Received Packet Processing at the *dev* Layer

Device driver processing

When a packet has been received by the NIC, and the DMA transfer to an *sk_buff* has completed, and interrupt is generated. The device driver's received packet interrupt handler

- calls *eth_type_trans()* to establish the packet type (e.g. 0x800 for ETH_P_IP) and remove the *MAC* header and then either:

- calls *netif_rx()* as shown in this extract from the 3c59x driver.

```
2373    outw(RxDiscard, ioaddr + EL3_CMD); /* Pop top Rx pkt. */
2374    skb->protocol = eth_type_trans(skb, dev);
2375    netif_rx(skb);
```

- or calls *netif_rx_schedule()* as shown in this extract from the *e100* driver
- drivers that use this interface must provide their own "poll" function whose mission will be described later.

The address of the driver's poll function is stored in the *struct net_device* at device initialization time:

```
2575    netdev->poll = e100_poll;
```

```
1976    if (likely(netif_rx_schedule_prep(netdev))) {
1977       e100_disable_irq(nic);
1978       __netif_rx_schedule(netdev);
1979    }
1980
1981    return IRQ_HANDLED;
```

Queuing the packet with *netif_rx()*

The *netif_rx()* function is defined in *net/core/dev.c.* It runs in the context of the hardware interrupt that signaled the completion of the DMA transfer. It serves as a front end to *__netif_rx_schedule()* for the older drivers. Its mission is to queue the *sk_buff* for processing by the *dev* layer. The buffer may, however, be dropped during processing for congestion control. After queuing the packet, *netif_rx()* invokes *netif_rx_schedule* which raises the NET_RX_SOFTIRQ.

The *softnet* data structure

We had previously seen the *softnet_data* structure used as a respository for packets that had completed transmission and for *net_devices* that needed to be redriven.

It is also used to hold incoming packets on per-cpu queues so that no locking is needed. Note the *ugly inconsistency* in the way the two queues are defined.

The *softnet_data* array, defined in *include/linux/netdevice.h.*, consists of a *struct softnet_data* for each CPU.

```
604/*
605 * Incoming packets are placed on per-cpu queues so that
606 * no locking is needed.
607 */
608
609 struct softnet_data
610 {
611     struct net_device       *output_queue;
612     struct sk_buff_head     input_pkt_queue;
613     struct list_head        poll_list;
614     struct sk_buff          *completion_queue;
615
616     struct net_device       backlog_dev;    /* Sorry. 8) */
617 #ifdef CONFIG_NET_DMA
618     struct dma_chan         *net_dma;
619 #endif
620 };
```

The *backlog_dev* is an ugly hack used in the transition between "new" and "old" style device drivers.

Input congestion management

These are the old congestion management parameters.

```
1073 int netdev_max_backlog = 300;
1074 /* These numbers are selected based on intuition and some
1075  * experimentatiom, if you have more scientific way
1076  * please go ahead and fix things.
1077  */
1078 int no_cong_thresh = 10;
1079 int no_cong = 20;
1080 int lo_cong = 100;
1081 int mod_cong = 290;
1082
```

Modern congestion management

Now there is a binary drop/no drop threshold and a somewhat saner strategy for dealing with load balancing.

```
1548 int netdev_max_backlog = 1000;  /* drop threshold */
1549 int netdev_budget = 300;        // max pkts per activation
1550 int weight_p = 64;              /* old backlog weight */
```

The thresholds still exist in the *comments* today, but were never used at all to the best of my knowledge. Now congestion management is binary for old style devices.

When a backlog limit is reached, all new incoming packets are simply dropped.

New style devices don't use the *input_packet* queue at all. If they aren't polled frequently enough, their receive rings will fill up and when that happens *they will simply stop receiving.*

The *backlog_dev*

Each CPU has bogus *backlog_dev* device that servse as a proxy for *any* real device with having an "old style" driver. Only a few elements of the *net_device* structure that pertain to arriving packet management are used.

These are initialized in *net_dev_init()*

```
3491 /*
3492  *      This is called single threaded during boot, so no need
3493  *      to take the rtnl semaphore.
3494  */
3495 static int __init net_dev_init(void)
3496 {

3519    /*
3520     * Initialise the packet receive queues.
3521     */
3522
3523    for_each_possible_cpu(i) {
3524       struct softnet_data *queue;
3525
3526       queue = &per_cpu(softnet_data, i);
3527       skb_queue_head_init(&queue->input_pkt_queue);
3528       queue->completion_queue = NULL;
3529       INIT_LIST_HEAD(&queue->poll_list);
3530       set_bit(__LINK_STATE_START, &queue->backlog_dev.state);
3531       queue->backlog_dev.weight = weight_p;
3532       queue->backlog_dev.poll = process_backlog;
3533       atomic_set(&queue->backlog_dev.refcnt, 1);
3534    }
```

The *net_dev_init()* function also registers the network softirqs.

```
3536    netdev_dma_register();
3538    dev_boot_phase = 0;
3540    open_softirq(NET_TX_SOFTIRQ, net_tx_action, NULL);
3541    open_softirq(NET_RX_SOFTIRQ, net_rx_action, NULL);
```

The *netif_rx* function

The function is the *dev* layer receive entry point used by *old style* device drivers. Note that the old style congestion management parameters survive even though they were never used at all. (Maybe Linus invented them?)

```
1554
1555 /**
1556  * netif_rx -  post buffer to the network code
1557  * @skb: buffer to post
1558  *
1564  * return values:
1565  * NET_RX_SUCCESS (no congestion)
1566  * NET_RX_CN_LOW   (low congestion)
1567  * NET_RX_CN_MOD   (moderate congestion)
1568  * NET_RX_CN_HIGH  (high congestion)
1569  * NET_RX_DROP     (packet was dropped)
1570  *
1571  */
```

For super high speed devices polling is actually more efficient than being interrupt driven. Some of the advantages of polling may be offset to a degree by the use of large frame sizes and interrupt coalescing. We will not address the *netpoll()* facility.

```
1573 int netif_rx(struct sk_buff *skb)
1574 {
1575    struct softnet_data *queue;
1576    unsigned long flags;
1577
1578    /* if netpoll wants it, pretend we never saw it */
1579    if (netpoll_rx(skb))
1580       return NET_RX_DROP;
```

If the device driver has not already time stamped the packet, it is done here.

```
1582    if (!skb->tstamp.off_sec)
1583       net_timestamp(skb);
```

The local variable *queue* is set to point to the *struct sofnet_data* for this cpu.

```
1584
1585    /*
1586     * The code is rearranged so that the path is the most
1587     * short when CPU is congested, but is still operating.
1588     */
1589    local_irq_save(flags);
1590    queue = &__get_cpu_var(softnet_data);
1591
1592    __get_cpu_var(netdev_rx_stat).total++;
```

Testing for full queue conditions

The length of the input packet queue is compared against its maximum backlog. If the queue is full, the *sk_buff* is discarded. The value of *netdev_max_backlog* is now declared to be 1000 packets in net/core/dev.c. It used to be the case that the *throttle* flag was tested to see if the packet should be dropped. The throttle flag was set when *netdev_max_backlog* was reached. After a CPU was throttled a complete draining of the queue had to occur before unthrottling occured. Now it is a simple one-shot test.

```
1593    if (queue->input_pkt_queue.qlen <= netdev_max_backlog) {
```

The following compound *if* first tests to see if the input queue is *not* empty. If the queue is not empty, the fast path is taken. The call to *dev_hold()* increments the reference counter of the *net_device* to reflect the fact that the *sk_buff* holds a reference to it.

```
1594        if (queue->input_pkt_queue.qlen) {
1595 enqueue:
1596            dev_hold(skb->dev);
1597            __skb_queue_tail(&queue->input_pkt_queue, skb);
1598            local_irq_restore(flags);
1599            return NET_RX_SUCCESS;
1600        }
```

Arrival here means the queue was empty. The *netif_rx_action* function will be called to place the bogo device *backlog_dev* on the backlog queue and to schedule the NET_RX_SOFTIRQ. This is followed by a backward jump to queue the packet.

```
1602        netif_rx_schedule(&queue->backlog_dev);
1603        goto enqueue;
1604    }
```

If the queue is full, the packet is dropped here.

```
1606    __get_cpu_var(netdev_rx_stat).dropped++;
1607    local_irq_restore(flags);
1608
1609    kfree_skb(skb);
1610    return NET_RX_DROP;
1611 }
1612
```

Scheduling the *net_rx_softirq*

The real action occurs in *__netif_rx_schedule()*. All of these wrappers just ensure that

- the net_device is started and
- the net_device is not presently already scheduled.

Having a single *net_device* serviced by two instances of *net_rx_action()* on different CPUs at the same time would be catastrophic error!

When called from *netif_rx*, the only device passed in is the bogus *backlog_dev,* but when called by a device driver the *actual net_device on which the packet arrived will be passed.*

```
851 static inline void netif_rx_schedule(struct net_device *dev)
852 {
853    if (netif_rx_schedule_prep(dev))
854       __netif_rx_schedule(dev);
855 }
```

Serialization of the RX processing

The e100.c device driver produced by Intel for the e10/100/1000 family of devices doesn't use *netif_rx_schedule()*. It directly calls *netif_rx_schedule_prep()* and *netif_rx_schedule()*

```
838 static inline int netif_rx_schedule_prep(
                        struct net_device *dev)
839 {
840    return netif_running(dev) && __netif_rx_schedule_prep(dev);
841 }

663 static inline int netif_running(const struct net_device *dev)
664 {
665    return test_bit(__LINK_STATE_START, &dev->state);
666 }
```

Here an atomic test and set is done. This ensures that a specific device can be scheduled on *at most one CPU at a time.* It is possible for *net_rx_action()* to run concurrently on multiple CPUs, but the concurrent versions will provide RX service to different devices.

Recall that similar processing occurred in *devxmit()* but there the bit used was *__LINK_STATE_SCHED.* Hence it would be possible for a single device to simultaneously receive TX service on one CPU and RX service on another.

```
832 static inline int __netif_rx_schedule_prep(
                              struct net_device *dev)
833 {
834  return !test_and_set_bit(__LINK_STATE_RX_SCHED, &dev->state);
835 }
```

The *__netif_rx_schedule()* function

This function adds either the bogus *backlog_dev* or a real *net_device* onto the *poll_list* of the *softnet* data structure. The call to *dev_hold()* increments the reference count for the *net_device*. This will be eventually dropped after the *net_device* is serviced by the *softirq*.

```
1118 void __netif_rx_schedule(struct net_device *dev)
1119 {
1120    unsigned long flags;
1121
1122    local_irq_save(flags);
1123    dev_hold(dev);
1124    list_add_tail(&dev->poll_list,
                &__get_cpu_var(softnet_data).poll_list);
```

The quota is a packet count measure. Each time the device is scheduled its quota is boosted. The weight for the *e100* is 16, but the weight for the *backlog_dev* is 64. The *backlog_dev* serves as a proxy for all old drivers, but there is also one *backlog_dev* per CPU.

```
  168 #define E100_NAPI_WEIGHT   16
```

```
1125    if (dev->quota < 0)
1126       dev->quota += dev->weight;
1127    else
1128       dev->quota = dev->weight;
```

The call to *__raise_softirq_irqoff()* causes the kernel daemon in whose context the softirq runs to be awakened.

```
1129    __raise_softirq_irqoff(NET_RX_SOFTIRQ);
1130    local_irq_restore(flags);
1131 }
```

Softirqs

In early versions of Linux processing of received packets took place in the context of what was called a *bottom half*. The *softirq* mechanism, which was designed to replace the *bottom half* was introduced in kernel 2.4. The primary advantage of the *softirq* mechanism is that separate instances of a specific *softirq* may run concurrently on different processors. Bottom halves were permitted to run only on one CPU at a time.

There are a maximum of 32 softirqs and creating new ones is strongly discouraged. Note that your timer handler is invoked in the context of TIMER_SOFTIRQ. Hence creating lots of timers and/or doing a lot of processing in timer handlers has a negative impact on network performance.

> *PLEASE, avoid to allocate new softirqs, if you need not _really_ high frequency threaded job scheduling. For almost all the purposes tasklets are more than enough. F.e. all serial device BHs et al. should be converted to tasklets, not to softirqs.*

```
214
215 enum
216 {
217    HI_SOFTIRQ=0,
218    TIMER_SOFTIRQ,
219    NET_TX_SOFTIRQ,
220    NET_RX_SOFTIRQ,
221    BLOCK_SOFTIRQ,
222    TASKLET_SOFTIRQ
223 };
```

Registering a *softirq*

Recall that the function *net_dev_init()* which runs at boot time registered two softirq handlers.

```
3539
3540        open_softirq(NET_TX_SOFTIRQ, net_tx_action, NULL);
3541        open_softirq(NET_RX_SOFTIRQ, net_rx_action, NULL);
```

The parameters include:

- the numeric id which serves as a *lower is better* priority
- the address of the handler
- an optional pointer

These values are saved in the *softirq_vec* array of 32 elements using the *nr* parameter as an array index.

```
326 void open_softirq(int nr, void (*action)(struct
                                softirq_action*), void *data)
327 {
328         softirq_vec[nr].data = data;
329         softirq_vec[nr].action = action;
330 }
```

Raising a softirq

An array of structures of type *irq_cpustat_t* is indexed by CPU ID. The *_softirq_pending* word is a map in which a 1 bit means the soft_irq is pending. The *__raise_softirq_irqoff(nr)* function *just sets the proper bit* to indicate that the requested softirq is pending

```
 7 typedef struct {
 8        unsigned int __softirq_pending;
 9        unsigned long idle_timestamp;
10        unsigned int __nmi_count;       /* arch dependent */
11        unsigned int apic_timer_irqs;   /* arch dependent */
12 } ____cacheline_aligned irq_cpustat_t;
13
14 DECLARE_PER_CPU(irq_cpustat_t, irq_stat);
```

If the caller knows it is running in the context of hardware irq (as device *rx* handlers are), then it is slightly more efficient to just call *__raise_softirq_irqoff()* directly. On each return from hard or soft irq processing the kernel will check for pending softirqs. If not in the context of hard or soft interrupt, the wakeup occurs at the end of interrupt processing.

```
295 /*
296  * This function must run with irqs disabled!
297  */
298 inline fastcall void raise_softirq_irqoff(unsigned int nr)
299 {
300    __raise_softirq_irqoff(nr);
301
302    /*
303     * If we're in an interrupt or softirq, we're done
304     * (this also catches softirq-disabled code). We will
305     * actually run the softirq once we return from
306     * the irq or softirq.
307     *
308     * Otherwise we wake up ksoftirqd to make sure we
309     * schedule the softirq soon.
310     */
311     if (!in_interrupt())
312           wakeup_softirqd();
313 }
```

The __raise_softirq_irqoff() function

```
 238 #define __raise_softirq_irqoff(nr) \
      do { or_softirq_pending(1UL << (nr)); } while (0)

#define or_softirq_pending(x)  or_pda(__softirq_pending, (x))
```

Waking up the softirqd.

Each CPU runs its own instance of the softirq daemon. The address of its *task_struct* is maintained in the *ksoftirqd* pointer of the per CPU data structure.

```
 55 static inline void wakeup_softirqd(void)
 56 {
 57   /* Interrupts are disabled: no need to stop preemption */
 58     struct task_struct *tsk = __get_cpu_var(ksoftirqd);
 59
 60     if (tsk && tsk->state != TASK_RUNNING)
 61             wake_up_process(tsk);
 62 }
```

Running the *softirq's*

The *ksoftirqd* function runs in the context of the per-cpu kernel daemons. They peridically wake up and invoke *do_softirq()* to process the softirqs that are pending on *this* CPU.

```
470 static int ksoftirqd(void * __bind_cpu)
471 {
472     set_user_nice(current, 19);
473     current->flags |= PF_NOFREEZE;
474
475     set_current_state(TASK_INTERRUPTIBLE);
476
```

This is basically a *do forever* loop.

```
477     while (!kthread_should_stop()) {
```

The deamon goes to sleep here if nothing is pending.

```
478         preempt_disable();
479         if (!local_softirq_pending()) {
480             preempt_enable_no_resched();
481             schedule();   // sleep wakeup occurs here
482             preempt_disable();
483         }
484
```

The deamon wakes up and processes pending softirqs.

```
485         __set_current_state(TASK_RUNNING);
486
487         while (local_softirq_pending()) {
488         /* Preempt disable stops cpu going offline.
489            If already offline, we'll be on wrong CPU:
490            don't process */
491            if (cpu_is_offline((long)__bind_cpu))
492               goto wait_to_die;
493            do_softirq();
494            preempt_enable_no_resched();
495            cond_resched();
496            preempt_disable();
497         }
498         preempt_enable();
499         set_current_state(TASK_INTERRUPTIBLE);
500     }
501    __set_current_state(TASK_RUNNING);
502    return 0;
503


504 wait_to_die:
505        preempt_enable();
506        /* Wait for kthread_stop */
507        set_current_state(TASK_INTERRUPTIBLE);
508        while (!kthread_should_stop()) {
509                schedule();
510                set_current_state(TASK_INTERRUPTIBLE);
511        }
512        __set_current_state(TASK_RUNNING);
513        return 0;
514}
```

The *do_softirq()* function

The *do_softirq* function is now an assembly language hybrid that performs context management functions and invokes *__do_softirq*. The call to *local_irq_save()* disables interrupts on this CPU. Thus the *__do_softirq()* function is always invoked with hardware irqs disabled.

```
180 asmlinkage void do_softirq(void)
181 {
182    unsigned long flags;
183    struct thread_info *curctx;
184    union irq_ctx *irqctx;
185    u32 *isp;
186
187    if (in_interrupt())
188        return;
189
190    local_irq_save(flags);
191
192    if (local_softirq_pending()) {
193        curctx = current_thread_info();
194        irqctx = softirq_ctx[smp_processor_id()];
195        irqctx->tinfo.task = curctx->task;
196        irqctx->tinfo.previous_esp = current_stack_pointer;
197
198     /* build the stack frame on the softirq stack */
199        isp = (u32*) ((char*)irqctx + sizeof(*irqctx));
200
201        asm volatile(
202                        "       xchgl   %%ebx,%%esp     \n"
203                        "       call    __do_softirq    \n"
204                        "       movl    %%ebx,%%esp     \n"
205                        : "=b"(isp)
206                        : "0"(isp)
207                        : "memory", "cc", "edx", "ecx", "eax"
208       );
209    /*
210     * Shouldnt happen, we returned above if in_interrupt():
211     */
212         WARN_ON_ONCE(softirq_count());
213    }
214
215    local_irq_restore(flags);
216 }
```

The *__do_softirq()* funciton

The *__do_softirq()* function is defined in *kernel/softirq.c*. It invokes the appropriate action handler for each softirq raised.

```
#define MAX_SOFTIRQ_RESTART 10

206 asmlinkage void __do_softirq(void)
207 {
208     struct softirq_action *h;
209     __u32 pending;
210     int max_restart = MAX_SOFTIRQ_RESTART;
211     int cpu;
```

The variable *pending* is a bit mask that indicates which of the possible 32 soft irqs are presently pending. It is set to *irq_stat[cpu].__softirq_pending*.

```
213     pending = local_softirq_pending();
214     account_system_vtime(current);
215
216    __local_bh_disable((unsigned  long)
                              __builtin_return_address(0));
217     trace_softirq_enter();
```

The first operations after the *restart* tag is to reset *irq_stat[cpu].__softirq_pending* to 0 and then enable hardware interrupts on this processor.

```
219     cpu = smp_processor_id();
220 restart:
221     /* Reset the pending bitmask before enabling irqs */
222     set_softirq_pending(0);
223
224     local_irq_enable();
225
```

Processing the *softirq_vec*

"h" is set to point to first element in *softirq_vec* array. Elements are indexed by *softirq* number and contain handler and data pointers.

```
226     h = softirq_vec;
227
228     do {
```

Softirqs are checked in order of their priority (HI_SOFTIRQ, NET_TX_SOFTIRQ ...)
and the respective function handler is called. In the case of NET_RX_SOFTIRQ, the handler is *net_rx_action()*.

```
229         if (pending & 1) {
230             h->action(h);
231             rcu_bh_qsctr_inc(cpu)
```

"*h*" is now set to point to next element in *softirq_vec* array. The value of *pending* is shifted right by one so that the current candidate bit is in the low order position.

```
233         h++;
234         pending >>= 1;
235     } while (pending);
236
237     local_irq_disable();
```

If new *softirqs* (including those handled above) have been raised, they are handled as well as long as the maximum number of 10 iterations is not reached.

```
239     pending = local_softirq_pending();
240     if (pending && --max_restart)
241         goto restart;
242
```

If the maximum number of iterations is reached and pending is not zero then the remainder must be handled on the next scheduling cycle.

```
243     if (pending)
244        wakeup_softirqd();
245
246     trace_softirq_exit();
247
248     account_system_vtime(current);
249     _local_bh_enable();
250 }
```

Received packet handling in the *softirq*.

Delivery of a packet to the transport layer is performed in the context of the *NET_RX_SOFTIRQ* handler, *net_rx_action()* which is invoked by the *__do_softirq()* function which was just described. The call to *do_softirq()* was triggered when *__netif_rx_schedule()* executed the line of code shown:

```
1129     __raise_softirq_irqoff(NET_RX_SOFTIRQ);
```

The *net_rx_action()* function resides in *net/core/dev.c* and was previously shown to have been installed as the handler for the NET_RX_SOFTIRQ. As might be expected, its mission is to consume packets from the queue *input_packet_queue* or the *device_driver* and then to pass them on to the proper handler.

```
1905 static void net_rx_action(struct softirq_action *h)
1906 {
```

A unique structure of type *struct softnet_data* is associated with each CPU for the purpose of managing input and output queues at the interface between the protocols and the device driver. Here *queue* is initalized to point to *softnet_data* structure for this CPU. In the 2.4 code, the variable *budget* was spelled *bugdet!* The value limits the number of packets that can be processed in a single run of the softirq.

```
int netdev_budget = 300;  // max packets to be consumed
```

```
1907     struct softnet_data *queue =
                                &__get_cpu_var(softnet_data);
1908     unsigned long start_time = jiffies;
1909     int budget = netdev_budget;
1910     void *have;
1911
1912     local_irq_disable();
```

The *poll_list* is a list of *net_devices* that have pending packets. It may also contain the bogus *backlog_dev* which serves as a proxy for all of the old style device drivers. Only old style devices use the input_packet_queue. New style devices drivers are responsible for maintaining there own queues, or simply consuming from the packet ready side of the Rx ring.

Note that "list stealing" is not performed here. The reasons for this are that (1) it is not guaranteed that it will be possible to process the entire list in one activation and (2) a device that consumes its quota of packets is moved to the tail of the list.

```
1914    while (!list_empty(&queue->poll_list)) {
1915       struct net_device *dev;
1916
```

In addition to being limited to 300 packets processing is also limited to 2 jiffies.

```
1917       if (budget <= 0 || jiffies - start_time > 1)
1918          goto softnet_break;
1919
1920       local_irq_enable();
1921
```

The next *net_device* in the list and the lock on its poll function are obtained.

```
1922       dev = list_entry(queue->poll_list.next,
1923              struct net_device, poll_list);
1924       have = netpoll_poll_lock(dev);
1925
```

If the device quota is not exceeded, then the devices poll function is invoked. The poll function is responsible for decrementing budget. It returns 0 if the queue of pending packets is completely drained and -1 if the quota is exhausted. The *dev->poll* function of *backlog_dev* is *process_packets*. For the *e100* driver it is *e100_poll.*

```
1926        if (dev->quota <= 0 || dev->poll(dev, &budget)) {
```

Falling into this code means *dev->quota* is now <= 0. The device is moved to the end of the line and the quota is renewed if it is negative.

```
1927            netpoll_poll_unlock(have);
1928            local_irq_disable();
1929            list_move_tail(&dev->poll_list, &queue->poll_list);
1930            if (dev->quota < 0)
1931               dev->quota += dev->weight;
1932            else
1933               dev->quota = dev->weight;
```

Device successfully drained its input queue.

```
1934        } else {
1935            netpoll_poll_unlock(have);
1936            dev_put(dev);
1937            local_irq_disable();
1938        }
1939    }

----- dma device handling --------

1954    local_irq_enable();
1955    return;
1956
```

If we ran out of time or packets, the *softirq* is raised so that packet consumption can continue on the next scheduling cycle.

```
1957 softnet_break:
1958    __get_cpu_var(netdev_rx_stat).time_squeeze++;
1959    __raise_softirq_irqoff(NET_RX_SOFTIRQ);
1960    goto out;
1961 }
```

The *process_backlog* function

This is the *poll* function for the old style drivers that use the *netif_rx()* interface. Modern drivers provide their own, but their functionality should be similar.

```
1858 static int process_backlog(struct net_device *backlog_dev,
                                  int *budget)
1859 {
1860    int work = 0;
1861    int quota = min(backlog_dev->quota, *budget);
1862    struct softnet_data *queue = &__get_cpu_var(softnet_data);
1863    unsigned long start_time = jiffies;
1864
1865    backlog_dev->weight = weight_p;  // value is 64 use is ???
1866    for (;;) {
1867       struct sk_buff *skb;
1868       struct net_device *dev;
1869
1870       local_irq_disable();
1871       skb = __skb_dequeue(&queue->input_pkt_queue);
```

If the queue is empty the job is complete.

```
1872       if (!skb)
1873          goto job_done;
1874       local_irq_enable();
1875
```

The device driver *must* set skb->dev.

```
1876       dev = skb->dev;
1877
```

Passing the packet to the network layer occurs here. New style drivers will call this function from their poll routine.

```
1878        netif_receive_skb(skb);
1879
```

The value of *work* is the number of packets processed Note that a separate *jiffie* timer is run for each invocation of a poll function such as *process_backlog()*

```
1880        dev_put(dev);
1881
1882        work++;
1883
1884        if (work >= quota || jiffies - start_time > 1)
1885           break;
1886
1887    }
1888
```

Arrival here implies we ran out of time or quota...

```
1889    backlog_dev->quota -= work;
1890    *budget -= work;
1891    return -1;
```

The input queue was successfully drained. After updating the device quota, it is deleted from the poll list and scheduling is re-enabled.

```
1892
1893 job_done:
1894    backlog_dev->quota -= work;
1895    *budget -= work;
1896
1897    list_del(&backlog_dev->poll_list);
1898    smp_mb__before_clear_bit();
1899    netif_poll_enable(backlog_dev);
1901    local_irq_enable();
1902    return 0;
1903 }
```

```
 900 static inline void netif_poll_enable(struct net_device *dev)
 901 {
 902     clear_bit(__LINK_STATE_RX_SCHED, &dev->state);
 903 }
```

The *netif_receive_skb* function

Device drivers that provide their own "poll" functions now call this function to deliver packets to network layer handlers.

```
1763 int netif_receive_skb(struct sk_buff *skb)
1764 {
1765    struct packet_type *ptype, *pt_prev;
1766    struct net_device *orig_dev;
1767    int ret = NET_RX_DROP;
1768    unsigned short type;
1769
1770    /* if we've gotten here through NAPI, check netpoll */
1771    if (skb->dev->poll && netpoll_rx(skb))
1772       return NET_RX_DROP;
1773
1774    if (!skb->tstamp.off_sec)
1775       net_timestamp(skb);
1776
1777    if (!skb->input_dev)
1778       skb->input_dev = skb->dev;
1779
```

The *net_device* pointer (which was set by the device driver) is potentially adjusted here.
The *skb_bond()* function is defined in net/core/dev.c. It assigns the *sk_buff* to the master device for present device if such exists.

```
1780    orig_dev = skb_bond(skb);
1781
1782    if (!orig_dev)
1783       return NET_RX_DROP;
1784
1785    __get_cpu_var(netdev_rx_stat).total++;
1786
```

This assumes that the device set *skb->data* to point just beyond the MAC header. The network and transport layer header pointers are set to the same spot.

```
1787    skb->h.raw = skb->nh.raw = skb->data;
1788    skb->mac_len = skb->nh.raw - skb->mac.raw;
1789
```

We have seen this problem in *udp_rcv()*. If a packet is to be delivered to multiple recipients it must be shared or cloned. But we don't know whether to increment *skb->users* until we know there will be another recipient.

```
1790    pt_prev = NULL;
1791
1792    rcu_read_lock();
1793
1794 #ifdef CONFIG_NET_CLS_ACT
1795    if (skb->tc_verd & TC_NCLS) {
1796       skb->tc_verd = CLR_TC_NCLS(skb->tc_verd);
1797       goto ncls;
1798    }
1799 #endif
1800
```

Delivery to *ptype_all* handlers

Protocols which wish to receive all incoming packets are linked into a list pointed to by *ptype_all*. These protocols have type ETH_P_ALL and are processed before considering the protocols that consume only a specific packet type. Most of these are assumed to be "read only" type applications and so buffers are shared rather than cloned.

```
1801    list_for_each_entry_rcu(ptype, &ptype_all, list) {
```

Even though every packet handler in this chain says it wants to see all packets, it can also say that it wants to limit the packets to those received on a specific device. If *ptype->dev* is NULL, then any device is acceptable.

The oddball use of *pt_prev* is done because of the necessity of sharing an skb. It should be necessary to share if and only if there is more than one protocol interested. The actual sharing occurs in *deliver_skb()*. Note that *pt_prev* was initially set to NULL so no actual deliver occurs for the first *ptype* found.

```
1802        if (!ptype->dev || ptype->dev == skb->dev) {
1803            if (pt_prev)
1804                ret = deliver_skb(skb, pt_prev, orig_dev);
1805            pt_prev = ptype;
1806        }
1807    }
1808
```

We won't worry about NET_CLS_ACT / diverters / bridges

```
1809 #ifdef CONFIG_NET_CLS_ACT
1810    if (pt_prev) {
1811       ret = deliver_skb(skb, pt_prev, orig_dev);
1812       pt_prev = NULL; /* noone else should process this
                              after*/
1813    } else {
1814       skb->tc_verd = SET_TC_OK2MUNGE(skb->tc_verd);
1815    }
1816
1817    ret = ing_filter(skb);
1818
1819    if (ret == TC_ACT_SHOT || (ret == TC_ACT_STOLEN)) {
1820       kfree_skb(skb);
1821       goto out;
1822    }
1823
1824    skb->tc_verd = 0;
1825 ncls:
1826 #endif
1827
1828    handle_diverter(skb);
1829
1830    if (handle_bridge(&skb, &pt_prev, &ret, orig_dev))
1831       goto out;
1832
```

Delivery to specific handlers

This is the point at which specific handlers (such as ip_rcv) that were registered with *dev_add_pack()* are invoked. The 16 bit packet type (ETH_P_IP == 0x800) is used as a hash key.

```
1833    type = skb->protocol;
1834    list_for_each_entry_rcu(ptype,
           &ptype_base[ntohs(type)&15], list) {

1835       if (ptype->type == type &&
1836           (!ptype->dev || ptype->dev == skb->dev)) {
1837          if (pt_prev)
1838             ret = deliver_skb(skb, pt_prev, orig_dev);
1839          pt_prev = ptype;
1840       }
1841    }
1842
```

Delivery to the last handler

To deliver the "unshared" copy the *rcv* handler for the network layer is invoked directly.

```
1843    if (pt_prev) {
1844       ret = pt_prev->func(skb, skb->dev, pt_prev, orig_dev);
1845    } else {
1846       kfree_skb(skb);
1847       /* Jamal, now you will not able to escape explaining
1848        * me how you were going to use this. :-)
1849        */
1850       ret = NET_RX_DROP;
1851    }
1852
1853 out:
1854    rcu_read_unlock();
1855    return ret;
1856 }
```

The *deliver_skb* function

This function implicitly shares the *sk_buff* and then invokes the protocol handler pointed to by the *struct packet_type*.

```
1689 static __inline__ int deliver_skb(struct sk_buff *skb,
1690                struct packet_type *pt_prev,
1691                struct net_device *orig_dev)
1692 {
1693    atomic_inc(&skb->users);
1694    return pt_prev->func(skb, skb->dev, pt_prev, orig_dev);
1695 }
1696
```