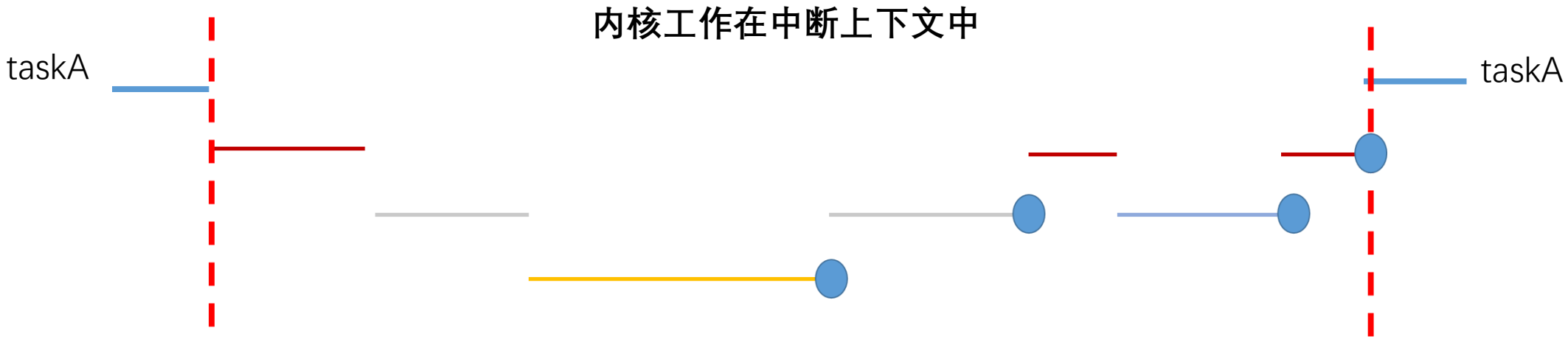
中断

中断机制

- CPU正在执行一个任务 taskA。收到一个中断请求。
  - CPU 暂停执行taskA。 去处理中断
  - 中断处理结束后
    - 调度：中断处理程序有唤醒一个更重要的进程，执行那个进程
    - 恢复执行 taskA

- 借由中断，CPU 用户态 → 核心态
  IRET 核心态 → 用户态

例：　4次中断，4次IRET后恢复执行被中断的taskA ——



最先发生的中断 ——，最后一个执行完毕

中断嵌套

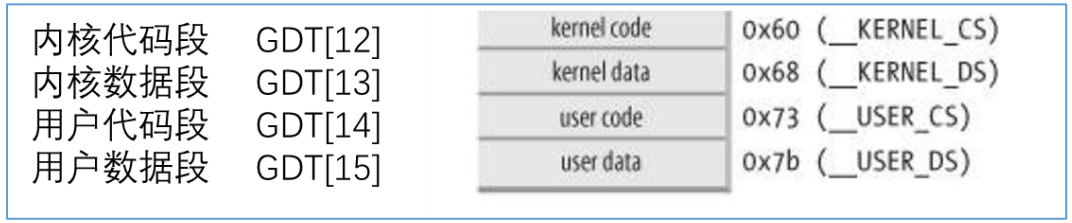
- taskA 有3种情况：
- 1、执行某个APP；　2、执行系统调用；　3、执行中断处理程序

- 情况 3，我们说系统工作在中断上下文。其它情况，我们说，系统没有工作在中断上下文。
  Linux响应中断，这个叫做中断嵌套。

- Linux系统，不为中断设置优先级。这也就是说，Linux认为所有的中断优先级相等。
- 只有2种类型的中断处理程序
  - 关中断运行。新出现的中断请求，开中断后响应。
  - 开中断运行。如果，中断处理程序开中断运行，运行期间，系统响应一切中断请求。

中断向量

- 80X86芯片支持256种中断。每种中断对应一种CPU必须理睬的事件。 称 整数0~255 为中断向量。
- 内存里有一张中断向量表（IDT），这是256个元素的数组，登记着256种中断的处理方式。数组的元素8字节，其中最重要的是2个字段，这是中断处理程序的入口地址：0x60 和 1个偏移量。



- 每颗CPU，有IDTR。所有IDTR指向内存中的 IDT。
- CPU响应中断时，会用中断向量做下标搜IDT，取
  - 0x60 → CS ； 偏移量 → EIP 这是CPU硬件在调用中断处理程序

详细的中断响应过程

- CPU正在执行一个任务 taskA。收到一个中断请求。
  - CPU 暂停执行taskA。 去处理中断
  - 中断处理结束后
    - 中断处理程序有唤醒一个更重要的进程，执行那个进程
    - 恢复执行 taskA

- CS、EIP、SS、ESP、EFLAGS 压 现运行进程核心栈。

- 用中断向量查IDT。分别将值0x60和中断处理程序的入口地址赋CS和EIP。 效果是：

  1、CPU切换至核心态运行　2、CPU开始执行内核代码段中的中断处理程序。

  另外，CPU响应中断，硬件自动关中断。

- 中断处理程序跑 ~　1、push　EAX~通用寄存器，在核心栈里

  2、中断处理过程。。。

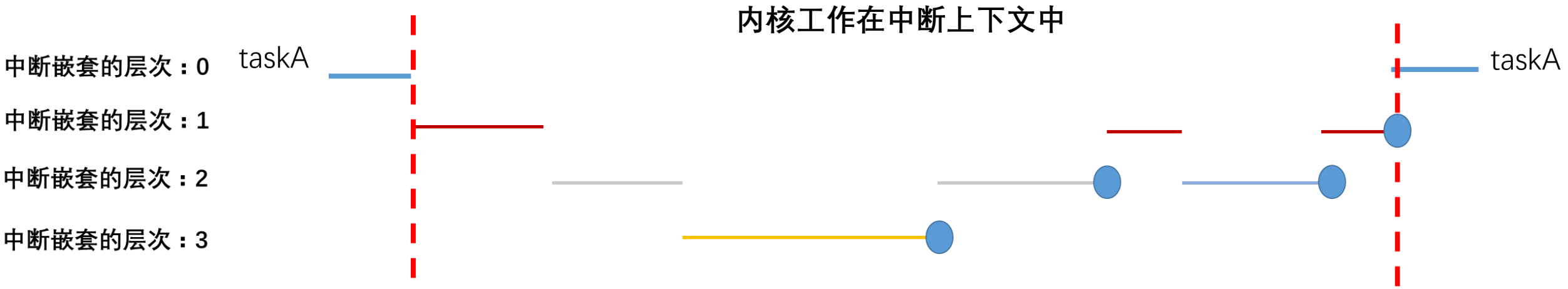
  （所以，若非中断处理程序主动STI，中断处理程序工作在关中断环境下，是否开中断，何时开中断，主动权在程序员手里）。

中断返回

- CPU正在执行一个任务 taskA。收到一个中断请求。
  - CPU 暂停执行taskA。 去处理中断
  - 中断处理结束后
    - 中断处理程序有唤醒一个更重要的进程，执行那个进程
    - 恢复执行 taskA

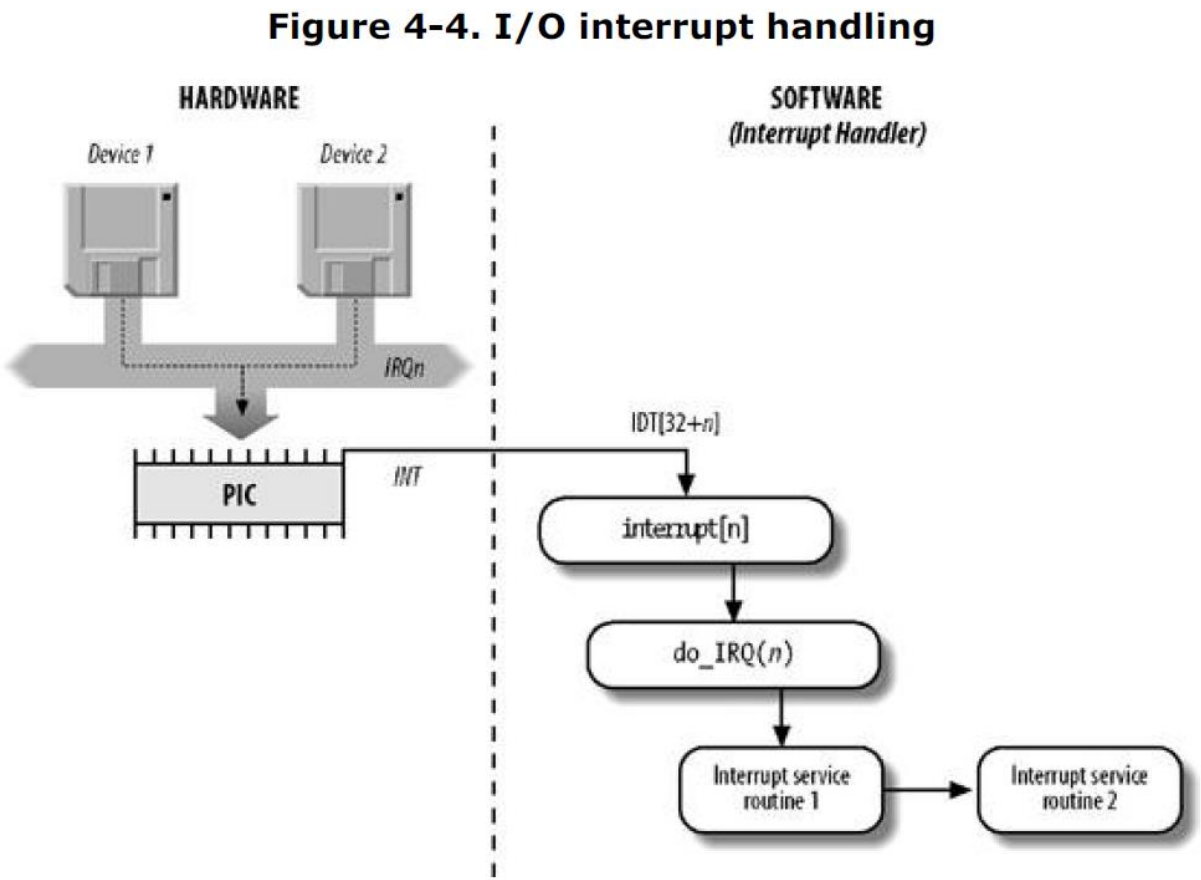
- 现运行进程还是优先级最高的进程嘛?
  - 不是! 进程切换
  - 是 从核心栈 pop EAX~通用寄存器; IRET，恢复执行taskA

例：若taskA是执行应用程序，求每个时间段，中断嵌套的层次



记下来中断嵌套的层次，知晓系统是不是工作在中断上下文中。
这个数据结构是现运行进程PCB的成员：preempt_count 中 hardirq 的值。

中断处理程序和中断服务例程（ISR）

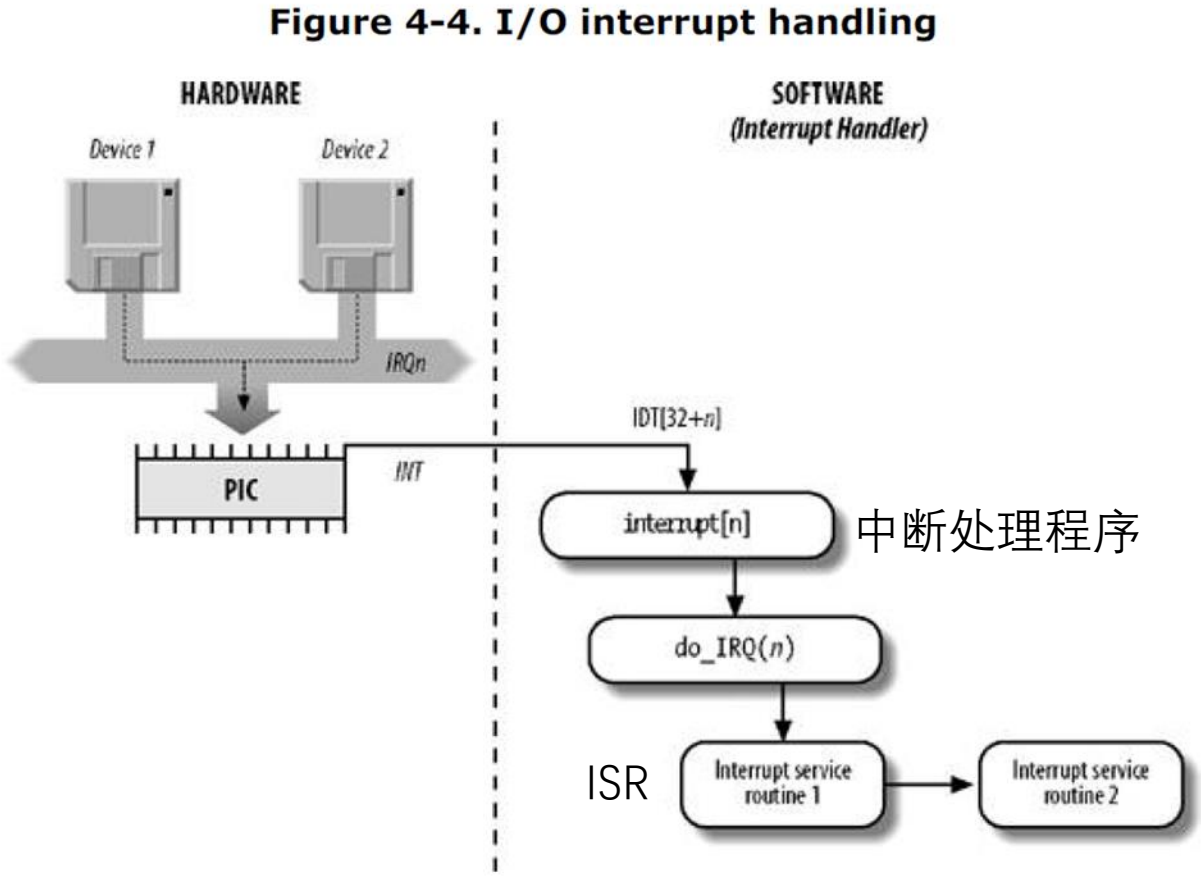


每根IRQ线，唯一对应一个中断处理程序。
每个外设有一个ISR。
一根IRQ线可以挂好多外设，所以一个中断处理程序会管好多ISR。

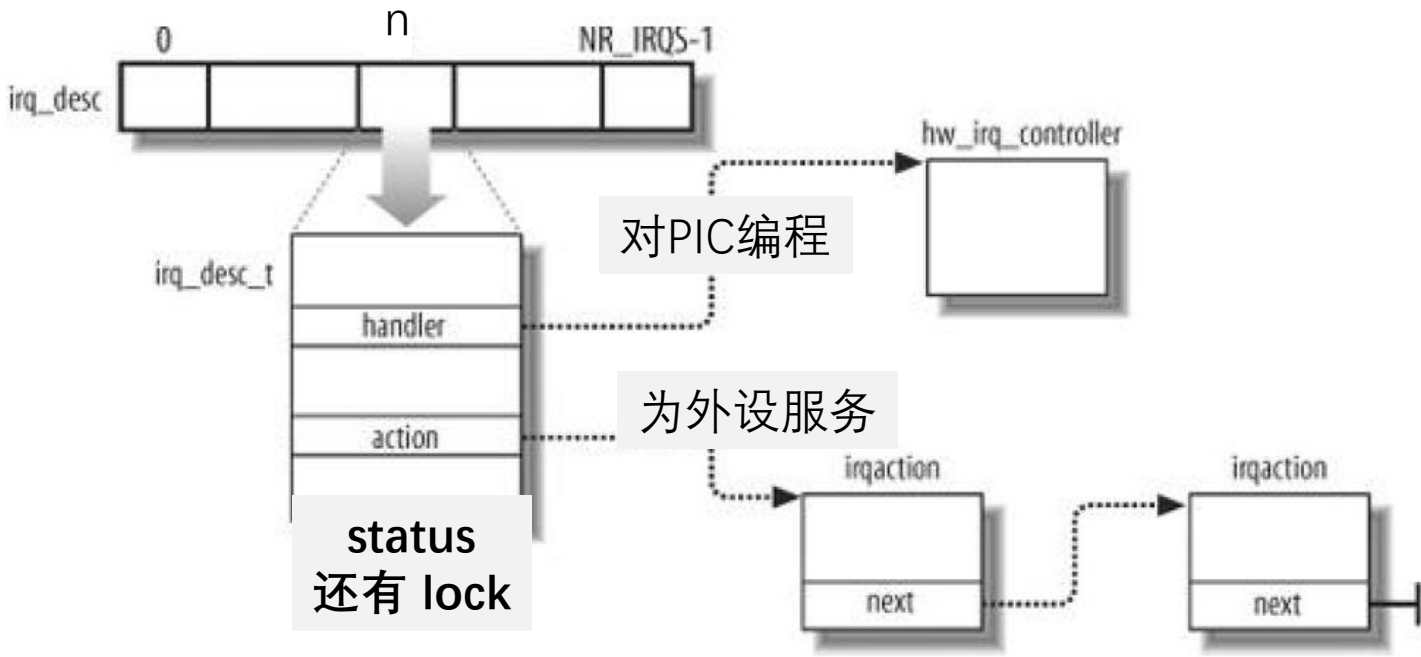
/*IDT[n+32]登记的中断处理程序为挂在IRQn上的所有外设服务*/
中断处理程序（n+32）
{
    遍历ISR(i)
        ISRi （）
}

```
ISRi （）
{
    if( 我管的设备有IO )
        处理 （）；
    返回
}
```

IRQ数据结构：中断请求描述符数组



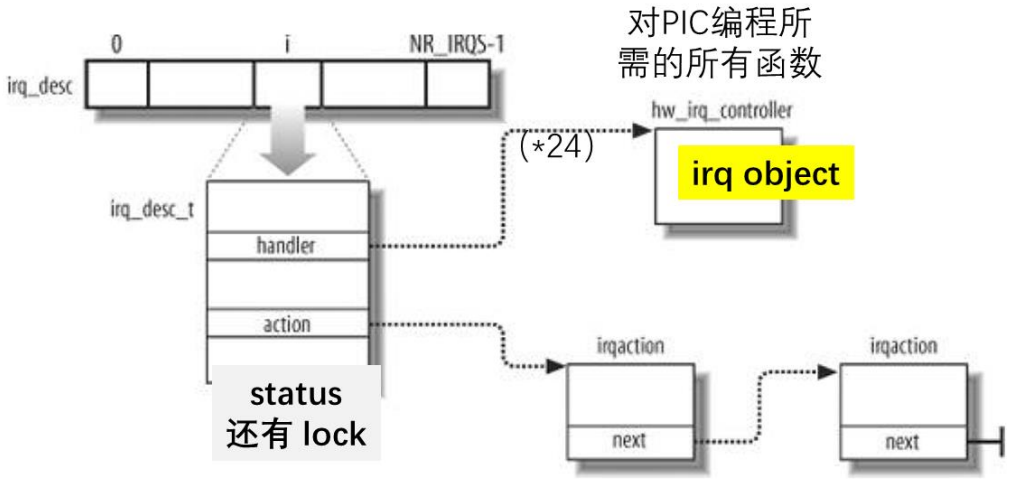
中断处理程序



每根 irq 线，有一个 irq_desc_t 结构，登记着系统为挂在这根 irq 线上的外设服务，所需的所有函数 & irq线的状态。

IRQ数据结构 1
对PIC编程（控制中断请求线irqn）

for 使用2块8259A的单处理器系统

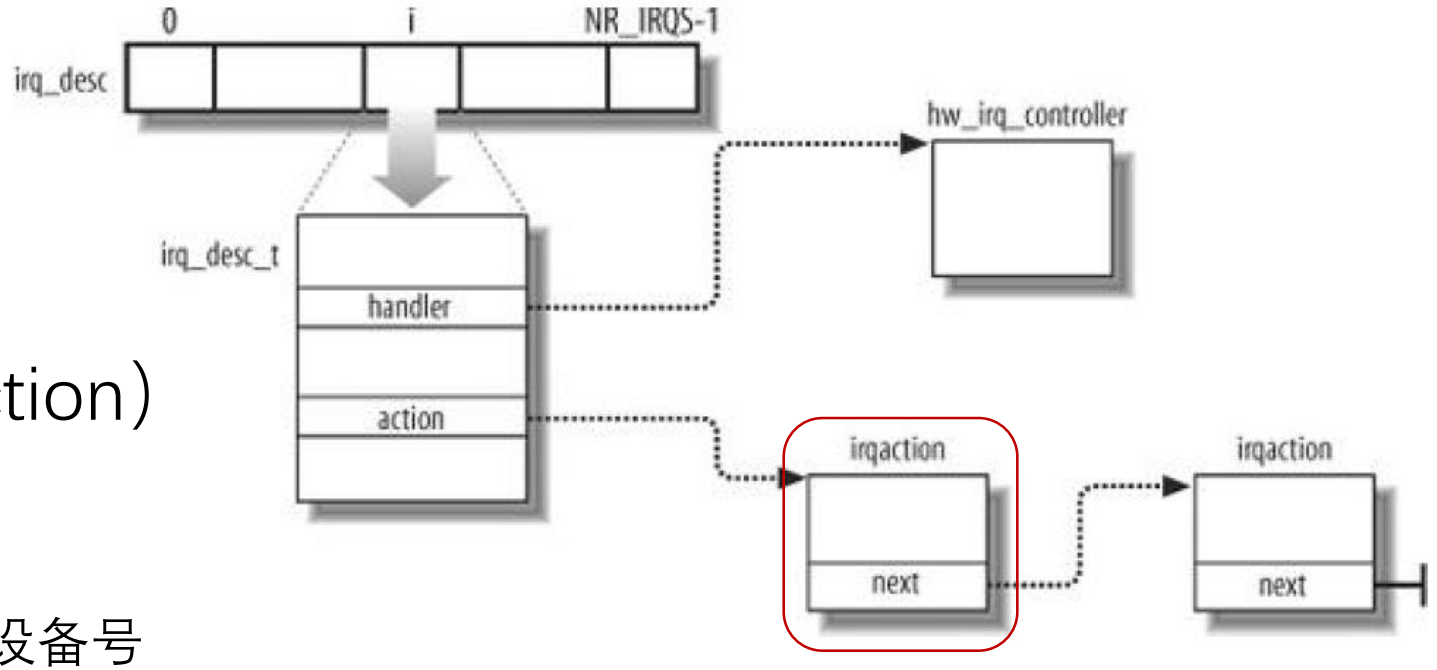



```
struct hw_interrupt_type i8259A_irq_type = {
    .typename     = "XT-PIC",
    .startup      = startup_8259A_irq,
    .shutdown     = shutdown_8259A_irq,
    .enable       = enable_8259A_irq,
    .disable      = disable_8259A_irq,
    .ack          = mask_and_ack_8259A,
    .end          = end_8259A_irq,
    .set_affinity = NULL
};
```

1、handler->ack( )　　(mask_and_ack_8259A( ))：
收到irq请求　CPU向8259A的指定端口写其收到的中断向量#，向8259A确认收妥中断请求。

2、 handler->end( )　　(end_8259A_irq( )) :
irq请求（ISR）处理结束　CPU向8259A的0x20h端口发送EOI命令。收到EOI之后，8259A才能向CPU转发同级或低优先级的中断请求。

IRQ数据结构 2
对外设编程



- irqaction（一个外设一个irqaction）
  - **handler　ISR程序的入口地址**
  - dev_id
    - 最低端的配置，给出设备的主次设备号
    - 更好一点，直接指向devtab，其中有设备IO请求队列&可以找到供这个外设使用的缓存
  - irq　　n
  - dir　　/proc/irq/n
  - name　外设的名字
  - flags
  - next

Table 4-7. Flags of the irqaction descriptor

Flag name	Description
SA_INTERRUPT	The handler must execute with interrupts disabled.
SA_SHIRQ	The device permits its IRQ line to be shared with other devices.
SA_SAMPLE_RANDOM	The device may be considered a source of events that occurs randomly; it can thus be used by the kernel random number generator. (Users can access this feature by taking random numbers from the */dev/random* and */dev/urandom* device files.)

- 中断处理程序需要知道
  - 每个外设的缓存在内存中的位置
  - 可能需要知道，每个外设的IO请求队列
- 这2个关键的信息，登记在外设的devtab中。

PS：内核为每个外设配有一个或多个缓存，用来存放该外设的IO数据。缓存在RAM中的地址一般登记在这个设备的dev_tab中。而，dev_tab登记在这个设备的块设备开关表或字符设备开关表中。

- irqaction的dev_id，帮助ISR（中断处理程序）找到外设的devtab!

补充的细节：

1、硬中断处理程序执行的时候，与之对应的irq线disable（叫做屏蔽掉这根irq线）。所以，不用担心相同的硬中断处理程序嵌套执行的情况。

这大大简化了中断处理程序的编程复杂度！！

（所有中断处理程序不用是可重入的了）

2、时钟硬中断处理程序是关中断运行的。

其它许多中断处理程序是开中断运行的。响应挂在其它irq线上的设备的中断请求。

结论：中断处理程序运行过程中，系统屏蔽与之相对应的irq线上所有设备的中断请求。这意味着什么？

中断的上半段和下半段

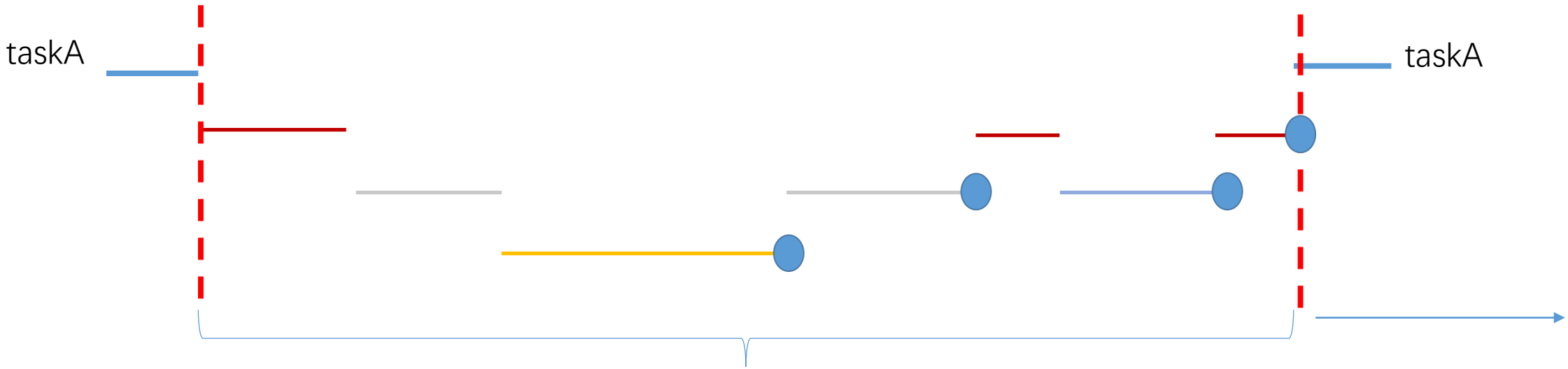
意味着什么？

中断嵌套

- taskA 有3种情况：
- 1、执行某个APP； 2、执行系统调用； 3、执行中断处理程序

- 情况 3，我们说系统工作在中断上下文。其它情况，我们说，系统没有工作在中断上下文。
  Linux响应中断，这个叫做中断嵌套。

- Linux系统，不为中断设置优先级。这也就是说，Linux认为所有的中断优先级相等。
- 只有2种类型的中断处理程序
  - 关中断运行。新出现的中断请求，开中断后响应。
  - 开中断运行。如果，中断处理程序开中断运行，运行期间，系统响应一切中断请求。


taskA
taskA
CPU 接收外设送来的第一批数据
CPU 可以接收外设送来的下批数据

中断的上半段和下半段

意味着什么?

中断嵌套

- taskA 有3种情况：
- 1、执行某个APP； 2、执行系统调用； 3、执行中断处理程序

- 情况 3，我们说系统工作在中断上下文。其它情况，我们说，系统没有工作在中断上下文。
  Linux响应中断，这个叫做中断嵌套。

- Linux系统，不为中断设置优先级。这也就是说，Linux认为所有的中断优先级相等。
- 只有2种类型的中断处理程序
  - 关中断运行。新出现的中断请求，开中断后响应。
  - 开中断运行。如果，中断处理程序开中断运行，运行期间，系统响应一切中断请求。

+ 在中断处理程序中，将与硬件交互的部分屏蔽中断的部分缩至最短

Linux中断处理程序的框架（naive）

中断的上半段
关中运行的critical section
开中运行的Noncritical section
（禁irqn）

开irqn
中断的下半段
（响应所有中断请求）

IRET

A1:
称中断的下半段为可延迟函数（deferrable function）。
试问，这部分执行的时候，会被谁延迟呢？ 会很久嘛?

中断的上半段和下半段

Linux中断处理程序的框架（naive）

中断的上半段
关中运行的critical section
开中运行的Noncritical section
（禁irqn）

开irqn
中断的下半段
（响应所有中断请求）

IRET

中断需要处理的工作
上半段：对时间非常敏感的任务，
和硬件相关的任务，
要保证不被其它中断（特别是相同的中断）打断的任务
下半段：其它所有~

典型的需要由上半段执行的任务包括：
handler->ack( )： 编程PIC，确认中断到达。
可能需要读数据。
可能需要向外设发下一个IO命令。
handler->end( )： 编程PIC，告诉它ISR运行完毕。

举例：需要由下半段执行的任务 TCP/IP协议栈处理网络包

Linux中断处理程序的框架（naive）

中断的上半段
关中运行的critical section
开中运行的Noncritical section
（禁irqn）

开irqn
中断的下半段
（响应所有中断请求）

IRET

进　化

Linux中断处理程序的框架

中断的上半段

标记有活没干完（optional）

开irqn

看下标记，需要的话

去做没干完的活（中断的下半段）

IRET

Linux中断处理程序的框架

中断的上半段

标记有活没干完（optional）

开irqn

看下标记，需要的话

去做没干完的活（中断的下半段）

IRET

软中断由

softirq_action结构表示，它定义在<linux/interrupt.h>中：

```
/ *
 *本结构代表一个软中断项
 */
struct softirq_action {
        void (*action) (struct softirq_action *);       /* 待执行的函数 */
        void *data;                                     /* 传给函数的参数 */
};
```

kernel/softirq.c中定义了一个包含有32个该结构体的数组。

```
static struct softirq_action softirq_vec[32];
```

每个被注册的软中断都占据该数组的一项。因此最多可能有32个软中断。注意，这是一个定

每个CPU有一个softirq_pending，32个bit的整数。用来登记需要处理的软中断。
把第n个bit置1，就是标记有 n#软中断 需要处理。

下半段执行n#软中断处理函数。这样做：
softirq_vec[n].action(softirq_vec[n]);

Linux中断处理程序的框架

中断的上半段

标记有活没干完（optional）

取消irqn的屏蔽

看下标记，需要的话

去做没干完的活（中断的下半段）

IRET

每个CPU有一个softirq_pending，32个bit的整数。用来登记需要处理的软中断。
把第n个bit置1，就是标记有 n#软中断 需要处理。

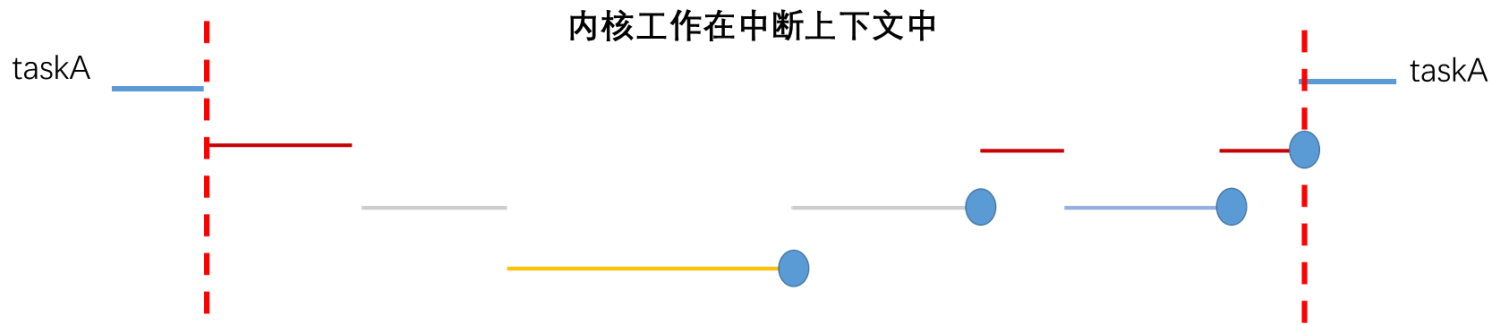
下半段执行n#软中断处理函数。这样做：
softirq_vec[n].action(softirq_vec[n]);

每个 ISR 用固定的软中断号 n

软中断	下标（优先级）	说明
HI_SOFTIRQ	0	处理高优先级的 tasklet
TIMER_SOFTIRQ	1	和时钟中断相关的 tasklet
NET_TX_SOFTIRQ	2	把数据包传送到网卡
NET_RX_SOFTIRQ	3	从网卡接收数据包
SCSI_SOFTIRQ	4	SCSI 命令的后台中断处理
TASKLET_SOFTIRQ	5	处理常规 tasklet

情景分析：　数据包到网卡，系统执行网络中断处理程序

下半部，什么时候可以开始执行



嵌套执行的多个中断，上半段全部结束的时候

Linux中断处理程序的框架（almost done）

中断的上半段

标记有活没干完（optional）

取消irqn的屏蔽

看下标记，需要的话

去做没干完的活（中断的下半段）

IRET

每个中断处理程序用固定的软中断号 n

1、preempt_count. hardirq_count++;
2、应答PIC，告诉它中断请求已经收妥，开中断（可选）；
3、接收外设IO数据，将其存入缓存（内核）
PS：键盘这种数据传送量少的设备，有这一步（这是读数据缓存）；
以DMA方式工作的设备（网络、磁盘），没这一步；
~~~~~~
4、置1 softirq_pending的n比特；
5、preempt_count. hardirq_count--;
6、开irqn，EOI（ ISR已经执行完毕）；
7、if(preempt_count. hardirq_count == 0 && softirq_pending！=0 && ~)
找到softirq_pending中所有置为1的bit x，
依次执行softirq_vec[x].action；
~~~~~~
8、 IRET

情景分析： 现运行进程跑APP。
数据包到网卡，系统响应，执行网络中断处理程序（3#）。其间，发生了时钟中断（1#）

软中断执行的过程中，系统响应了新的中断请求

- 硬中断处理程序执行时会激活新的软中断。带来2个问题：
  1、中断下半段和中断上半段会同时访问**softirq_pending**，这个变量需要互斥访问。
  2、新激活的软中断什么时候执行？

Linux 软中断处理　do_softirq( )，almost done

- **关中断**
- 循环10次
  - pending = _ _softirq_pending;　//pending是函数do_softirq( )的局部变量
  - _ _softirq_pending = 0;　**local_irq_enable( )开中断；**
  - if(pending)
  - {

```
h指针变量指向soft_irq_vec数组的起始地址;
do {
        if( pending & 1 )
            h_action(h);
        h++;
         pending >>= 1;
} while(pending);
```

  - }
  - **关中断**

do_softirq( )第1次循环，如果系统响应了新的中断请求（可能会有好多好多），这些硬中断激活的软中断，do_softirq( )执行第2次循环时处理。do_softirq( )第2次循环，如果系统响应了新的中断请求（可能会有好多好多），这些硬中断激活的软中断，do_softirq( )执行第3次循环时处理。。。。。。

Linux 软中断处理　do_softirq( )，almost done

- **关中断**
- 循环10次
  - pending = _ _softirq_pending;　//pending是函数do_softirq( )的局部变量
  - _ _softirq_pending = 0;　**local_irq_enable( )开中断；**
  - if(pending)
  - {
    h指针变量指向soft_irq_vec数组的起始地址;
    do {
    if(pending & 1)
    h_action(h);
    h++;
    pending >>= 1;
    } while(pending);
  - }
  - **关中断**

if（preempt_count . softirq_count ！ = 0）
return；
preempt_count . softirq_count ++；

如果10次之后，_ _softirq_pending还是非0
就
wakeup_softirqd( )，唤醒 *ksoftirqd/n* kernel thread
之后，现运行进程中断返回。

所有的活做完以后，　preempt_count . softirq_count --；

ksoftirqd/n（优先级最低的进程，每个CPU有自己的ksoftirq线程，n是CPU的代号）

```
for(;;) {
    set_current_state(TASK_INTERRUPTIBLE );
    schedule( );
    /* now in TASK_RUNNING state */
    while (local_softirq_pending( )) {
        preempt_disable();
        do_softirq( );    //系统idle的时候，也可以处理软中断啦
        preempt_enable();
        cond_resched( );    //有没有唤醒优先级高的进程?
    }
}
```

- Fact：CPU没有收到中断请求，积压的软中断任务不会得到运行。

- 如果此轮中断任务非常繁重，前面一张PPT的 do_softirq 将循环10 轮。如果10轮结束的时候，还有软中断没有处理完毕。唤醒ksoftirq。这个进程优先级很低，所有进程，包括APP优先级都比这个进程高。∴IRET退出中断上下文，回用户态。无论是否被中断的APP恢复运行，APP开始跑，用户感到系统有了响应。

- 若系统idle，IRET后，ksoftirq运行，它将执行 do_softirq（）完成尚未完成的软中断任务。
  - do_softirq 循环10 轮~
    这个策略不配上ksoftirq，后果很严重。如果网络中断发生于系统idle的时候，在do_softirq执行期间收到的以太数据包，除了前10个，后面的全丢啦!

do_irq（）　中断处理过程

- 保存现场
- DS指向用户数据段
- 硬中断计数器++
- 对PIC编程
  - ACK　~
- 对外设编程，执行ISR~
  - 可能需要读数据
  - 复位外设
  - 发下一个IO命令
  - 可能需要标记，中断下半段等待处理
  - EOI　开irqn
- 硬中断计数器--
- irq_exit　有下半段需要处理吗？　Y, do_softirq()
- ret_from_intr　恢复现场，IRET　//if（preempt_count != 0）　中断返回
  else　执行中断返回前的例行调度

中断的上半段

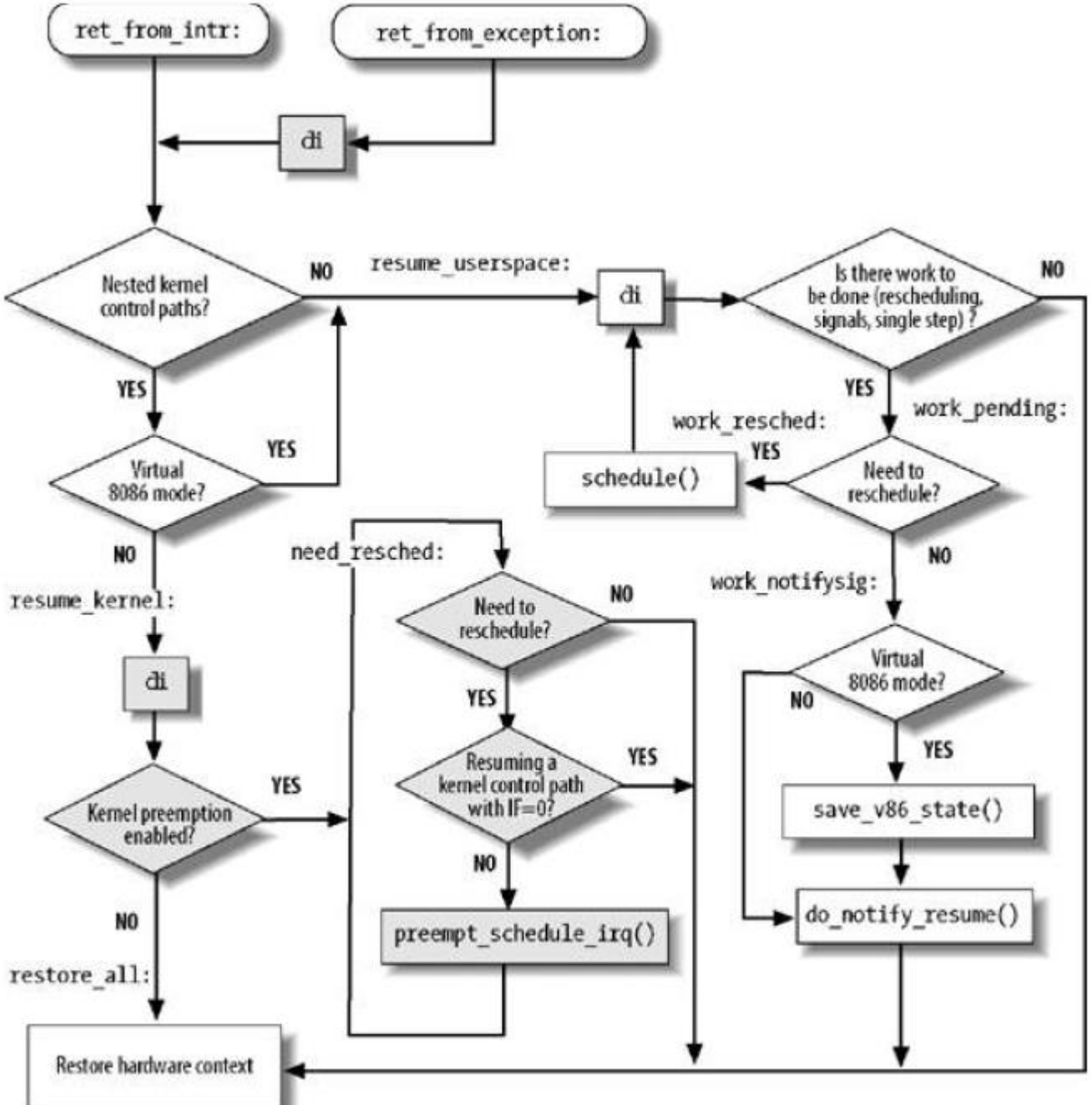
可剥夺的内核

中断返回

1、现运行进程需要让出CPU嘛？
2、让出CPU，安全嘛？（现运行进程可以被抢占嘛）

表4-10：preempt_count的字段

位	描述
0～7	抢占计数器(max value = 255)
8～15	软中断计数器 (max value = 255)
16～27	硬中断计数器 (max value = 4096)
28	PREEMPT_ACTIVE 标志



SMP中断系统

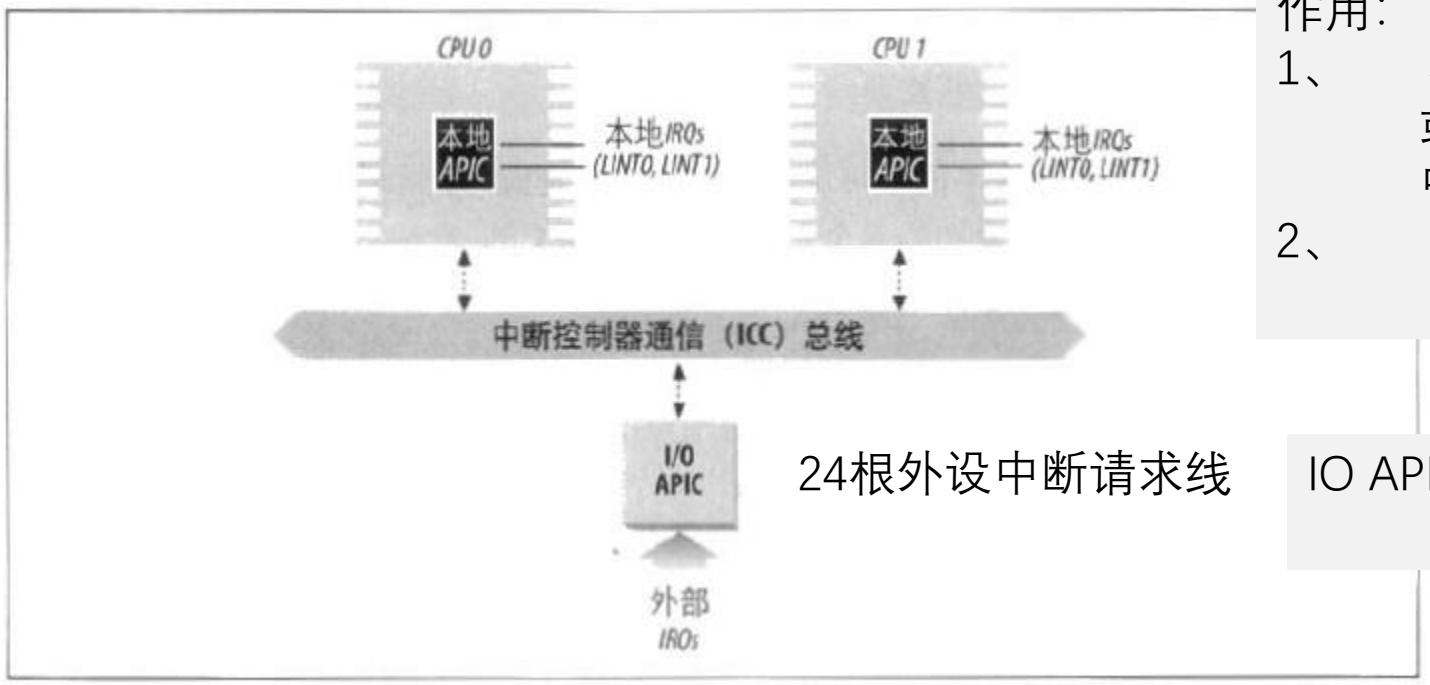


图 4-1：多 APIC 系统

每个CPU带一块APIC
作用：
1、 收到IO APIC的中断信号
或 其它CPU的中断信号时，
中断CPU
2、 向其它CPU发处理器间中断信号

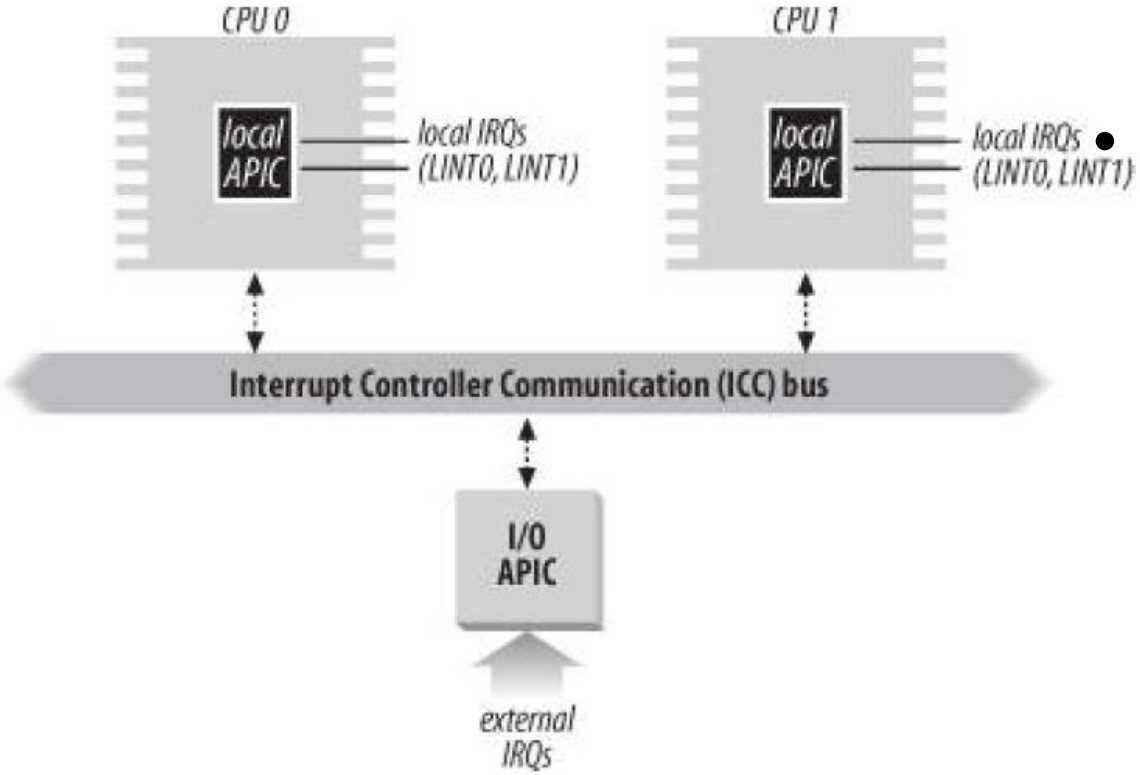
24根外设中断请求线

IO APIC的作用：
将外设中断请求分发给不同的APIC（CPU）

Advanced Programmable Interrupt Control

SMP系统中，IRQ请求的分发

- 中断请求round robin到每个处理器



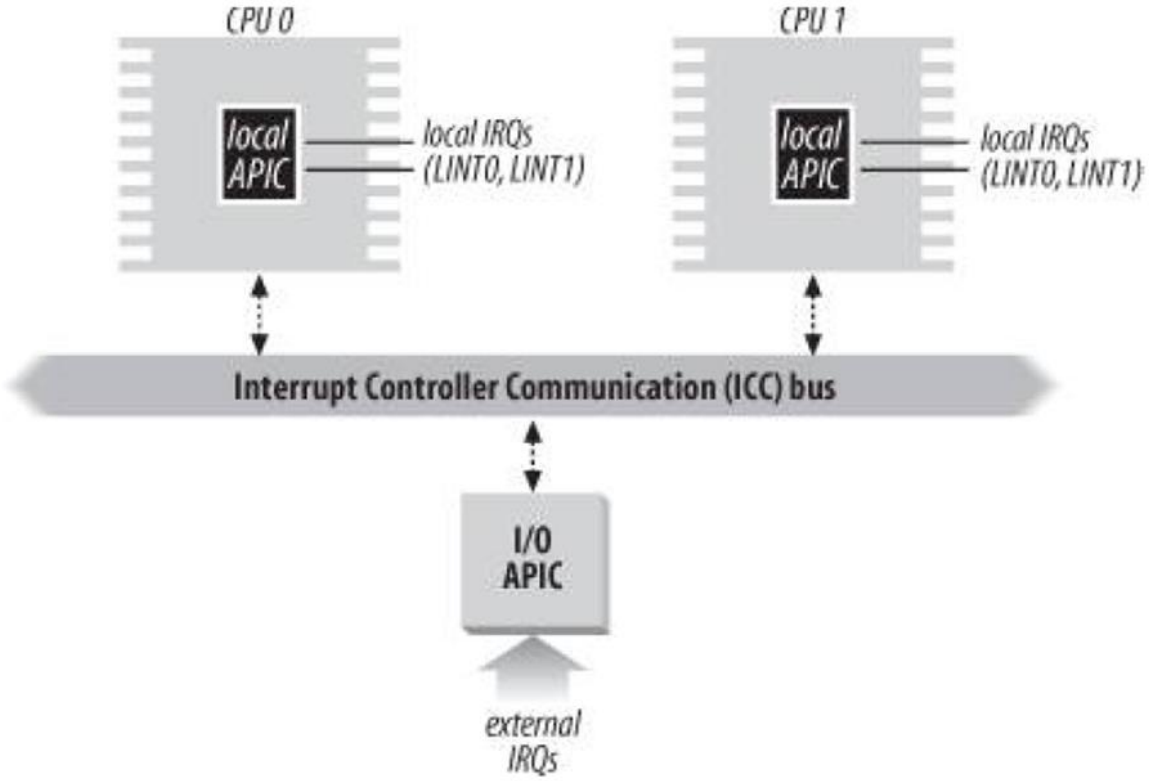
- 实现
  - 所有APIC中，任务优先级寄存器TPR值固定且相等（不用）
  - APIC中仲裁优先级寄存器（0~15）的使用
    - 将中断分配给值最高的APIC。
    - 其它，值加1。如果值是15，赋值为获得中断的APIC的仲裁优先级寄存器+1
    - 获得中断的APIC， 值清0

中断重定向表（24项），每一项对应于一根IRQ线，登记：
中断向量、优先级（不用）、目标处理器（全部）、处理器选择规则（默认，也就是动态分配）

IPI

- 不过IRQ线，过ICC 总线

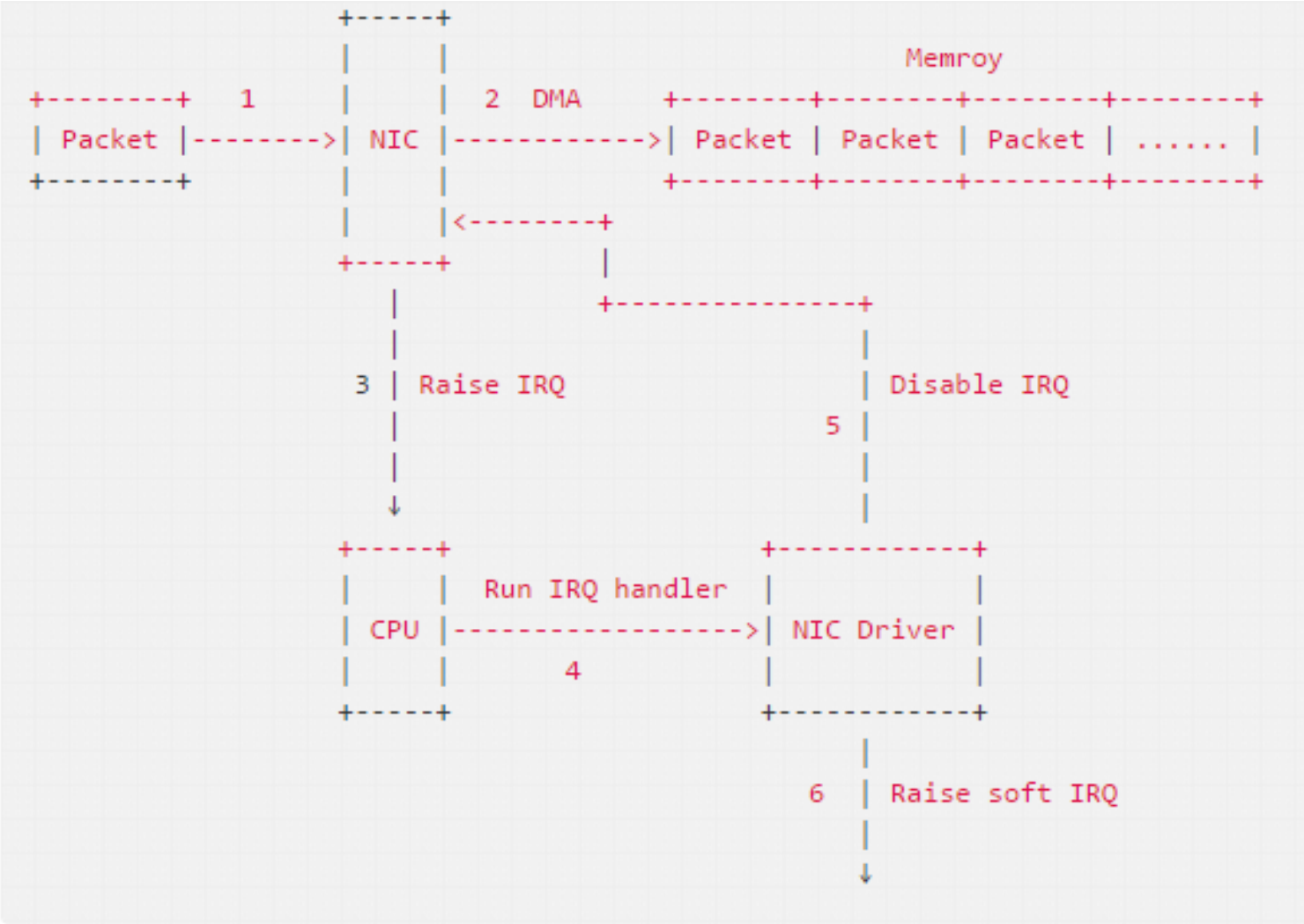
- 3种IPI
  - CALL_FUCTION_VECTOR
    主叫CPU将一个函数的入口地址存在全局变量call_data中，之后发送中断
  - RESCHEDULE_VECTOR
  - INVALIDATE_TLB_VECTOR



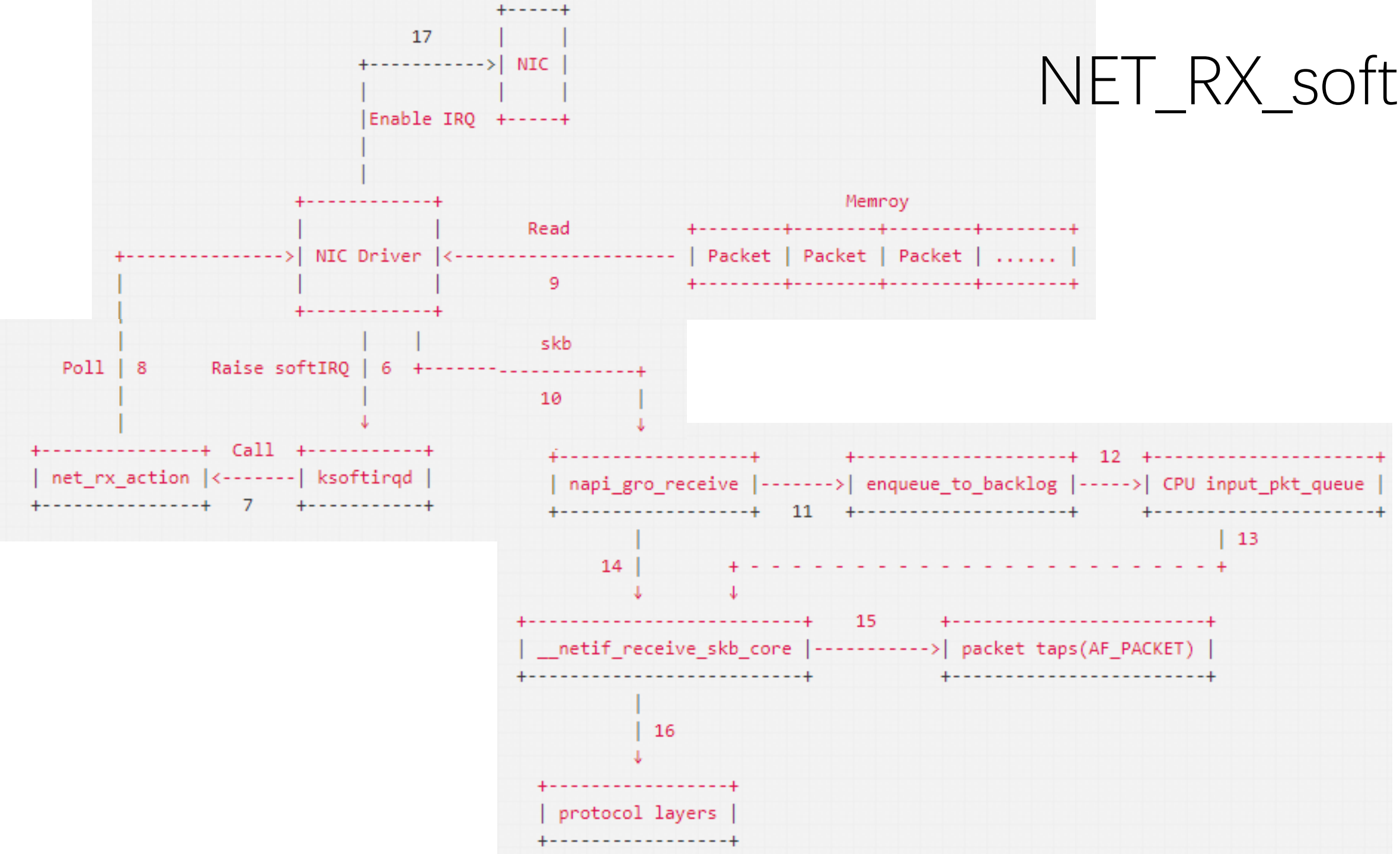
禁中断线的意义

- 数据的一致性：保护内核数据结构的互斥访问
- 提高IO系统的工作效率：同一个外设不要频繁中断CPU

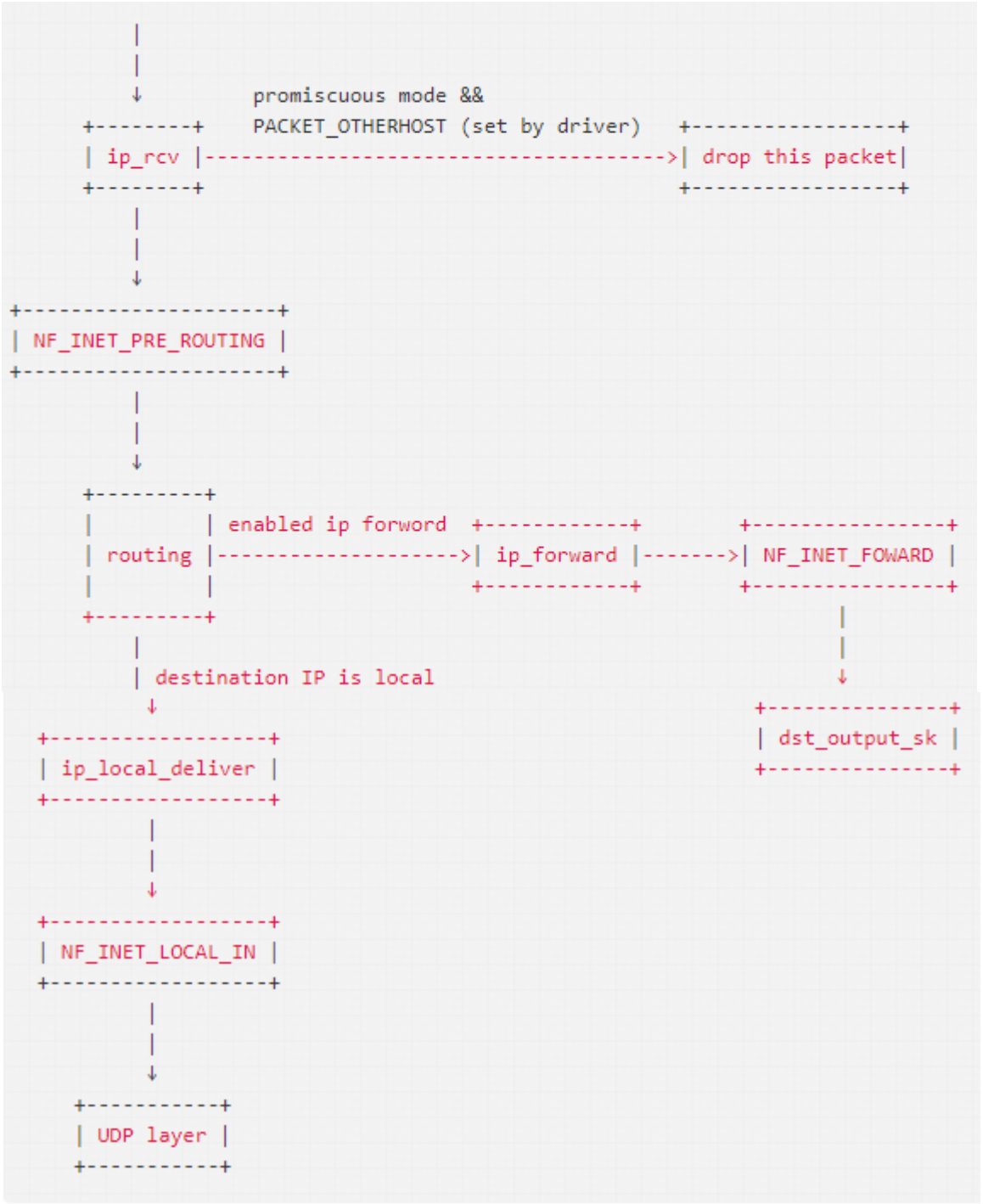
实例：Linux网络　数据包的接收过程



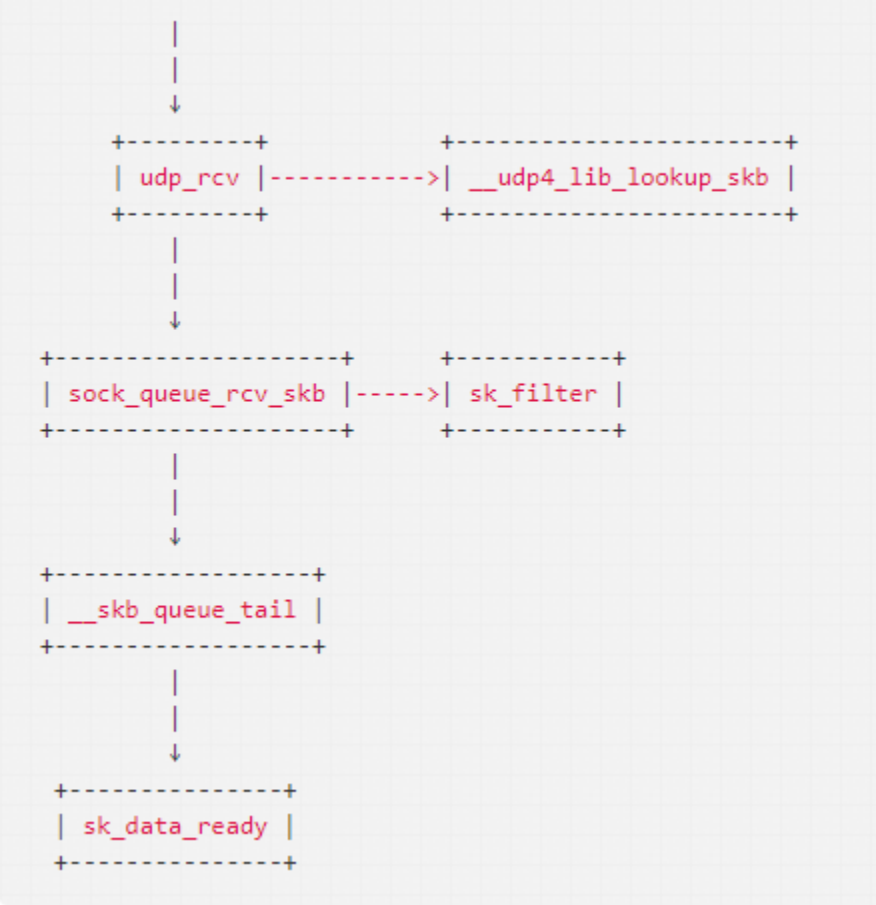
NET_RX_softirq



IP 层


promiscuous mode &&
PACKET_OTHERHOST (set by driver)
ip_rcv
drop this packet
NF_INET_PRE_ROUTING
routing
enabled ip forword
ip_forward
NF_INET_FOWARD
destination IP is local
dst_output_sk
ip_local_deliver
NF_INET_LOCAL_IN
UDP layer

UDP层



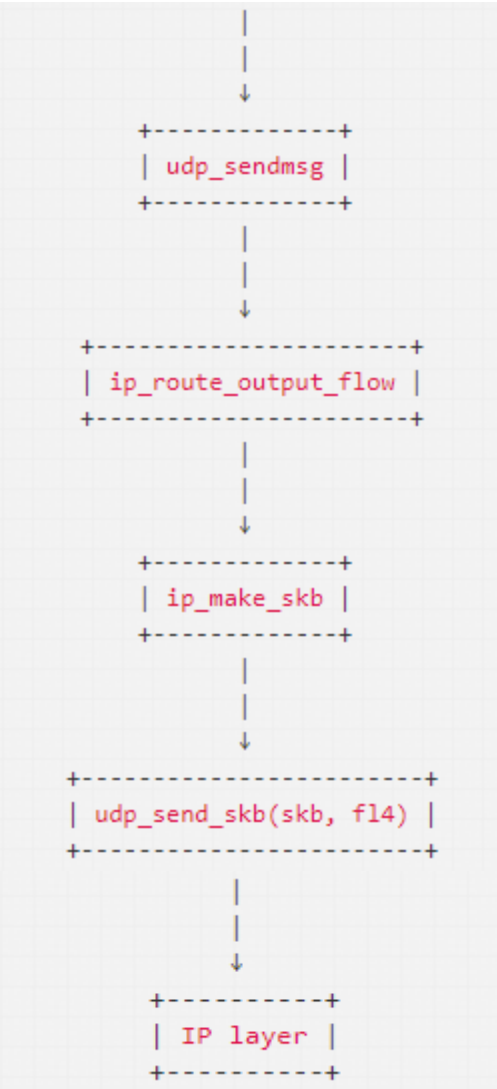
- udp_rcv： udp_rcv函数是UDP模块的入口函数，它里面会调用其它的函数，主要是做一些必要的检查，其中一个重要的调用是__udp4_lib_lookup_skb，该函数会根据目的IP和端口找对应的socket，如果没有找到相应的socket，那么该数据包将会被丢弃，否则继续
- sock_queue_rcv_skb： 主要干了两件事，一是检查这个socket的receive buffer是不是满了，如果满了的话，丢弃该数据包，然后就是调用sk_filter看这个包是否是满足条件的包，如果当前socket上设置了filter，且该包不满足条件的话，这个数据包也将被丢弃（在Linux里面，每个socket上都可以像tcpdump里面一样定义filter，不满足条件的数据包将会被丢弃）
- __skb_queue_tail： 将数据包放入socket接收队列的末尾
- sk_data_ready： 通知socket数据包已经准备好

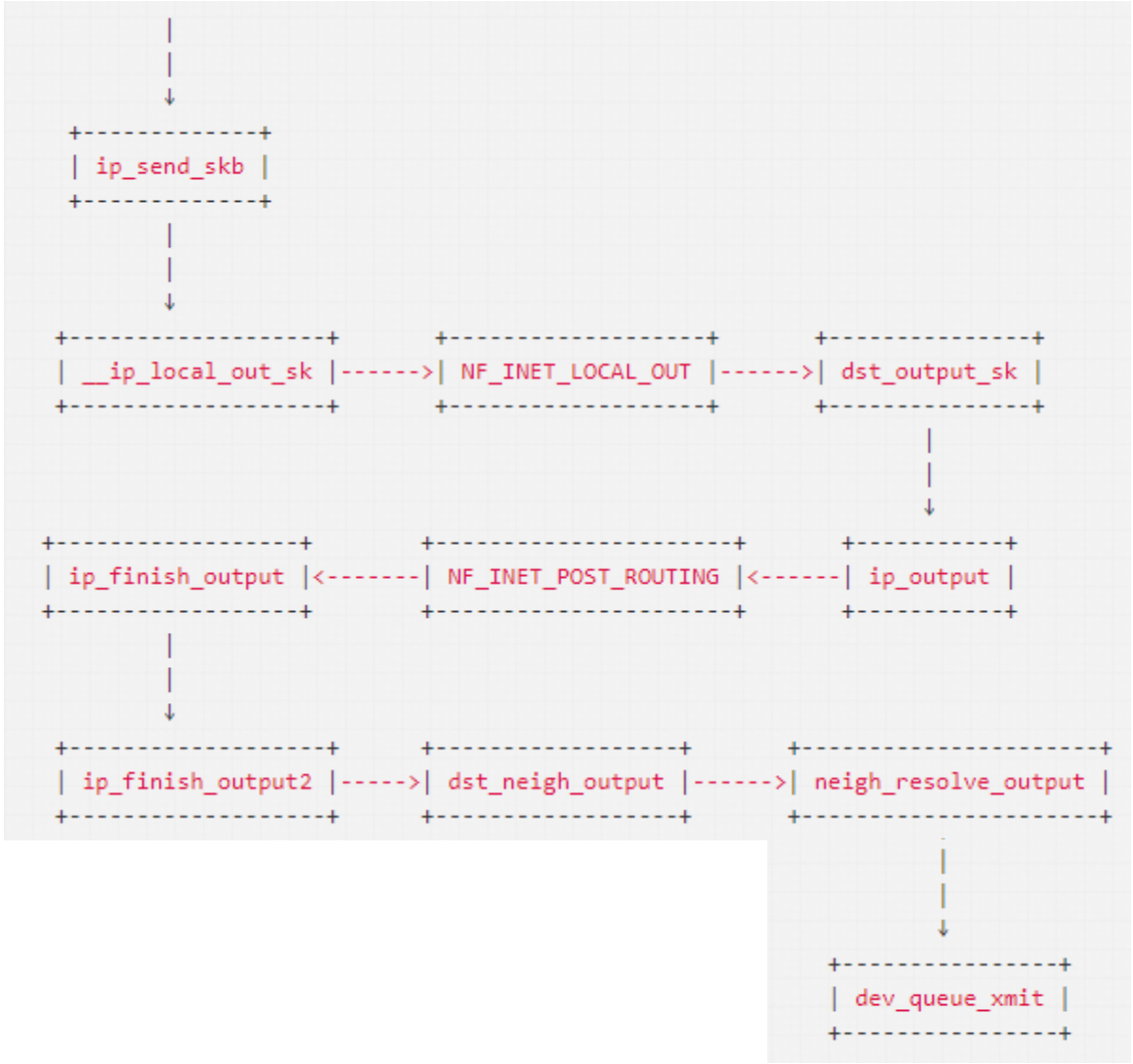
socket层

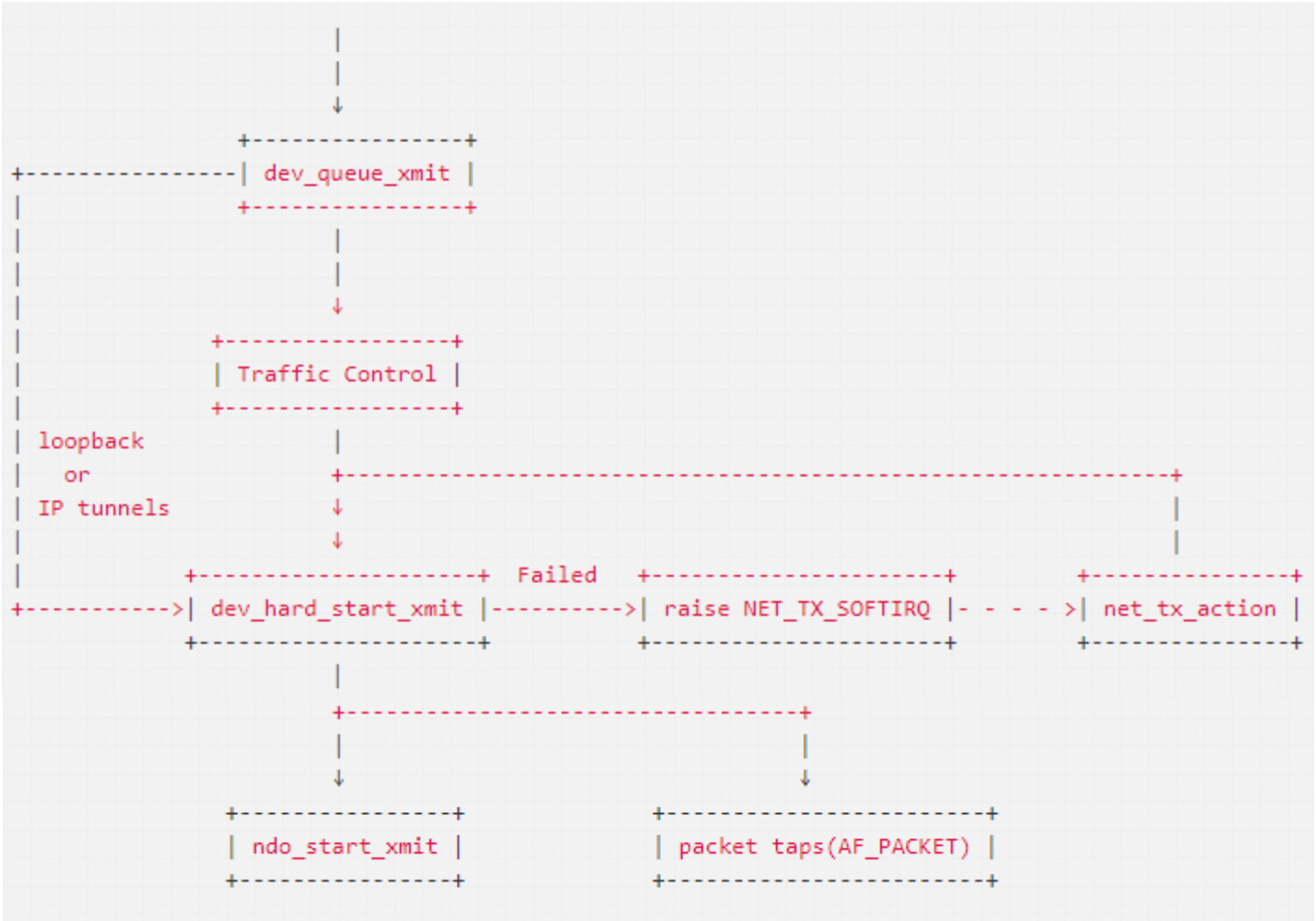
- 应用层一般有两种方式接收数据
  - 一种是recvfrom函数阻塞在那里等着数据来，这种情况下当socket收到通知后，recvfrom就会被唤醒，然后读取接收队列的数据；
  - 这里，作者对内核应该不是太熟。应该是，被recvfrom阻塞的进程，被网络中断程序唤醒，之后，将**socket接收队列**中队首的UDP包读入自己的数据段。recvfrom函数拿到了UDP包（想要的网络数据），返回。

  - 另一种是通过epoll或者select监听相应的socket，当收到通知后，再调用recvfrom函数去读取接收队列的数据。两种情况都能正常的接收到相应的数据包。

The end


udp_sendmsg
ip_route_output_flow
ip_make_skb
udp_send_skb(skb, fl4)
IP layer


ip_send_skb
__ip_local_out_sk
NF_INET_LOCAL_OUT
dst_output_sk
ip_output
NF_INET_POST_ROUTING
ip_finish_output
ip_finish_output2
dst_neigh_output
neigh_resolve_output
dev_queue_xmit


dev_queue_xmit
Traffic Control
loopback
or
IP tunnels
dev_hard_start_xmit
Failed
raise NET_TX_SOFTIRQ
net_tx_action
ndo_start_xmit
packet taps(AF_PACKET)