IP Layer Input Packet Processing

The receive functions of the IP layer include:

IP header validation;
IP header option processing;
Routing of the input packet to the proper transport.

However, to accomplish this limited mission a surprisingly large amount of processing is performed. A significant amount of this processing involves managing the layout and ownership of the *sk_buff*. This the primary concern of the *ip_rcv()* function which is defined in net/ipv4/ip_input.c.

```
380 /*
381  *      Main IP Receive routine.
382  */
383 int ip_rcv(struct sk_buff *skb, struct net_device
                  *dev, struct packet_type *pt)
384 {
385     struct iphdr *iph;
```

When the interface is in promiscuous mode, any *sk_buff*, not directed to this host, is discarded without any processing. The packet type was set to PACKET_OTHERHOST by *net_rx_action()* if the packet was neither broadcast nor multicast and the destination MAC address was not the same as the MAC address carried by the *struct netdevice* representing the interface upon which the packet arrived. Operating an in interface in promiscuous mode on a *true broadcast medium* is the only legitimate cause of this situation.

```
387     /*   When the interface is in promisc. mode,
             drop all the crap that it receives, do
             not try to analyse it.
389      */
390     if (skb->pkt_type == PACKET_OTHERHOST)
391        goto drop;
392
```

These are the counters that SNMP uses.

```
393     IP_INC_STATS_BH(IpInReceives);
```

Dealing with shared *skb's*

The skb_share_check() function determines if the sk_buff is shared. If so, the sk_buff is cloned, the original is freed, and a pointer to the clone is returned. If it is not shared, a pointer to the original is returned. If the sk_buff is shared, but the attempt to clone it fails, NULL is returned. Buffers may be shared at this point if multiple handlers for a specific packet type have been registered and have indicated that they understand shared skb's. The fact that the shared skb's become unshared at such a low level in the stack calls their usefulness into some question.

```
395     if ((skb = skb_share_check(skb, GFP_ATOMIC)) == NULL)
396        goto out;
```

The skb_share_check() function, defined in include/linux/skbuff.h, determines if the buffer is shared, and if so attempts to clone it. Freeing the original buffer decrements the use count, but actually frees the buffer only when the use count becomes 0. A cloned buffer necessarily has a use count exceeding one, and so call to kfree_skb() simply decrements it.

```
343 static inline struct sk_buff *skb_share_check(struct
        sk_buff *skb, int pri)
344 {
345     if (skb_shared(skb)) {
346          struct sk_buff *nskb;
347          nskb = skb_clone(skb, pri);
348          kfree_skb(skb);
349          return nskb;
350     }
351     return skb;
352 }
```

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

```
324 static inline int skb_shared(struct sk_buff *skb)
325 {
326     return (atomic_read(&skb->users) != 1);
327 }
```

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

```
347 struct sk_buff *skb_clone(struct sk_buff *skb, int
        gfp_mask)
348 {
349     struct sk_buff *n;
350
```

Each CPU maintains a pool of free struct sk_buff' headers. If the pool is empty then it is necessary to allocate from the cache managed by the slab allocator.

```
351     n = skb_head_from_pool();
352     if (!n) {
353         n = kmem_cache_alloc(skbuff_head_cache,
                gfp_mask);
354         if (!n)
355             return NULL;
356     }
```

The skb_head_from_pool() function detaches and returns the first sk_buff header in the list or returns NULL if the list is empty.

```
112 static __inline__ struct sk_buff
                *skb_head_from_pool(void)
113 {
114     struct sk_buff_head *list =
            &skb_head_pool[smp_processor_id()].list;

116     if (skb_queue_len(list)) {
117         struct sk_buff *skb;
118         unsigned long flags;
119
120         local_irq_save(flags);
121         skb = __skb_dequeue(list);
122         local_irq_restore(flags);
123         return skb;
124     }
125     return NULL;
126 }
```

Instead of using memcpy(), skb_clone() copies a single structure element at a time.

```
358 #define C(x) n->x = skb->x
359
360     n->next = n->prev = NULL;
361     n->list = NULL;
362     n->sk = NULL;
363     C(stamp);
364     C(dev);
365     C(h);
366     C(nh);
367     C(mac);
368     C(dst);
369     dst_clone(n->dst);
370     memcpy(n->cb, skb->cb, sizeof(skb->cb));
371     C(len);
372     C(data_len);
373     C(csum);
374     n->cloned = 1;
375     C(pkt_type);
376     C(ip_summed);
377     C(priority);
378     atomic_set(&n->users, 1);
379     C(protocol);
380     C(security);
381     C(truesize);
382     C(head);
383     C(data);
384     C(tail);
385     C(end);
```

```
386     n->destructor = NULL;
387 #ifdef CONFIG_NETFILTER
388     C(nfmark);
389     C(nfcache);
390     C(nfct);
391 #ifdef CONFIG_NETFILTER_DEBUG
392     C(nf_debug);
393 #endif
394 #endif /*CONFIG_NETFILTER*/
395 #if defined(CONFIG_HIPPI)
396     C(private);
397 #endif
398 #ifdef CONFIG_NET_SCHED
399     C(tc_index);
400 #endif
401
402     atomic_inc(&(skb_shinfo(skb)->dataref));
403     skb->cloned = 1;
404 #ifdef CONFIG_NETFILTER
405     nf_conntrack_get(skb->nfct);
406 #endif
```

Finally, skb_clone() returns a pointer to the cloned sk_buff.

```
407     return n;
408 }
```

IP Header Validation..

Back in ip_rcv, the pskb_may_pull() is called. It ensures that IP header is entirely present in kmalloc'd area. It moves (pulls) the IP header from unmapped page fragments into the kmalloc'd area if required. We are not aware of any device drivers that create this ugly situation, but rectifying it requires an unbelievably tedious 10 pages of code which will not be examined here.

```
398     if (!pskb_may_pull(skb, sizeof(struct iphdr)))
399          goto inhdr_error;
```

After ensuring that IP header is properly resident in the kmalloc'd area, IP header validation is performed. Validation includes ensuring that:

the length of the datagram is at least the 20 byte length of an IP header;
the IP version is 4;
the checksum is satisfactory;
the packet length reported in the IP header is consistent with the length reported in the struct skbuff.

```
401     iph = skb->nh.iph;
402
```

Verify that header length and version number are satisfactory.

```
414     if (iph->ihl < 5 || iph->version != 4)
415          goto inhdr_error;
```

Pull IP any header options into kmalloc'd area of the sk_buff. This call is a return visit to the 10 pages of torture previously referenced.

```
417     if (!pskb_may_pull(skb, iph->ihl*4))
418          goto inhdr_error;
```

Verify the IP header checksum using ip_fast_csum . The header pointer must be reloaded here because the pskb_may_pull() operation may have rebuilt the whole skb.

```
420     iph = skb->nh.iph;
421
422     if (ip_fast_csum((u8 *)iph, iph->ihl) != 0)
423          goto inhdr_error;
```

Verify that the length reported in iph->totlen is acceptable. If the length reported in iph->tot_len is greater than that reported in skb->len, or if it is less than the length of IP header, then there is a definite problem.

```
425     {
426          u32 len = ntohs(iph->tot_len);
427          if (skb->len < len || len < (iph->ihl<<2))
428               goto inhdr_error;
```

However, it is legal for skb->len to exceed iph->tot_len. For example, it is typical for ethernet drivers to allocate buffers of MTU size. When this occurs, skb->len is adjusted downward to become consistent with iph->tot_len.

```
430          /*   Our transport medium may have padded the
                  buffer out. Now we know it is IP we can
                  trim to the true length of the frame.
                  Note this now means skb->len holds
                  ntohs(iph->tot_len).
433           */
434          if (skb->len > len) {
435               __pskb_trim(skb, len);
```

skb->ip_summed check? ... why only when sk buff is trimmed?? .. is it because we trimmed it...and checksum is not valid any more ..

```
436               if (skb->ip_summed == CHECKSUM_HW)
437                    skb->ip_summed = CHECKSUM_NONE;
438          }
439     }
440
```

Finally, the packet is passed to the netfilter facilty. The "okfn", ip_rcv_finish, is called if the netfilter finds the packet to be acceptable.

```
441     return NF_HOOK(PF_INET, NF_IP_PRE_ROUTING, skb,
442               dev, NULL, ip_rcv_finish);
```

In the event of any error, the sk_buff is discarded.

```
444 inhdr_error:
445     IP_INC_STATS_BH(IpInHdrErrors);
446 drop:
447     kfree_skb(skb);
448 out:
449     return NET_RX_DROP;
450 }
```

Trimming the skb to match the header length

__pskb_trim is defined in include/linux/skbuff.h.

```
946 static inline int __pskb_trim(struct sk_buff *skb,
                               unsigned int len)
947 {
```

If there is no data in unmapped page fragments, the trim operation, simply updates the values of the tail and len members. Otherwise ___pskb_trim gets called.

```
948     if (!skb->data_len) {
949         skb->len = len;
950         skb->tail = skb->data+len;
951         return 0;
952     } else {
953         return ___pskb_trim(skb, len, 1);
954     }
955 }
```

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

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

739 int ___pskb_trim(struct sk_buff *skb, unsigned int
                   len, int realloc)
740 {
```

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

```
741     int offset = skb_headlen(skb);
742     int nfrags = skb_shinfo(skb)->nr_frags;
743     int i;
744
```

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

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

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

```
746          int end = offset + skb_shinfo(skb)
                                    ->frags[i].size;
747          if (end > len) {
748               if (skb_cloned(skb)) {
749                    if (!realloc)
750                         BUG();
751                    if (!pskb_expand_head(skb, 0, 0,
                              GFP_ATOMIC))
752                         return -ENOMEM;
753               }
```

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

```
754               if (len <= offset) {
755                    put_page(skb_shinfo(skb)
                                    ->frags[i].page);
756                    skb_shinfo(skb)->nr_frags--;
757               } else {
758                    skb_shinfo(skb)->frags[i].size
                                     = len-offset;
759               }
760          }
```

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

```
761          offset = end;
762     }
```

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

```
764     if (offset < len) {
765          skb->data_len -= skb->len - len;
766          skb->len = len;
767     }
```

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

```
               else {
768            if (len <= skb_headlen(skb)) {
769                 skb->len = len;
770                 skb->data_len = 0;
771                 skb->tail = skb->data + len;
772                 if (skb_shinfo(skb)->frag_list &&
                                        !skb_cloned(skb))
773                      skb_drop_fraglist(skb);
774            }
```

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

```
               else {
775                 skb->data_len -= skb->len - len;
776                 skb->len = len;
777            }
778     }
779
780     return 0;
781 }
```