbuff has skb->users == 1 and skb->cloned == 1, the old skbuff has skb->cloned == 1 and skb->users decremented by one.

net/core/skbuff.c

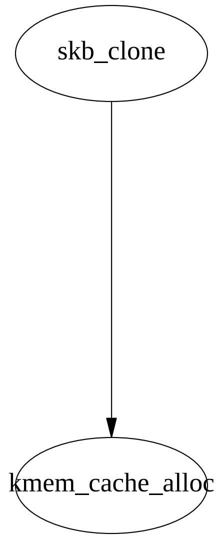
```
static inline struct sk_buff *skb_share_check(struct sk_buff *skb, int pri)
{
     might_sleep_if(pri & __GFP_WAIT);
     if (skb_shared(skb)) {
          struct sk_buff *nskb = skb_clone(skb, pri);
          kfree_skb(skb);
          skb = nskb;
     }
     return skb;
}
```

net/core/skbuff.c

skb_unshare checks if an skbuff is cloned. In such a case makes a complete copy of the skbuff (header and data), decrements skb->users and returns a pointer to the new skbuff.

10.9.3. 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
#ifdef CONFIG_BRIDGE_NETFILTER
	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

10.9.4. skb_get()

File : `include/linux/skbuff.h`

It atomically increments the number of users of the skbuff returning back a pointer to it. The complete skbuff (header and data area) is shared in this way and if it needs to be modified, then a copy should be done.

```
/**
 *   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;
}

```

10.9.5. Cloning and udp multicasting/broadcasting

Udp multicast/broadcast data have to be delivered to each listener. Each listener has a separate receive queue, so an skbuff containing a multicast/broadcast udp packet has to be cloned for each listener, and enqueued on its own socket receive queue (the enqueuing of the skbuff makes necessary the adjustment of their next and prev pointers and so each listener needs its unique copy of the header). The udp_v4_mcast_deliver function runs through the list of listeners (using udp_v4_mcast_next) and it enqueues a clone of the skbuff in the socket receive queue of it (udp_queue_rcv_skb).

include/linux/skbuff.h

```
static int udp_v4_mcast_deliver(struct sk_buff *skb, struct udphdr *uh,
                      u32 saddr, u32 daddr)
{
     struct sock *sk;
     int dif;

     read_lock(&udp_hash_lock);
     sk = sk_head(&udp_hash[ntohs(uh->dest) & (UDP_HTABLE_SIZE - 1)]);
     dif = skb->dev->ifindex;
     sk = udp_v4_mcast_next(sk, uh->dest, daddr, uh->source, saddr, dif);
     if (sk) {
          struct sock *sknext = NULL;

          do {
               struct sk_buff *skb1 = skb;

               sknext = udp_v4_mcast_next(sk_next(sk), uh->dest, daddr,
                                    uh->source, saddr, dif);
               if(sknext)
                    skb1 = skb_clone(skb, GFP_ATOMIC);
```

```

                        if(skb1) {
                            int ret = udp_queue_rcv_skb(sk, skb1);
                            if (ret > 0)
                                /* we should probably re-process instead
                                 * of dropping packets here. */
                                kfree_skb(skb1);
                        }
                        sk = sknext;
                    } while(sknext);
                } else
                    kfree_skb(skb);
                read_unlock(&udp_hash_lock);
                return 0;
            }
```

include/linux/skbuff.h

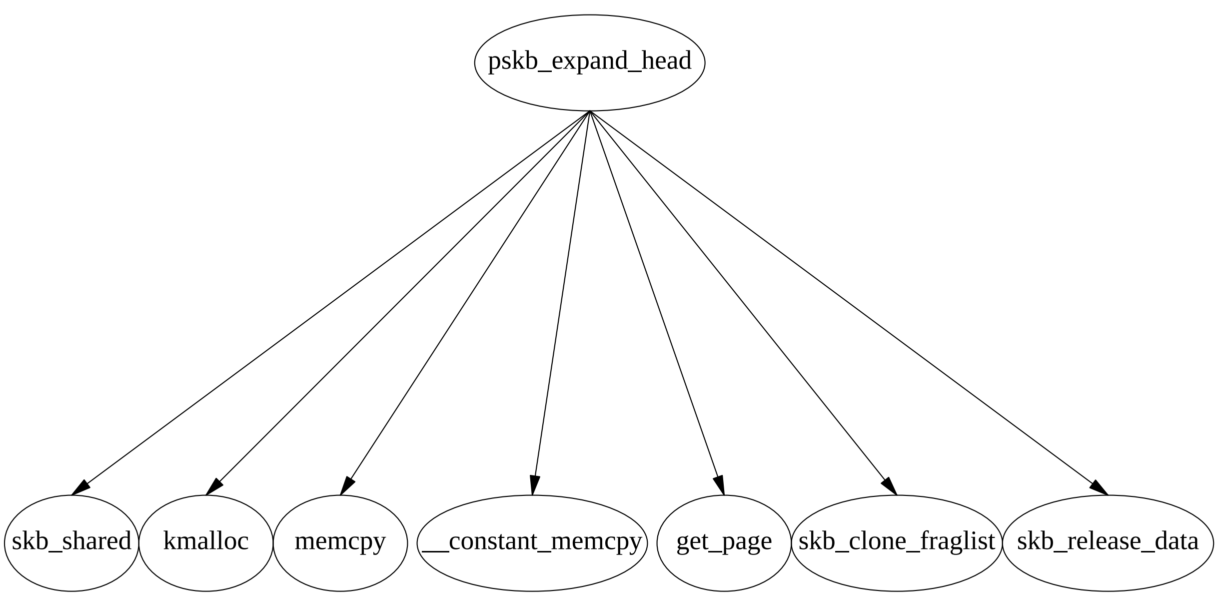
10.9.6. Sharing of fragment list

When you copy a list fragmented skbuff you copy the header and you just increase the users counter on the following skbuffs, because usually you dont need to perform any operation on the remaining skbuffs (you add/strip headers only on the first skbuff). This seems to be the only sensible use of skbuff sharing.

10.10. Miscellaneous functions

10.10.1. pskb_expand_head

File : `include/linux/skbuff.h`



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. The other skbuffs in the skbuff chain are shared through calls to skb_get that increments the dataref field.

include/linux/skbuff.h

```
/**
 *   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;
}
```

include/linux/skbuff.h

10.10.2. skb_realloc_headroom

File : `include/linux/skbuff.h`
It makes a private copy of the skb checking that there is at least `headroom` bytes of free space at the beginning of the buffer. The copy is made with `pskb_copy` if there is already enough space, otherwise it is done first by cloning the skbuff with `skb_clone` and then copying expanding the header with `pskb_expand_head`.

include/linux/skbuff.h

```
/* Make private copy of skb with writable head and some headroom */

struct sk_buff *skb_realloc_headroom(struct sk_buff *skb, unsigned int headroom)
{
        struct sk_buff *skb2;
        int delta = headroom - skb_headroom(skb);

        if (delta <= 0)
                skb2 = pskb_copy(skb, GFP_ATOMIC);
        else {
                skb2 = skb_clone(skb, GFP_ATOMIC);
                if (skb2 && pskb_expand_head(skb2, SKB_DATA_ALIGN(delta), 0,
                                             GFP_ATOMIC)) {
                        kfree_skb(skb2);
                        skb2 = NULL;
                }
        }
        return skb2;
}

```

include/linux/skbuff.h

10.10.3. skb_pad

File : `include/linux/skbuff.h`
In some cases it is possible that network cards using DMA transfer data beyond the buffer end because of size of the unit of transfer. It is therefore necessary sometimes to fill some space after the buffer with zeroes. This function can be expensive because if there is not enough tailroom in the skbuff it has to copy and expand it. This function is called to be assured there are `pad` bytes of zeroes after the data. This function is only called by the skb_padto function.

include/linux/skbuff.h

```
struct sk_buff *skb_pad(struct sk_buff *skb, int pad)
{
     struct sk_buff *nskb;

     /* If the skbuff is non linear tailroom is always zero.. */
     if (skb_tailroom(skb) >= pad) {
          memset(skb->data+skb->len, 0, pad);
          return skb;
     }

     nskb = skb_copy_expand(skb, skb_headroom(skb), skb_tailroom(skb) + pad, GFP_ATOMIC);
     kfree_skb(skb);
     if (nskb)
          memset(nskb->data+nskb->len, 0, pad);
     return nskb;
}
```

include/linux/skbuff.h

10.10.4. skb_padto

File : `include/linux/skbuff.h`
This function uses the `skb_pad` function to pad the data in the skbuff with zeroes up to `len` bytes.
It is used to pad ethernet frames to the minimum size prescribed by the standard. 60 bytes excluded the frame check sequence (FCS) that is computed by the card (4 bytes). Inside ethernet drivers it is normally used this way : skb_padto(skb,`skb`= where ETH_ZLEN is 60.

include/linux/skbuff.h

```
static inline struct sk_buff *skb_padto(struct sk_buff *skb, unsigned int len)
{
     unsigned int size = skb->len;
     if (likely(size >= len))
          return skb;
     return skb_pad(skb, len-size);
}
```

include/linux/skbuff.h

10.10.5. skb_set_owner_r,skb_set_owner_w

These two function are called when the owner socket of an skbuff is known. During output the owner socket is known since the beginning and thus they are called inside the allocation functions (sock_alloc_skb,sock_alloc_pskb ..). In input packets need to be processed to discover the owner socket.

include/net/sock.h

```
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);
}

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);
}
```

include/net/sock.h

10.10.6. get_page

This function is part of the memory management functions. It simply atomically increases the reference counter on the page, relying on the standard COW (copy on write) kernel mechanism to copy the page when it is needed. If the kernel was compiled with huge page support (CONFIG_HUGETLB_PAGE) then this function is defined with a further check. If the page is a large/huge page (named compound in the code because it is seen also as a set of small pages), then the lru.next field points to the head (the first small page of the large page). It is the refcount of that head page that is incremented in that case.

include/linux/mm.h

```
static inline void get_page(struct page *page)
{
        if (PageCompound(page))
                page = (struct page *)page->lru.next;
        atomic_inc(&page->count);
}
```

include/linux/mm.h

---------------------- comment : it 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 of Cox[2] 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

----

10.11. IRIX

With some devices a bus adapter is interposed between the device and the system bus, in this case the bus adapter can translate between bus addresses used by the devices (for instance PCI cards on a PCI bus) and system addresses. 64 bits addresses are divided accroding to their most significant 2 bits into 4 segments : 3 of them access memory throught the TLB and so are mapped (user process space xkuseg=00, supervisor mode space xksseg=01, kernel virtual space xkseg=00)

and one accesses memory through physical addresses (xkphys=10).
Using dma maps with pci devices

Allocate a map (on the pci address space): pccio_dmamap_alloc(device identifier:bus and slot, device descriptor,bytes,falgs)

(You can get a default device descriptor calling device_desc_default_get with the device identifier) You activate the map calling pciio_dmamap_addr() or pciio_dmamap_list(). These operations set up a translation table on the bus adapter that converts bus to/from system addresses.

[3, 4, 5, 6, 7, 8, 9, 10, 11] # end

Table of Contents

References

1. K.C.Knowlton, “A Fast Storage Allocator,” *Communications of ACM,* Vol.8, October 1965 (1965).

2. Alan Cox, "Network Buffers and Memory Management," *Linux Journal*, 29 (September 29, 1996).

3. Richard Stevens, *TCP/IP Illustrated vol 1, The protocols,* p. Addison Wesley (1994).

4. G. Wright, R. Stevens, *TCP/IP Illustrated vol. 2, The implementation,* p. Addison Wesley (1995).

5. Daniel P. Bovet, Marco Cesati, *Understanding the Linux kernel, 2nd edition,* p. O'Reilly (December 2002).

6. Marshall Kirk McKusick, Keith Bostic , Michael J. Karels, John S. Quarterman, *The Design and Implementation of the 4.4 BSD Operating System, 2nd edition,* p. Addison Wesley (1996).

7. Mel Gorman, *Understanding the Linux Virtual Memory Manager,* p. available on the web (July, 2003).

8. Jeff Bonwick (Sun Microsystems), "The Slab Allocator: An Object-Caching Kernel Memory Allocator," *USENIX Summer Technical Conference* (1994).

9. Brad Fitzgibbons (gatech), *The Linux Slab Allocator,* p. on the web (October 30, 2000).

10. Mika Karlstedt (University of Helsinki), *Linux 2.4 memory management,* p. on the web (2002).

11. B.Goodheart,J.Cox,J.R.Mashey, *The Magic Garden Explained: The internals of Unix System V Release 4,* p. Prentice Hall (1994).