# 中断处理

中断信号由外部设备发起，准确来说是由外部设备的控制器发起，因为外部设备本身并不能发起信号。必须网卡设备，的那个网络数据包到达网卡，网卡的控制器就向IO APIC发送中断信号，IO APIC把信号发送给本地APIC，本地APIC把信号传送给CPU，如果根据当时情况，要处理这个中断，就保存当时的运行上下文，切换到中断上下文中，根据IDT查找对应的处理函数进行处理。处理完成后，需要恢复中断之前的状态。

大致过程就如上面所述，但是具体这个过程是怎么执行的，关键几点如下：

1、设备控制器如何发送中断信号

2、APIC如何接受中断信号，以及做了什么处理

3、处理器收到中断信号又做了什么操作

在此之前，我们需要介绍下中断控制器8259A和APIC

8259A中断控制器由两片8259A芯片级联组成，每个芯片有8个中断输入引脚，其中IRQ2被用来连接从芯片，所以一共可以支持15个中断号，这也就是早期采用8259A中断控制器只能使用15个外部中断的原因，使用8259A中断控制器的工作架构如下：

每个外部设备连接一条中断线，当设备需要中断CPU时，通过这些中断线，发送中断请求。中断控制器感知到这些中断请求，会设置中断控制器中的中断请求寄存器的相应位为1，鉴于多个中断可能并发到达，中断控制器具备中断判优功能，当其选定一个中断作为当前响应中断时，会清除中断请求寄存器中的对应位，然后设置中断服务寄存器的某些位为1，表明CPU正在服务于某个中断请求。

另外8259A还有一个8位的中断屏蔽寄存器，每一位对应于一个中断线，当对应的位被设置后，表明要屏蔽这些中断。为了处理不同优先级的中断，中断控制器还有同一个优先权判决器，当一个中断到达时，判断到达的中断优先级和ISR中正在服务的中断优先级的大小，若高于正在服务的中断的优先级，需要打断当前中断的处理，转而处理新到达的中断请求，否则，不予理会。

中断触发方式：

中断请求输入端IR0～IR7可采用的中断触发方式有电平触发和边沿触发两种，由初始化命令字ICW1中的LTIM位来设定。

当LTIM设置为1时，为电平触发方式。i8259A检测到IRi（i＝0～7）端有高电平时产生中断。如采用这种触发方式，中断请求信号在被响应后应及时撤除，否则可能引起不该有的第二次中断。

当LTIM设置为0时，为边沿触发方式。8259A检测到IRi端有由低到高的跳变信号时产生中断

当外部设备请求服务时，设置自己对应寄存器的位为1，即成了高电平，那么中断控制器端就可以接受到中断信号，进入中断的处理。

由于8259A中断控制器只能应用与单处理器，且其中断源的限制，后来Intel开发了高级可编程中断控制器APIC

APIC由两部分：本地APIC和IO APIC。本地APIC和逻辑CPU绑定，它控制传递给逻辑处理器中断信号和产生IPI中断（这是处理器间中断，只用于多处理器情况）、

本地APIC可以接受一下中断源：

本地连接的IO设备，指直接链接在处理器中断管脚LINT0和LINT1上。

外部链接的IO设备，这些中断源由外部IO设备产生比如键盘、鼠标等。外部IO设备连接IO APIC，IO APIC 把中断信号发送给一个或者多个CPU(本地APIC)。

处理器间中断IPI，一个处理器可以通过发送IPI，中断其他的处理器或者处理器组。

APIC定时器，用于定时向其绑定的处理器发送中断信号

APIC内部错误，当本地APIC自身出现错误，可通过编程让其给绑定的处理器发送中断信号。

以上中断源称为本地中断源，当本地APIC接收到一个中断信号，会通过某个发送协议把信号发送给处理器核心，具体可以通过一组被称之为local vector table 的APIC寄存器设置某个中断源的中断号。

而当接收外部中断时，则需要通过IO APIC，那么local vector table是个什么东西呢？

local vector table

LVT允许用户通过编程指定特定中断的处理动作，每个中断对应其中的一个表项，具体由一下几个32位寄存器组成：

LVT CMCI Register（FEE0 02F0h）

LVT Timer Register(FEE0 0320h)

LVT Thermal Monitor Register(FEE0 0330h)

LVT Performance Counter Register(FEE0 0340h)

LVT LINT0 Register(FEE0 3350h)

LVT LINT1 Register(FEE0 0360h)

LVT Error Register(FEE0 0370h)

本地APIC和IO APIC 关系如下：

由上图可以看到，IO APIC其实是作为一个PCI设备挂载在PCI总线上，和传统的PIC相比，IO APIC最大的作用在于中断的分发，外部设备不直接连接在本地APIC，而是连接在IO APIC,由IO APIC处理中断消息后发送给本地APCI。IO APIC一般由24个中断管脚，每个管脚对应一个RTE，并且其各个管脚没有优先级之分，具体中断的优先级由其对应的向量决定，即前面所说的local vector table。每当IO APIC接收到一个中断消息，就根据其内部的PRT表格式化出一条中断消息，发送给本地APIC。PRT表格式如下：

关于硬件先暂且介绍到这里吧，描述硬件实在感觉力不从心，感兴趣的可参考具体的硬件手册。

处理器收到中断信号又做了什么操作

在此之前我们需要明白几个概念：硬件中断、软件中断、异常

虽然前面描述的不够详细，但是相信还是可以看出，中断源可以分为两部分：本地中断源和外部中断源。本地中断源有些场合又称为软件中断，因为没有具体的硬件与之对应。而那些由具体硬件触发的中断则称为硬件中断。而异常则是程序指令流执行过程中的同步过程，比如程序执行过程中遇到除零错，很显然此时程序无法继续运行，只能处理完了这个异常，才可以继续运行。异常的同步特性和中断的异步又是一个明显的区别。另外在linux中为了让内核延期执行某个任务，也提出了一个软中断（software interrupt）的概念，这点在windows中与之对应的机制为DPC，即延迟过程调用。这两点咱们后面在说。

暂且不说中断异常的区别，系统使用一套机制来处理中断和异常，即在内核中维护了一张IDT（Interrupt Descriptor Table）中断描述符表，寄存器IDTR保存有表的基址。每个表项为8个字节。记录对应中断的处理函数的地址以及其他的一些控制位。所以每个中断对应一个表项。0-31号中断号位系统为预定义的中断和异常保留的，用户不得使用，所以硬件中断号从32开始分发。

每当CPU接收到一个中断或者异常信号，CPU首先要做的决定是否响应这个中断（具体由中断控制器根据中断优先级决定是否给CPU发送中断信号），如果决定响应，就终止当前运行进程的运行，根据IDTR寄存器获取中断描述符表基地址，然后根据中断号定位具体的中断描述符。这里中断描述符可分为两种情况：

中断门和陷阱门

任务门

1、 当中断描述符对应的是中断门或者陷阱门时，处理历程运行在当前进程的上下文中，即不需要发生进程上下文的切换，只是如果处理历程和当前进程的运行级别不同，则需要发生栈的切换，具体如下：

如果当前进程运行在level 3即用户态，则当中断发生时：

CPU从TSS中得到新栈的段描述符和栈指针

把段描述符和栈指针压栈

然后依次把EFLAGS、CS、EIP压栈

如果是一个异常引起一个错误码，则把错误码在EIP之后压栈

如果中断发生时当前进程运行在内核态，则就不需要发生栈的切换，仅仅需要执行上述的后两步。

具体动作参考下图：

2、当中断描述符对应一个任务门时，意味着此次中断的处理由一个单独的程序执行，和当前进程无关。使用新的任务处理中断的优缺点也很明显：

被中断进程的上下文会自动保存。

新的任务会使用一个新的内核栈，这就避免了如果中断是由栈错误引起的，使用中断门或者陷阱门带来的system crash。

处理程序运行在自己的地址空间中，和其他的程序隔离比较好。

当然缺点也很明显，每次中断都会进行任务的切换，进程上下文的切换所带来的开销要比上面两种方式大的多，并且每次中断都要进行两次进程切换：中断进入和中断返回。造成中断响应延迟过大

由于x86架构下的任务是非重入的，即一个中断处理程序执行期间会关中断，那么此时其他的进程就得不到调度，假如说这个处理程序很繁琐，那么会出现CPU处理时间分配不均的情况，且其他的中断得不到响应，这是不能允许的。所以操作系统在之前的基础上把中断处理历程分成两部分：上半部和下半部。上半部主要处理哪些中断来了必须要处理的事情，这个过程会关闭中断，所以此过程尽可能的短，在上半部处理结束，就开启中断。下半部主要处理不那么急迫的事情，这个过程开启中断，这样就增加了中断响应的效率。Linux和windows都采用了这种机制。LInux中使用软中断，而windows总则使用DPC延迟过程调用。

下面我们主要分析Linux下的softirq机制：

软中断可以使内核延期执行某个任务，他们的运作方式和具体的硬件类似，甚至可以说这里就是模拟的硬件中断，所以称之为软件中断也不为过。既然提到软中断，那么自然就设计到几个点：

软中断的注册

软中断的触发

软中断的处理

在3.11.1的内核版本中定义了10个软中断，并且系统不建议用户自己添加软中断，所以对于软中断基本用于已定义好的功用，而如果用户需要，可以使用其中的一个类型即TASKLET\_SOFTIRQ

具体的软中断类型如下：

复制代码

1 enum

2 {

3 HI\_SOFTIRQ=0,

4 TIMER\_SOFTIRQ,

5 NET\_TX\_SOFTIRQ,

6 NET\_RX\_SOFTIRQ,

7 BLOCK\_SOFTIRQ,

8 BLOCK\_IOPOLL\_SOFTIRQ,

9 TASKLET\_SOFTIRQ,

10 SCHED\_SOFTIRQ,

11 HRTIMER\_SOFTIRQ,

12 RCU\_SOFTIRQ, /\* Preferable RCU should always be the last softirq \*/

13

14 NR\_SOFTIRQS

15 };

复制代码

每个CPU维护一个软中断位图\_\_softirq\_pending，其实是一个32位的字段，每一位对应一个软中断。处理软中断时会获取当前CPU的软中断位图，根据各个位的设置，进行处理。

#define local\_softirq\_pending() \_\_get\_cpu\_var(irq\_stat).\_\_softirq\_pending

1、软中断的注册

软中断的核心机制是一张表，类似于IDT，包含32个softirq\_vec结构，该结构很简单：就是一个函数地址，每个软中断对应其中的一个，所以现在也仅仅使用前10项。

1 struct softirq\_action

2 {

3 void (\*action)(struct softirq\_action \*);

4 };

系统通过open\_softirq函数注册一个软中断，具体就是在softirq\_vec数组中根据中断号设置其对应的处理例程。

1 void open\_softirq(int nr, void (\*action)(struct softirq\_action \*))

2 {

3 softirq\_vec[nr].action = action;

4 }

nr是上面的一个枚举值，action便是对应软中断的处理函数。

2、软中断的触发

Linux系统通过raise\_softirq函数引发一个软中断，每个CPU有个软中断位图，有32位，最多可对应32个软中断，当置位图对应位为1时，表明触发了对应的软中断。在下次系统检查是否有软中断时就会被检测得到，从而进行处理。

复制代码

1 void raise\_softirq(unsigned int nr)

2 {

3 unsigned long flags;

4

5 local\_irq\_save(flags);

6 raise\_softirq\_irqoff(nr);

7 local\_irq\_restore(flags);

8 }

复制代码

核心函数在

复制代码

1 inline void raise\_softirq\_irqoff(unsigned int nr)

2 {

3 \_\_raise\_softirq\_irqoff(nr);

4

5 /\*

6 \* If we're in an interrupt or softirq, we're done

7 \* (this also catches softirq-disabled code). We will

8 \* actually run the softirq once we return from

9 \* the irq or softirq.

10 \*

11 \* Otherwise we wake up ksoftirqd to make sure we

12 \* schedule the softirq soon.

13 \*/

14 /\*如果我们没有在中断上下文中(硬中断或者软中断)，就唤醒软中断守护进程，否则之能等到从中断返回的过程中\*/

15 if (!in\_interrupt())

16 wakeup\_softirqd();

17 }

复制代码

1 void \_\_raise\_softirq\_irqoff(unsigned int nr)

2 {

3 trace\_softirq\_raise(nr);

4 or\_softirq\_pending(1UL << nr);

5 }

1 #define or\_softirq\_pending(x) this\_cpu\_or(irq\_stat.\_\_softirq\_pending, (x))

在raise\_softirq\_irqoff函数中看下，在设置了对应的位之后调用了in\_interrupt函数判断是否处于硬中断上下文或者软中断上下文，如果不在就调用wakeup\_softirqd唤醒守护进程处理软中断。否则的话等到中断退出的时候处理。

3、软中断的处理

处理时机：

软中断大概在三个地方会被检测是否存在，如果存在会进行处理:

从一个硬件中断返回时

在ksoftirqd内核线程中

在那些显式检查和执行待处理的软中断的代码中

中断上下文：CPU处于处理中断上半部或者下半部，内核用in\_interrupt来判断是否处于中断上下文。这是一个宏：

#define in\_interrupt() (irq\_count())

#define irq\_count() (preempt\_count() & (HARDIRQ\_MASK | SOFTIRQ\_MASK | NMI\_MASK))

可以看到这里中断上下文包括硬件中断、软件中断、NMI中断。说到这里，出现了一个preempt\_count（），LInux为每个进程的thread\_info结构中维护了一个preempt\_count字段，该字段是int型，因此有32位，用于支持内核抢占。当该字段为0的时候，表示当前允许内核抢占，否则不可以。具体请参考另一篇博文：Linux中的进程调度

处理过程：

软中断的处理核心都在do\_softirq函数。

复制代码

1 asmlinkage void do\_softirq(void)

2 {

3 \_\_u32 pending;

4 unsigned long flags;

5

6 if (in\_interrupt())

7 return;

8 /\*关闭所有中断 会保存eflags寄存器的内容\*/

9 local\_irq\_save(flags);

10

11 pending = local\_softirq\_pending();

12

13 if (pending)

14 \_\_do\_softirq();

15 /\*开启所有中断，恢复eflagS寄存器的内容\*/

16 local\_irq\_restore(flags);

17 }

复制代码

首先就会判断当前是否处于中断上下文，如果处于就直接返回，一个软中断既不能打断硬件中断也不能打断软件中断。如果不在中断上下文，就调用local\_softirq\_pending函数判断是否存在被触发的软中断，如果存在就进入if，调用\_\_do\_softirq函数， 否则开启中断，不做处理。

复制代码

1 asmlinkage void \_\_do\_softirq(void)

2 {

3 struct softirq\_action \*h;

4 \_\_u32 pending;

5 unsigned long end = jiffies + MAX\_SOFTIRQ\_TIME;

6 int cpu;

7 unsigned long old\_flags = current->flags;

8 int max\_restart = MAX\_SOFTIRQ\_RESTART;

9

10 /\*

11 \* Mask out PF\_MEMALLOC s current task context is borrowed for the

12 \* softirq. A softirq handled such as network RX might set PF\_MEMALLOC

13 \* again if the socket is related to swap

14 \*/

15 current->flags &= ~PF\_MEMALLOC;

16

17 pending = local\_softirq\_pending();

18 account\_irq\_enter\_time(current);

19

20 \_\_local\_bh\_disable(\_RET\_IP\_, SOFTIRQ\_OFFSET);

21 lockdep\_softirq\_enter();

22

23 cpu = smp\_processor\_id();

24 restart:

25 /\* Reset the pending bitmask before enabling irqs \*/

26 set\_softirq\_pending(0);

27

28 local\_irq\_enable();

29

30 h = softirq\_vec;

31

32 do {

33 if (pending & 1) {

34 unsigned int vec\_nr = h - softirq\_vec;

35 int prev\_count = preempt\_count();

36

37 kstat\_incr\_softirqs\_this\_cpu(vec\_nr);

38

39 trace\_softirq\_entry(vec\_nr);

40 h->action(h);

41 trace\_softirq\_exit(vec\_nr);

42 if (unlikely(prev\_count != preempt\_count())) {

43 printk(KERN\_ERR "huh, entered softirq %u %s %p"

44 "with preempt\_count %08x,"

45 " exited with %08x?\n", vec\_nr,

46 softirq\_to\_name[vec\_nr], h->action,

47 prev\_count, preempt\_count());

48 preempt\_count() = prev\_count;

49 }

50

51 rcu\_bh\_qs(cpu);

52 }

53 h++;

54 pending >>= 1;

55 } while (pending);

56

57 local\_irq\_disable();

58

59 pending = local\_softirq\_pending();

60 if (pending) {

61 if (time\_before(jiffies, end) && !need\_resched() &&

62 --max\_restart)

63 goto restart;

64

65 wakeup\_softirqd();

66 }

67

68 lockdep\_softirq\_exit();

69

70 account\_irq\_exit\_time(current);

71 \_\_local\_bh\_enable(SOFTIRQ\_OFFSET);

72 tsk\_restore\_flags(current, old\_flags, PF\_MEMALLOC);

73 }

复制代码

有了上面的铺垫，这里并不难理解。首先调用local\_softirq\_pending函数获取当前CPU软中断位图，然后调用\_\_local\_bh\_disable函数禁止本地软中断，接着调用lockdep\_softirq\_enter函数标记进入softirq context。下面的restart段就开始处理位图中的软中断了。

进入该节的首要操作对位图清零，因为随时可能有同种类型的软中断被触发，接着就调用local\_irq\_enable函数开启中断。下面h = softirq\_vec;是获取软中断描述符表的起始地址，进入do循环，从pending的第一位开始处理，每次pending右移1位，同时h++,所以h定位具体的软中断类型，pending判断是否被触发。如果被触发，那么进入if内部，内部就是调用了h->action(h)函数处理软中断;

在循环结束后，就再次关中断，然后重新读取pending，如果又有新的软中断被触发&&本次处理软中断未超时&&当前进进程的调度位TIF\_NEED\_RESCHED没有被设置&&重启次数没到最大限制，就再次执行restart节进行处理。否则只能唤醒守护进程下次再处理软中断。

之后就标记退出softirq context,开启软中断。

每个CPU都会有一个软中断守护进程ksoftirqd，同时也有一个软中断位图，我们触发的时候会指定CPU的id,各个CPU处理的软中断就不会影响，即使两个CPU处理同一类型的软中断。这样也避免了很多需要同步的操作，当然两个CPU都在处理同一类型的软中断，那么还是需要一定的同步来保障临界区的安全。如果在do\_softirq的末尾有未处理的软中断，就不得不唤醒守护进程进行处理；同样在raise\_softirq\_irqoff中在触发指定软中断后，判断是否在中断上下文，如果不在中断上下文就唤醒守护进程，否则下次检查调度的时候处理这些软中断。

基本的处理过程就如上所述，但是还是存在不少问题，前面代码片段中出现了很多开关中断的操作，为何需要有这些操作以及这些操作的原理如何？下面我们分析一下。

开关中断涉及到的函数主要有下面几个：

local\_irq\_save和local\_irq\_restore

local\_bh\_disable和local\_bh\_enable

local\_irq\_enable和local\_irq\_disable

其中1和3是针对hard irq，而2是针对soft irq。而且以上函数都是成对出现的。

local\_irq\_save和local\_irq\_restore是保存和恢复EFLAGS寄存器的状态，首先执行local\_irq\_save会保存EFLAGS寄存器的状态到一个变量，然后禁止本地中断（可屏蔽的外部中断），local\_irq\_restore会恢复EFLAGS寄存器到之前保存的状态。

local\_irq\_disable会直接禁止本地中断（可屏蔽的外部中断），而local\_irq\_enable会打开本地中断。这些都是针对可屏蔽外部中断，对于NMI和异常没有作用。

local\_bh\_disable会设置当前进程的抢占计数器，即增加对应的位，这样，当前进程就标识为不可抢占，也就关闭了软件中断。为什么说这里关闭了软件中断呢？因为前面我们设置了抢占计数器，而在每次检查准备调度时候,都会判断当前是否处于中断上下文，如果处于就不发生调度，从而不抢占当前进程。