中断

- 首先，网上诸多介绍中断硬件的资料大都参考自
  - `Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A: System Programming Guide, Part 1`
    - **Chapter 6 —— INTERRUPT AND EXCEPTION HANDLING**
    - **Chapter 10 —— ADVANCED PROGRAMMABLE INTERRUPT CONTROLLER (APIC)**
  - 即相关内容主要集中在**chapter 6**以及**chapter 10**

名词解释

- UP（Uni-Processor）
  - 系统只有一个处理器单元，即单核CPU系统
- SMP（Symmetric Multi-Processors）
  - 系统有多个处理器单元。各个处理器之间共享总线，内存等等。在操作系统看来，各个处理器之间没有区别
- PIC —— Programmable Interrupt Controller
- APIC —— Advanced Programmable Interrupt Controller
- MCH —— MEMORY CONTROL HUB内存控制中心
- ICH —— I/O controller Hub南桥

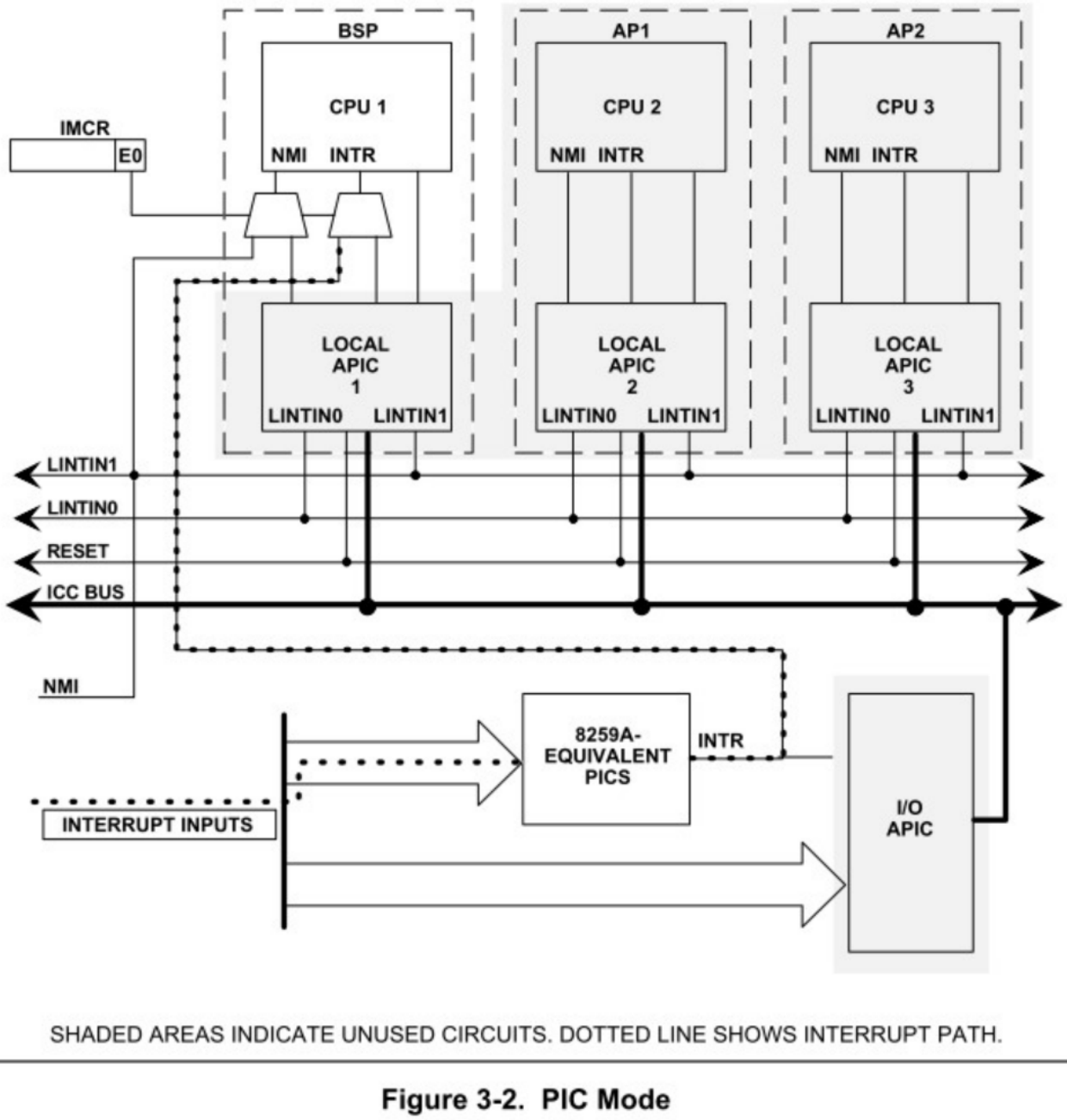


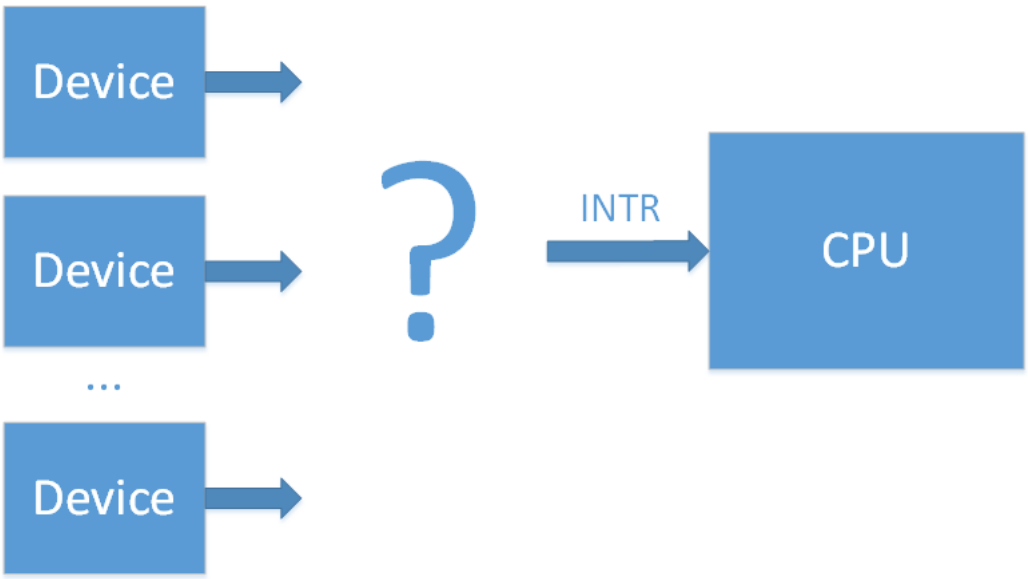
Figure 3-2. PIC Mode

- CPU有两个引脚 `LINT[1:0]` 是用来接收中断的，这两个引脚接到了**local APIC**
  - **Enable** local APIC的时候
    - CPU的 `LINT[1:0]` 可以通过APIC的本地向量表**LVT（Local Vector Table）来编程**
      - **LVT详见下文**
  - **Disable** local APIC的时候
    - these pins are configured as **INTR** and **NMI** pins
      - INTR —— 可以屏蔽的中断，一般就是**外部中断**
        - **中断向量号**一般会通过**总线**给过来
      - NMI（non-maskable interrupt） —— 不可mask的中断，**CPU必须无条件响应**
        - 触发方式
          - 对应的中断向量号是**2**，触发它的方式可以说有3种
            1. 外部中断直接assert NMI pin
            2. 处理器在**system bus or serial bus**中收到message with a delivery mode NMI —— 系统总线上有message，**且投递mode是NMI**
            3. 此外，还可以给INTR pin issue一个向量号为2的中断，进而触发**中断向量号2**对应的中断处理函数，但是同样的（类似int 2），其它硬件动作无法触发

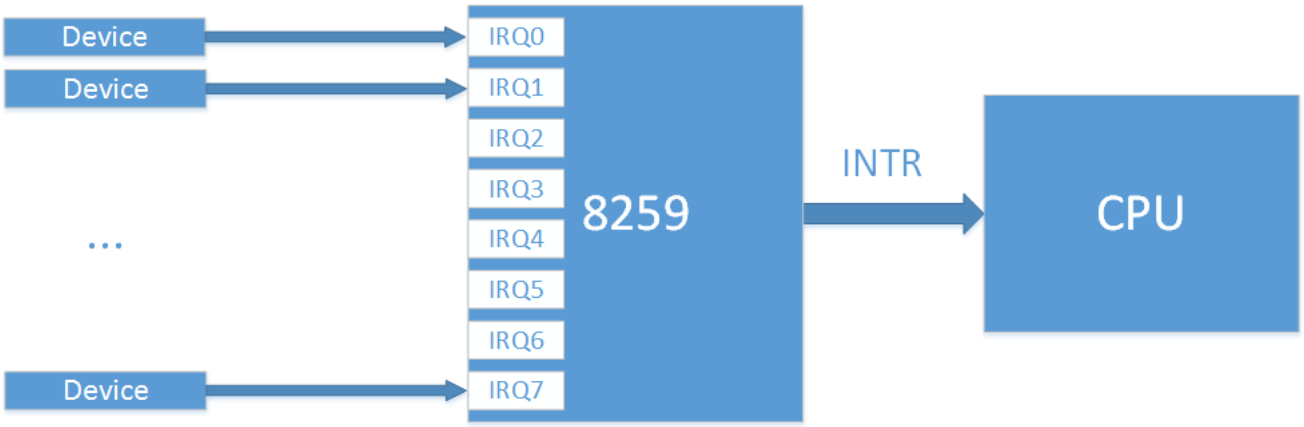
        - 这三种均会导致中断向量2对应的中断处理函数被执行，但是只有前两种会触发特定的硬件行为（NMI应该执行的）
            - **第三种无法触发NMI应该执行的硬件行为**
    - 那么如果多NMI应该如何处理呢
        - While an NMI interrupt handler is executing, the processor **blocks** delivery of subsequent NMIs until the next execution of the IRET instruction
            - CPU会防止下一个中断的进入
        - 这个**Block**能够**防止**NMI中断处理函数的**嵌套执行**，这个太TM重要了
            - 实际上更高优先级的中断是能够进入的，但是并**不会导致嵌套执行**

了解一下x86中断的发展历程

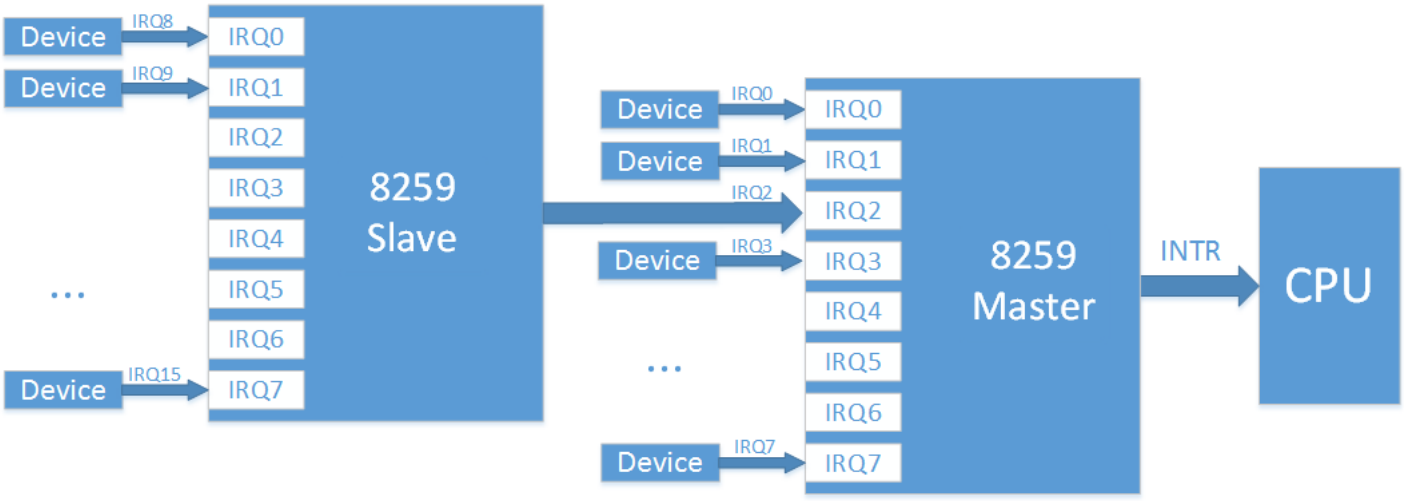
- ref
    - https://www.cnblogs.com/aalan/p/15906407.html
        - 中文翻译
    - https://habr.com/en/articles/446312/
        - 英文原版



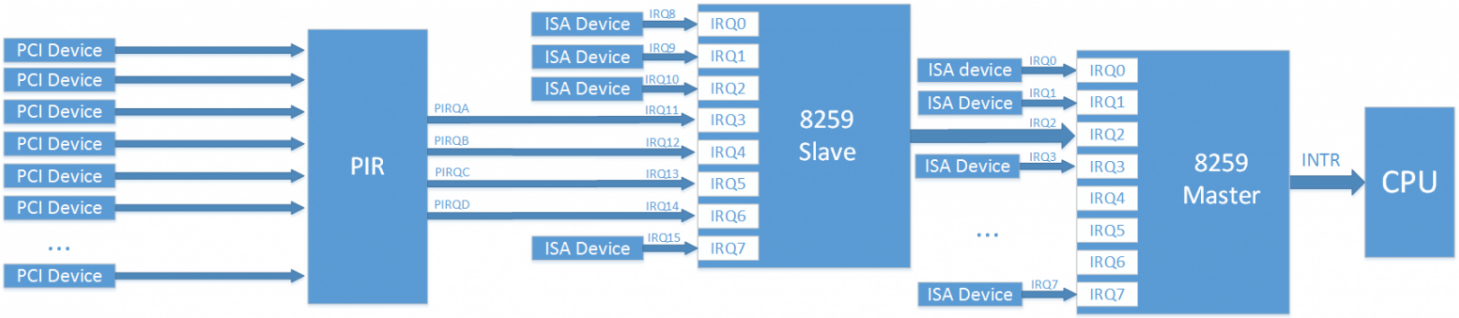
- CPU有很多外设，不可能将每个外设的中断线都连接到CPU上（这个通信效率确实高），为了解决这个问题，PIC（Programm Interrupt Controller）被设计出来



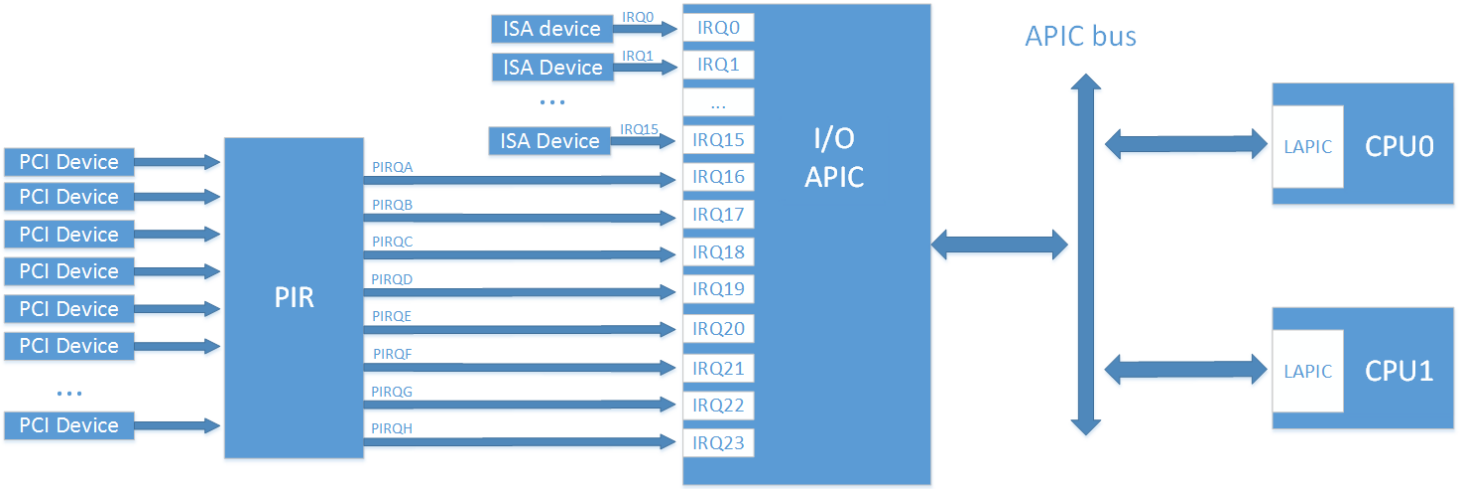
- 第一个中断控制器是8259A



- 很快，一个8259A很快不够用了，所以可以级联，这样对外我就有了**15条中断线**



- PCIe出来之后就变得更加复杂了，但这不是我关注的重点，所以略过



- 这里是质变的飞跃，APIC取代PIC，这也是UP到MP的变革，这也是我们所关注的架构，下文详细描述

- MSI以及MSI-X是如何投递中断message的

- 后期我理解了，**PCIe的写事务直接写的是LAPIC的寄存器**，真tm有意思啊

Chapter 6 —— INTERRUPT AND EXCEPTION HANDLING

- Software can also generate interrupts by executing the **INT n** instruction
  - 比如说**int 80**指令触发的就是**中断向量号80**所对应的中断处理函数
- **处理器**使用**中断向量号**作为对应**中断handler**在 **IDT(Interrupt Descriptor Table)** 中的 **index**
  - **IDT表由IDTR寄存器索引**，在保护模式下，可以存放在内存的任何位置
- Intel中断向量号的范围是是**0-255**，**0-31**是保留给系统的
  - **32-255**是用户定义的中断，通常是留给**外部IO设备的**
- CPU能收到的中断有**两种** —— **外部中断**/软件产生的中断（**内部中断**）
- **Local APIC**一般连接到**system-based I/O APIC**
  - **外部中断**会给到**IO APIC**，然后**IO APIC**将中断**重定向**到**Local APIC**
    - **重定向**可以通过**system bus（Intel Xeon）也可以通过APCI serial bus**
- 软件产生的中断
  - **int n**指令可以产生中断，例如执行**int 35**指令就会执行中断向量号35所对应的中断处理函数，这是个**软件中断**，是一种**内部中断**
    - `0-255` 都可以做这个指令的参数，但是也有差别，例如对于 `int 2` ，中断向量2对应的中断处理函数会执行，但是正常有NMI中断的时候，除了中断处理函数会执行外，硬件的一些行为还会激活，但是**int 2**并不会激活对应的**硬件行为**

一个重要的主题 —— ENABLING AND DISABLING INTERRUPTS

- **处理器**会根据**CPU的状态**以及**EFLAGS寄存器中的IF(Interrupt Enable Flag)以及RF(Resume Flag)来抑制某些**中断的产生
  - **IF**被**clear**的时候，能够禁止**可屏蔽中断**
    - 这个**clear**是在**EFLAGS寄存器被保存在stack**上**之后**做的
  - **IF**被**set**的时候，中断是**open**的
- IF标志不会影响NMIs，IF flag会受以下操作的影响
  - When an interrupt is handled through an interrupt gate, **the IF flag is automatically cleared**, which disables maskable hardware interrupts. (If an interrupt is handled through a trap gate, the IF flag is not cleared.)
    - **看起来禁中断是自动的，同类中断是不会再进来**，这可以理解为CPU的自我保护机制，**但是设备是否还会继续发送呢？** —— 后续学习
    - 这里需要看一下中断门（interrupt gate）与陷阱门（trap gate）的区别
      - **Trap Gates** and **Interrupt Gates** are **similar**, and their descriptors are structurally the same, differing **only** in the **Gate Type field**
      - The difference is that for Interrupt Gates, interrupts are **automatically** disabled upon entry and reenabled upon IRET, whereas this does not occur for Trap Gates
- IF不会影响**non-maskable中断**和**软中断(INT n)的响应，不影响异常**
  - 例如系统调用不会受该bit的影响，说白了**异常，系统调用这些优先级都高**
- 第一次可能忽略了一个重要的**topic**，那就是**谁会影响IF flag**
  1. As with the other flags in the EFLAGS register, the processor **clears the IF flag** in response to a hardware reset
     - 即硬件reset的时候，处理器会clear IF，这个不是我所关心的内容
  2. 有两条指令可以**显示**的**修改IF flag**
     - **STI (set interrupt-enable flag)** and **CLI (clear interrupt-enable flag)**
  3. 在任务切换的时候，EFLAGS是作为寄存器现场被保存在stack中，这个时候是可以直接修改的，随后在现场恢复的过程中，修改的if flag就会被set到EFLAGS寄存器中
  4. 还有就是中断触发的时候会**自动clear**

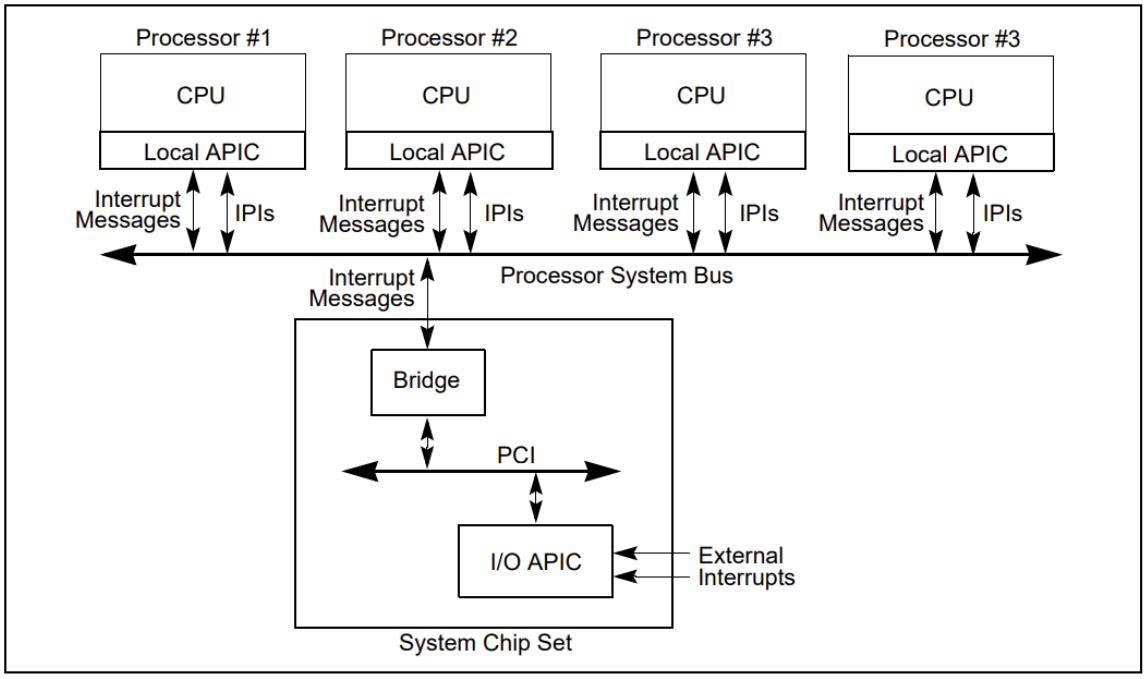


Figure 10-2. Local APICs and I/O APIC When Intel Xeon Processors Are Used in Multiple-Processor Systems

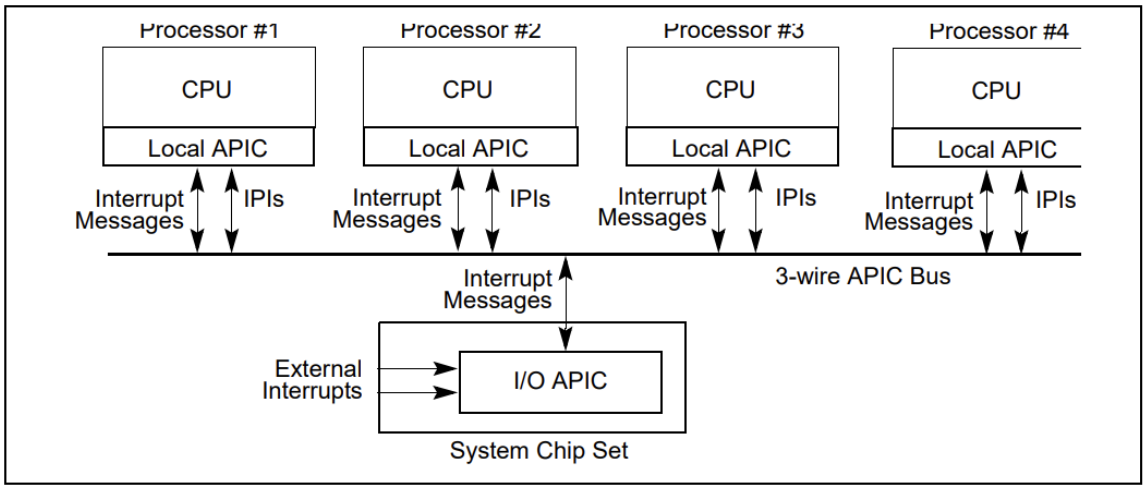
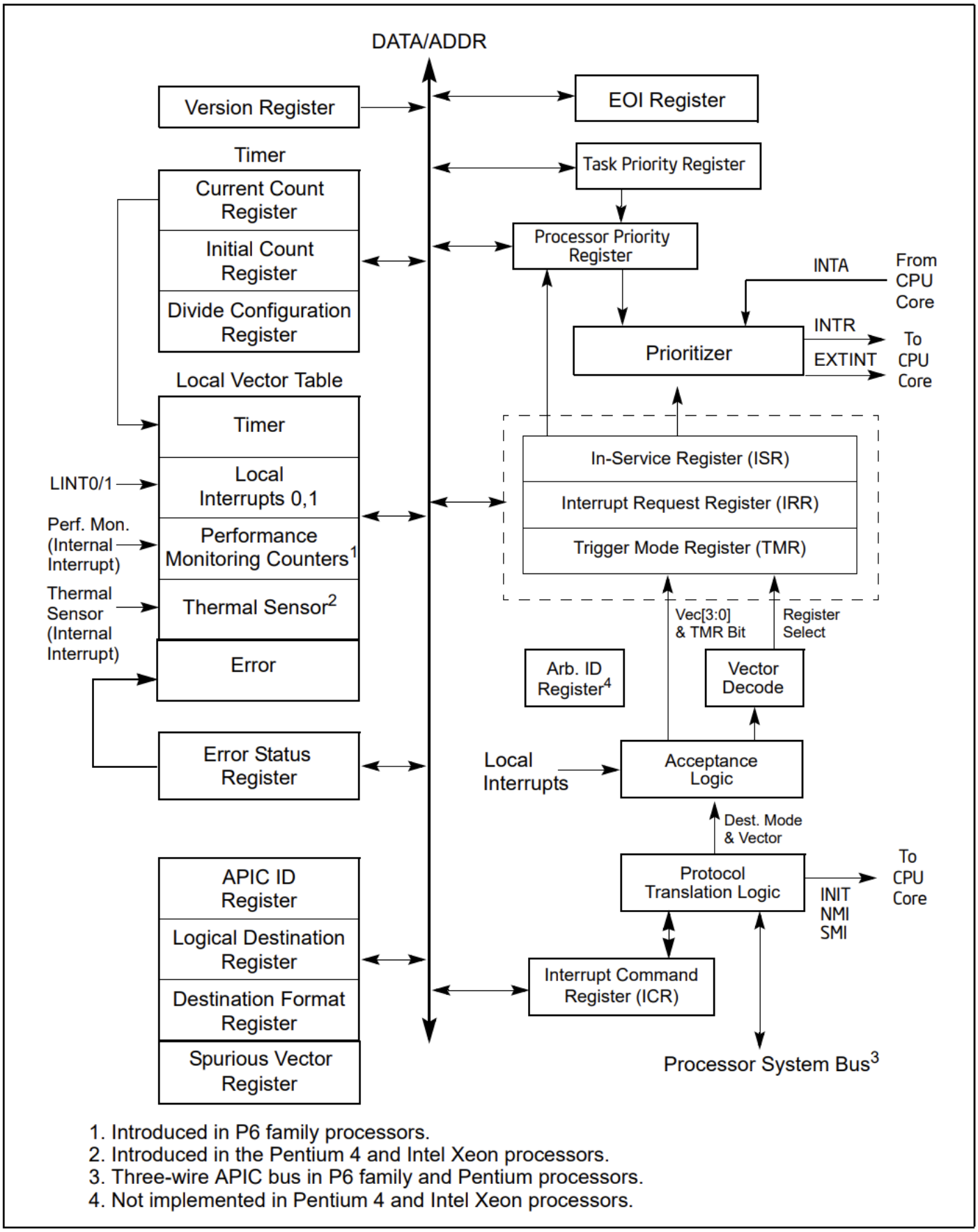


Figure 10-3. Local APICs and I/O APIC When P6 Family Processors Are Used in Multiple-Processor Systems

- 可以看到距离CPU最近的是Local APIC，这部分主要介绍的也是Local APIC



1. Introduced in P6 family processors.
2. Introduced in the Pentium 4 and Intel Xeon processors.
3. Three-wire APIC bus in P6 family and Pentium processors.
4. Not implemented in Pentium 4 and Intel Xeon processors.

Figure 10-4. Local APIC Structure

```
root@zc-System-Product-Name:/home/zc# cat /proc/iomem
##### ... #####
fee00000-fee00fff : Local APIC
  fee00000-fee00fff : Reserved
##### ... #####
```

- Local APIC的寄存器被map到起始地址为 `0xFEE00000` 的地址空间
- 但是在MP中，Local APIC有多个，但是这个4k的pa看起来只有一个，所以是有一个重定向的过程
  - The Pentium 4, Intel Xeon, and P6 family processors permit the starting address of the APIC registers to be **relocated** from 0xFEE00000H to another physical address by modifying the value in the 24-bit base address field of the IA32_APIC_BASE MSR. This extension of the APIC architecture is provided to help **resolve conflicts** with memory maps of existing systems and to allow individual processors in an MP system to map their APIC registers to different locations in physical memory
  - **反正这个是local APIC寄存器的地址**

- 单核时代，PIC（8259A）就可以，PIC的级联能够提供更多的中断引脚
- APIC是应对多核的，用于取代老旧的8259A PIC，每个core拥有自己的local APIC，整个机器有**一个或多个IO APIC**，设备的中断信号先到IO APIC，然后再到local APIC
- 自90年代以来，PCI总线发展出了MSI (Message Signalled Interrupt)，目前的机器中是以MSI为主要的中断机制，IOAPIC作为辅助，但CPU处仍使用Local APIC接收和处理中断
  - **MSI/MSI-X不过IO APIC了**
- Each local APIC consists of a set of APIC registers，The APIC registers are **memory mapped** and can be read and written to using the **MOV** instruction
  - **MSI/MSIX投递的地址**
- Local APIC（以下简称APIC）可以从以下来源接收中断
  1. Locally connected I/O devices —— 与LINT0和LINT1这两个引脚直接相连的IO设备
  2. 通过IOAPIC接收的外部中断，以及通过MSI方式收到的外部中断
     - **IOAPIC与MSI是两种机制 —— 这个要牢记**
  3. 核间中断IPI
  4. APIC Timer产生的中断
  5. Performance Monitoring Counter产生的中断
  6. 温度传感器产生的中断
  7. APIC内部错误引发的中断
- 对于以上的这些**中断源**
  - **1/4/5/6/7**称为**本地中断源**，这些中断不会指定**中断向量号**，所以需要**LVT（Local Vector Table）**
  - **IPI/IOAPIC/MSI(-X)** —— **IPI**与**IOAPIC/MSI(-X)是两种**中断，Loacl APIC有IPI message处理功能，**IPI**以及 **IOAPIC/MSI(-X)** 所投递的外部中断均由该机制处理
    - **IPI**：CPU写自己local APIC的ICR（**Interrupt Command Register**）寄存器就会在总线上产生一个IPI message，ICR寄存器中有一个重要的域是**Destination**
    - **IOAPIC/MSI(-X)**：本质与IPI相同，均为message

- 每一个Local APIC都有一个**独一无二的APIC ID**来标识自己

- 在Local APIC寄存器**spurious-interrupt vector register**中，有一个bit位代表disable or enable，说白了就是有**开关**提供给软件可以**关闭local APIC**

处理本地中断 —— LVT —— Local Vector Table

- 本地中断不会指定中断向量号，所以需要**LVT表**介入，check一下LVT中都保存了什么
- 上文提到本地中断源有5种，对于每一种中断源，LVT中都有一个**32bit的entry**
  - **0-7**位表示**中断号**，即CPU收到的中断向量号

ISSUING INTERPROCESSOR INTERRUPTS

- CPU写自己local APIC的ICR（**Interrupt Command Register**）寄存器就会在总线上产生一个IPI message，ICR寄存器中有一个重要的域是**Destination**
- ICR可以有如下用法 —— 就是用来发**核间中断**的
  - To send an interrupt to another processor
  - To allow a processor to **forward** an interrupt that it received but did not service to another processor for servicing

- To direct the processor to interrupt **itself** (perform a self interrupt)
- To deliver **special IPIs**, such as the start-up IPI (SIPI) message, to other processors

HANDLING INTERRUPTS —— 这部分异常重要

1. local APIC在总线上看到message后，首先判断自己是不是target，如果自己不是target，直接**discard message**，否则**accept message**
2. 如果自己是target，且如果中断是NMI/SMI/INIT/ExtINT/SIPI，直接交给CPU处理
3. 如果不是步骤2中的中断，如外设的中断，则**the local APIC sets the appropriate bit in the IRR(Interrupt Request Register)**
   - IRR寄存器共计**256 bit**，每**一位**代表一个**中断向量**
4. 对于在IRR中pending的中断，处理器基于优先级，**一次给CPU分发一个**
5. 一个中断被发送给CPU处理后，什么时候结束呢，中断完成routine会写local APIC中的 **EOI(end-of-interrupt)** 寄存器
   - The act of writing to the EOI register causes the local APIC to delete the interrupt from its **ISR(In-Service Register)** queue and send a message on the bus indicating that the interrupt handling has been completed

- 每一个中断向量是**8-bit**，其中高4位称为interrupt-priority class，所以根据优先级，中断向量一共有**16个group**，每一个group包含16个中断向量
  - **0-15**优先级最高，依次向后递减，IO外设的中断向量都到**160/170/180**之后了
- Local APIC中有两个寄存器**TPR(task-priority register)和PPR(processor-priority register)**，决定中断处理的顺序
  - The processor-priority class **determines** the **priority threshold** for **interrupting the processor**. The processor will deliver only those interrupts that have an interrupt-priority class higher than the processor-priority class in the PPR
    - If the processor-priority class is 0, the PPR does not inhibit the delivery any interrupt —— **all enable**
    - if it is 15, the processor inhibits the delivery of all interrupts —— **all disable**
  - The processor-priority mechanism does not affect the delivery of interrupts with the NMI, SMI, INIT, ExtINT, INIT-deassert, and start-up delivery modes
- IRR与ISR均为**256 bit寄存器**
  - The 256 bits in these registers represent the **256 possible vectors**
- IRR中保存的是**已经被accept**，但是**还未被dispatch**的**active**的interrupt request，当CPU准备接收下一个中断的时候，将IRR中优先级最高的中断取出（clear corresponding bit in IRR），然后给到ISR（set corresponding bit）表示CPU正在服务的中断请求
- While the processor is servicing the highest priority interrupt, the local APIC can send additional fixed interrupts by setting bits in the IRR. When the interrupt service routine issues a write to the EOI register, the local APIC responds by clearing the highest priority ISR bit that is set. It then repeats the process of clearing the highest priority bit in the IRR and setting the corresponding bit in the ISR. The processor core then begins executing the service routing for the highest priority bit set in the ISR
  - 重复执行上面这个流程就能够消化**IRR中pending的中断**了
- 这里将要涉及到我曾经**最疑惑**的地方了，解惑的感觉**真棒** —— 还是官方文档最牛逼
  - 如果对于同一个中断向量，local APIC可以set IRR与ISR的bit，所以 This means that for the Pentium 4 and Intel Xeon processors, the **IRR and ISR** can **queue two** interrupts for each interrupt vector
    - one in the **IRR** and one in the **ISR**
  - Any additional interrupts（**2个以上**）issued for the same interrupt vector are **collapsed** into the single bit in the IRR
    - **折叠，折叠，这个词的意思是折叠**，这么一想确实没有问题哦，中断无非是告诉你硬件的cq有内容了（举例），所以**折叠**无可厚非
      - **但是这个地方不同的处理器的处理是不同的，理解其中一种先**
- 紧跟着又是**下一个疑惑了很久**的事情，这部分内容真tm牛逼 —— **还是写英文合适一些**
  - If the local APIC receives an interrupt with an interrupt-priority class **higher** than that of the interrupt currently in service, and **interrupts are enabled in the processor core**, the local APIC dispatches the higher priority interrupt to the processor immediately (without waiting for a write to the EOI register). The currently executing interrupt handler is then interrupted so the higher-priority interrupt can be handled. When the handling of the higher-priority interrupt has been completed, the servicing of the interrupted interrupt is resumed
  - 但是我草，不是说EFLAGS中的IF会被clear掉么，这里被clear掉的时候，中断不是被禁止的么？—— 又是一个**大大的疑惑**
    - CPU通过gate进入中断处理程序，会自动clear if，这个看起来确实是这样的
    - 但是这句话其实是有一个**前提**的，**那就是interrupt是enabled的**，但是这个enable指的是什么呢？
- 下面描述一些关于**中断优先级**的问题，**并非intel官方文档**，但是需要理解
  - 先看一段描述
    - 就目前而言，中断是可以被更高优先级的中断所中断的（**这已经是成熟的解决方案了**），所以中断处理是有嵌套的，不考虑优先级，仅考虑嵌套，是会有问题的
      - 如果CPU检测到了异常，那么CPU应该立刻来处理这个异常，当CPU在处理这个异常的时候，如果有键盘的interrupt，CPU并不应该去处理键盘，而应该继续处理异常
    - 看起来将中断设计成FIFO能够解决这个问题，就是一个一个串行的来处理，但是又会引入新的问题
      - CPU在处理键盘的中断，但是这个时候发生了异常，如果CPU继续处理键盘的中断的话，那可能就会引起灾难性的后果
    - 所以就有了**中断优先级**的概念

    - 高优先级的中断可以中断低优先级中断的处理，中断进入的时候EFLAGS寄存器中的if确实也会被clear，**小小的脑袋大大的疑惑**，这个问题可以先**理解一波**了
        - ref
            - http://blog.chinaunix.net/uid-23629988-id-322333.html
                - Linux内核不支持中断嵌套（x86）？
            - https://cloud.tencent.com/developer/article/1517851?from=15425&areaSource=102001.1&traceId=nSHtdQ2aZDX4H9wKiiLas
                - Linux的中断可以嵌套吗?
        - 首先看一下intel手册中的描述
            1. 先把**异常，系统调用**这些排除出去，它们不受EFLAGS中IF的影响。即CPU在处理中断的过程中，if flag肯定是clear掉的，即**禁中断**
            2. 然后如果有高优先级中断（还是外部IO中断）到来，且CPU的中断是enable的，local APIC才会dispatch该中断给CPU
                - 这里我理解CPU中断的enable就是if flag，而if flag在处理中断的时候会被自动clear，所以只有在中断处理函数中重新set，才可能真的发生中断嵌套
        - 上面说的是CPU，下面我们继续说Linux内核在x86上的表现
            - 其实早期的linux确实是支持中断嵌套的，即如果在申请中断irq的时候带一个**flag=RQF_DISABLED**，那么这意味着在该中断的处理过程中，允许其它中断嵌套
                - 能够支持嵌套的原因就是在中断处理函数中，如果flag带了RQF_DISABLED位，那么实际上是软件主动将if flag重新set
            - 但是这个标志在2010年的某次commit中就已经作废了，在中断处理函数中再也不会手动set if flag了，所以中断发生后，硬件会屏蔽CPU对中断的响应（异常/系统调用等还是可以进来的），直到中断处理函数把这个事情了了
        - 浅看了一下 `/proc/interrupt` 这个文件，现代PCIe大都使用 `MSI/MSI-X` ，不同的中断向量会给到不同的CPU，除了**异常以及系统调用**这些，一个CPU上一般不太会再需要响应其它的外部IO中断（**这仅仅是一般情况哈**）

- 到此为止，intel官方手册中涉及的内容就总结这么多

深入分析Linux内核源码中有介绍中断，但是看起来比较老，留着以做参考

- ref
    - http://www.kerneltravel.net/book/book/第三章中断机制.pdf

老狼中断系列tips

- ref
    - https://zhuanlan.zhihu.com/p/26464793
    - https://zhuanlan.zhihu.com/p/26524241
    - https://zhuanlan.zhihu.com/p/26647697
- MSI支持32个中断，而MSI-X将其扩展到了2048个。当一个PCI设备想发送中断时，它会向其PCI配置空间Capability结构中的Message Address的地址（通常是0xFEExxxxx）写Message Data数据，消息会被该设备连接的root complex转给LAPIC，**而不需要通过IOAPIC中转。**Message Address中和IOAPIC类似，**也有目标APIC ID**，而Message Data中同样有Vector的数目。这样从电气机械的角度，MSI减少了对interrupt pin个数的需求。从而使得连接变得更简单，而且据研究，其相比INTx,有时能提高1/3的效能
    - 详见MSI/MSI-X
- 在8086时代，中断向量表位于**内存0地址**处
- `cat /proc/interrupts` 可以check被分配的中断向量号
    - **对于x86就是0-255**
- PCIe的配置空间中会有两个域，这两个PCIe插在slot的时候就定好了，不要与MSI/MSI-X搞混了
    - Interrupt Pin/Interrupt Line

MSI/MSI-X —— MESSAGE SIGNALLED INTERRUPTS

- 无论是MSI还是MSI-X，它完全取决于硬件，所占资源也位于配置空间或BAR空间
    - MSI均位于配置空间

    - MSI-X位于配置空间与BAR空间
- 但是PCIe设备在提交MSI/MSI-X中断请求时，**都是向MSI/MSI-X Capability结构中的Message Address的地址写Message Data数据**，从而组成一个**存储器写TLP**，向处理器提交中断请求，Local APIC会check是否接收这个message

*	MSI	MSI-X
中断向量数	32	2048
中断号约束	必须连续	可以随意分配
MSI信息存放	capability寄存器	MSI-X Table(BAR空间)

- MSI信息存放的位置其实都是PCIe设备自己的内部存储空间，这里与DRAM是没有半毛钱的关系的
- 首先msi也是可以注册多个中断号出来的，这个我之前 `msi/msi-x` 的理解是有偏差的，这部分需要补充一下
    - 大家都可以使用接口 `pci_alloc_irq_vectors` 来分配中断号（这个资源是硬件（**硬件设计过程中可以调整**）给的，如果超过硬件的上限，那么就会报错 [Errno 28] No space left on device）
- MSI的多中断号搞定之后，忽然发现qemu不支持多中断号，所以炜杰就去探索了一下，并且探索出了一个结果
    - `HBA_PARAMETER_1 += -device intel-iommu,intremap=on,caching-mode=on`
        - 重点是intremap参数，其他参数是为开启这项参数服务的
        - 想要在qemu中使用多个msi中断向量，需要使用intel-iommu的中断重映射
        - 重点是要开启iommu中的intremap, 除了宿主机上需要开启iommu,还有两个地方需要配置
        - qemu启动参数配置中需要令intremap=on, 编译虚拟机的内核镜像时也需要开启IOMMU
            - `Device Drivers->IOMMU Hardware Support->Support for Interrupt Remapping`
    - 与中断虚拟化有关系 https://lore.kernel.org/kvm/20200106154055.294322d0@w520.home/

实际项目开始之后MSI-X的补充

- 忽然有一天需要care一下MSI-X中断的细节了，主要是详细了解两张表的初始化过程，**MSI-X Table**以及**PBA Table**

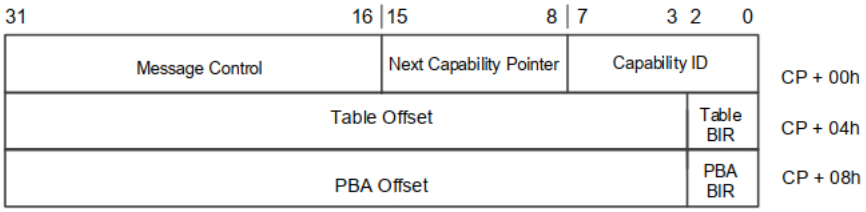


Figure 7-54: MSI-X Capability Structure

DWORD 3	DWORD 2	DWORD 1	DWORD 0		
Vector Control	Msg Data	Msg Upper Addr	Msg Addr	entry 0	Base
Vector Control	Msg Data	Msg Upper Addr	Msg Addr	entry 1	Base + 1*16
Vector Control	Msg Data	Msg Upper Addr	Msg Addr	entry 2	Base + 2*16
...	...	...	...	...	...
Vector Control	Msg Data	Msg Upper Addr	Msg Addr	entry (N-1)	Base + (N-1)*16

A-0384

Figure 6-11: MSI-X Table Structure

63 0		
Pending Bits 0 through 63	QWORD 0	Base
Pending Bits 64 through 127	QWORD 1	Base + 1*8
...	...	...
Pending Bits ((N-1) div 64)*64 through N-1	QWORD ((N-1) div 64)	Base + ((N-1) div 64)*8

A-0385

Figure 6-12: MSI-X PBA Structure

- 这是bar空间中关于MSI-X的描述图，其中**MSI-X table**与**PBA table**均位于bar空间中，其中**MSI-X table**有host配置，**PBA table**对Host是**只读**的
  - 其中配置空间中的**Table BIRTable Offset**与**PBA Offset**能够指向对应的**MSI-X table**与**PBA table**

1. MSI-X Table

- 由多个Entry组成，每个Entry对应一个中断请求
  - **Vector Control**表示的是控制信息
  - **Msg Data**表示的是**TLP写事务要写的数据**
    - data中包含中断向量号
  - **Msg Upper Addr + Msg Addr**表示的是**TLP写事务要写的地址**
    - 这个地址说白了就是每一个CPU对应的local APIC（高级可编程中断控制器）的**寄存器**
- **PCIe触发中断就是PCIe设备会根据MSI-X Table生成一个写事务去写CPU的本地中断控制器**

2. MSI-X Table的初始化

- 在驱动初始化的申请中断阶段会写这张表（**32bit的写**）
  - **详见下文驱动初始化过程中的中断申请**

3. MSI-X Table的更新

- 内核包括驱动基本**不会**主动更新这张表（内核里其实没看到）
  - 内核里本身也**没有**一些负载均衡类的任务在时刻监听不同中断在CPU上的负载情况

- 反而是这张表的更新接口直接**暴露给了用户态**
  - 用户态可以指定某个中断由某个CPU来处理
  - **所以用户态会有一些irq balance类的软件**，**MSI/MSI-X**这些他**都能balance**
- 这里简单描述一下用户态的操作以及设置亲核性的具体流程
  - **操作流程**
    - set the IRQ affinity (cpu 2 in my case) via echo 02 > `/proc/irq/$IRQ/smp_affinity`
  - 代码的执行流 —— 实际上就是修改PCIe BAR空间中的MSI-X Table
    - 算了，细节不看了，结论是没有改变的，这里确实有修改PCIe设备的MSI-X表
    - **但是基本上每个CPU大家都有份，均衡啥呀，有啥好均衡的**

4. PBA Table

- MSI-X Table有多少项，PBA Table就有多少bit，每一个bit与每一个entry相对应。**host纯只读**
- Host可以屏蔽某个entry对应的中断，这个时候硬件如果有尝试发中断，**那么对应pending bit就会置1**
  - 当然中断也发不出来哈，**那个写事务包发不出来的**
- 当软件unmask掉该中断（**即允许该中断发出来**）后，硬件依然会把这个中断再发回来，并且将相应的pending bit置0
- 如果host unmask了一个中断，并且硬件有尝试发送该中断，那么该中断发送不出来，且pending bit被置位1
  - 这种情况下，如果这个中断事件在host以某种方式被满足了，**那么硬件还得把这个pending bit清理掉**，避免中断被重新允许后，有虚假中断被发出来
- 这个pending位还有一种用法 —— **但是内核里没见过这样做的**
  - 他可以用来实现**纯粹的轮询**机制
  - 将所有的中断都无限期mask掉之后，软件不断轮询pending bit，然后发现置位1之后，开始处理中断事件，中断事件处理完毕之后，通过某种方式告知硬件，这个时候硬件会将pending bit置0直到下一次中断到来

中间间隔了一段时间，我才重新回来看中断相关的内容，所以我需要把之前最关键的内容在脑海中强化一波

1. 中断向量号 `0-255` ，这个是需要向系统申请的
2. local APIC控制中有两个关键寄存器**IRR(Interrupt Request Register)/ISR(In-Service Register)**，这个寄存器都是256bit的，**每一个bit对应一个中断向量号**
3. CPU一次只能处理一个中断请求，一旦有中断在处理，EFLAGS寄存器中的IF flag会被clear，那么只有不受IF flag影响的中断才会进来（非常高优先级的中断）
4. 这里还有一个关键点，local APIC中的IRR/ISR寄存器中只有256bit，即一个中断向量号只有一个bit与之对应，假设CPU在处理198号中断的时候，又有198号中断到来（甚至连续到来若干个），那这个时候中断会漏掉么，并不会，x86的某些CPU会将中断向量**折叠**，只要能触发中断处理函数，就需要软件介入，软件介入就会涉及到硬件寄存器，所有的事情都会处理掉
5. 最后说到了linux内核是否支持中断嵌套，从CPU的手册看来，中断确实是有可能嵌套的（**这里只考虑可屏蔽的外设的中断**），即不受IF flag影响的高优先级中断不考虑
   - 在linux kernel老一些的版本中，驱动开发者可以为自己的中断号指定一个参数，CPU在进入中断处理流程后，这里会主动set EFLAGS寄存器的if flag，所以CPU可以继续响应高优先级的中断，但是更新的内核直接把这个干掉了，**所以在一个core上可屏蔽中断是不会嵌套执行的**
   - 这里可以说说linux kernel最开始的样子， `2.6.35` 之前的内核认为中断处理函数有两种，slow handler/fast handler，对于fast handler，在CPU处理中断过程中，全程关中断，因为CPU处理这个中断很快，但是还有一个slow handler，这里认为该中断处理函数是耗时的，有可能miss其它紧急的中断，所以这里就允许中断的进入，这里就会造成**中断的嵌套**
   - 但是随着技术的发展
     - 硬件方面，CPU越来越快
     - 软件方面，linux kernel提出了**上半部**与**底半部**的机制
   - 这些都可以保证中断能够快速完成，所以在中断处理中允许中断的嵌套（只考虑可屏蔽中断）以及不再被需要。更深的嵌套栈对软件也是一种挑战，所以现在linux kernel在处理中断的过程中，**CPU全程关中断**
6. **MSI或MSI-X的中断不需要IO APIC，但是依然需要local APIC**

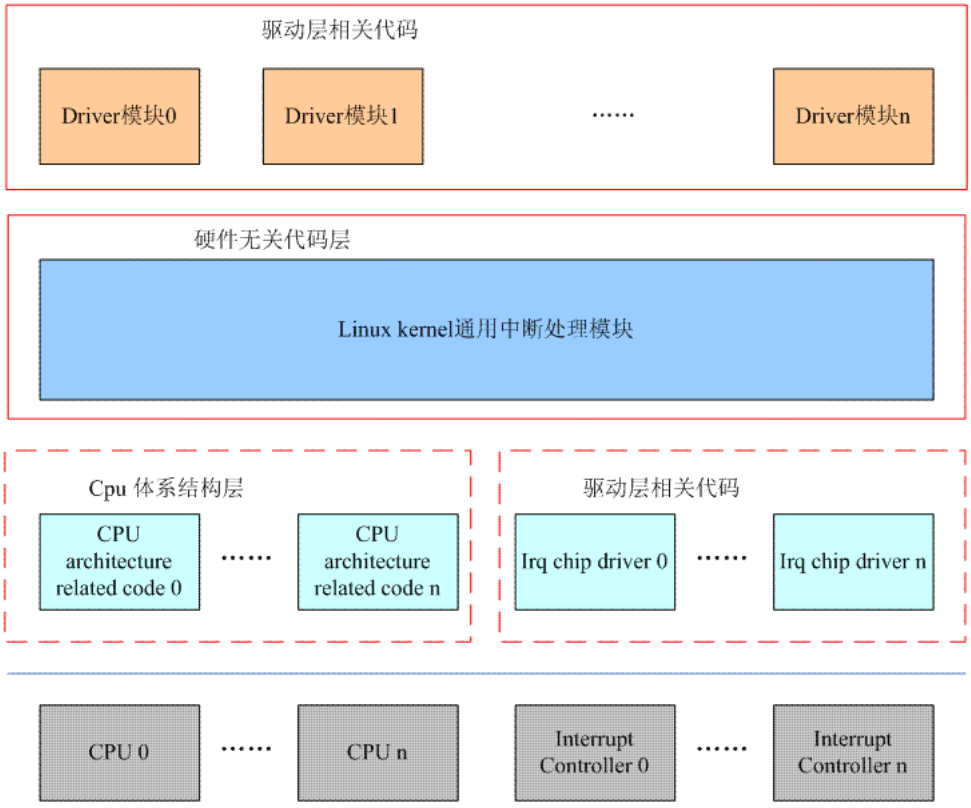
蜗窝科技的中断子系统概述

- ref

    - http://www.wowotech.net/irq_subsystem/interrupt_subsystem_architecture.html

- 外设类型的中断被送到**一个CPU**上就OK了，一件事被**投递一次**就可以了
- 0-255这256个中断号实际上是一个**虚拟的中断向量号**，与硬件无关，**仅仅是被CPU用来标识一个外设中断**
    - 更多还是在描述OS内的内容，即**软件部分**
- 第五部分 —— 驱动申请中断API
- 第八章 —— softirq
- 第九章 —— tasklet

准备结合linux内核代码淦一波中断 —— 主要是看pcie相关的



- 在qemu中， `24-28` 是给nvme分配的中断号，qemu模拟的是4核smp架构，所以nvme这里使能了4对队列以及外加一个admin队列，所以一共需要向系统申请5个中断号，在多次重启中，这个号基本不会变，就在认为他不变的基础上进行debug

```
       CPU0       CPU1       CPU2       CPU3
24:         10          0          0          0   PCI-MSI 65536-edge      nvme0q0
25:          0          0          0          0   PCI-MSI 65537-edge      nvme0q1
26:          0          1          0          0   PCI-MSI 65538-edge      nvme0q2
27:          0          0          0          0   PCI-MSI 65539-edge      nvme0q3
28:          0          0          0          0   PCI-MSI 65540-edge      nvme0q4
```

- 通过debug nvme驱动 `nvme_irq` 的调用栈，基本能够确定软件这里的入口是 `DEFINE_IDTENTRY_IRQ(common_interrupt)`

```
/*
 * common_interrupt() handles all normal device IRQ's (the special SMP
 * cross-CPU interrupts have their own entry points).
 */
DEFINE_IDTENTRY_IRQ(common_interrupt) {/* ... */}
/* 展开DEFINE_IDTENTRY_IRQ，结果如下，所以DEFINE_IDTENTRY_IRQ(common_interrupt)相当于定义了2个函数common_interrupt以及__common_interrupt */
static void __common_interrupt(struct pt_regs *regs, u32 vector);
__visible noinstr void common_interrupt(struct pt_regs *regs, unsigned long error_code)
{
        irqentry_state_t state = irqentry_enter(regs);
        u32 vector = (u32)(u8)error_code;

        instrumentation_begin();
        kvm_set_cpu_l1tf_flush_l1d();
        run_irq_on_irqstack_cond(__common_interrupt, regs, vector);
        instrumentation_end();
        irqentry_exit(regs, state);
}
static noinline void __common_interrupt(struct pt_regs *regs, u32 vector)
{
        struct pt_regs *old_regs = set_irq_regs(regs);
        struct irq_desc *desc;

        /* entry code tells RCU that we're not quiescent.  Check it. */
        RCU_LOCKDEP_WARN(!rcu_is_watching(), "IRQ failed to wake up RCU");

    /* 这里是一个非常关键的地方，其中vector是0-255的数，vector_irq是每CPU变量，在不同的CPU上vector相同，但是会对应到不同的irq_desc中，所以一切就合理起来了
     * 这个vector就是中断向量，最后压栈的那个数，MSI-X data中是包含这个内容的
     *
     * 到这里是什么时候的事情呢，中断刚压栈完成的时候，这是真的中断刚刚到达软件的时候
     */
        desc = __this_cpu_read(vector_irq[vector]);
        if (likely(!IS_ERR_OR_NULL(desc))) {
                handle_irq(desc, regs);
        } else {
                ack_APIC_irq();

                if (desc == VECTOR_UNUSED) {
                        pr_emerg_ratelimited("%s: %d.%u No irq handler for vector\n",
                                             __func__, smp_processor_`id(),
                                             vector);
                } else {
                        __this_cpu_write(vector_irq[vector], VECTOR_UNUSED);
                }
        }

        set_irq_regs(old_regs); /* 这个已经在恢复寄存器现场了 */
}
```

捋一下正儿八经的函数调用栈（中断）—— 上文是中断子系统初始化的过程

```
/* arch/x86/include/asm/idtentry.h */
SYM_CODE_START(irq_entries_start)
    vector=FIRST_EXTERNAL_VECTOR
    .rept NR_EXTERNAL_VECTORS
        UNWIND_HINT_IRET_REGS
0 :
        .byte   0x6a, vector /* 这个其实是进入中断上下文的最后一次压栈，将vector压栈，随后调用中断处理函数common_interrupt的时候就能够直接从stack上取得中
        jmp     asm_common_interrupt
        nop
        /* Ensure that the above is 8 bytes max */
        . = 0b + 8
        vector = vector+1
    .endr
SYM_CODE_END(irq_entries_start)
/* 上面每一个中断entry跳转的地方都是asm_common_interrupt，该函数的实现见下文 */
DECLARE_IDTENTRY_IRQ(X86_TRAP_OTHER,    common_interrupt); /* arch/x86/include/asm/idtentry.h */
/* ... */
#define DECLARE_IDTENTRY_IRQ(vector, func) idtentry_irq vector func
/* ... */
/*
 * Interrupt entry/exit. 直接就说这是中断的入口/出口
 *
 + The interrupt stubs push (vector) onto the stack, which is the error_code
 * position of idtentry exceptions, and jump to one of the two idtentry points
 * (common/spurious).
 *
 * common_interrupt is a hotpath, align it to a cache line
 */
 /* linux/arch/x86/entry/entry_64.S */
.macro idtentry_irq vector cfunc
        .p2align CONFIG_X86_L1_CACHE_SHIFT
        idtentry \vector asm_\cfunc \cfunc has_error_code=1
.endm
/* ... */
/**
 * idtentry - Macro to generate entry stubs for simple IDT entries
 * @vector:             Vector number
 * @asmsym:             ASM symbol for the entry point
 * @cfunc:              C function to be called
 * @has_error_code:     Hardware pushed error code on stack
 *
 * The macro emits code to set up the kernel context for straight forward
 * and simple IDT entries. No IST stack, no paranoid entry checks.
 */
.macro idtentry vector asmsym cfunc has_error_code:req
SYM_CODE_START(\asmsym)
        UNWIND_HINT_IRET_REGS offset=\has_error_code*8
        ASM_CLAC

        .if \has_error_code == 0
                pushq   $-1                     /* ORIG_RAX: no syscall to restart, 如果前面没有把中断号push到stack顶，那么现在push一个-1 */
        .endif

        .if \vector == X86_TRAP_BP /* 我从中断过来看到的不是BP */
                /*
                 * If coming from kernel space, create a 6-word gap to allow the
                 * int3 handler to emulate a call instruction.
                 */
                testb   $3, CS-ORIG_RAX(%rsp)
                jnz     .Lfrom_usermode_no_gap_\@
                .rept   6
                pushq   5*8(%rsp)
                .endr
                UNWIND_HINT_IRET_REGS offset=8
.Lfrom_usermode_no_gap_\@:
        .endif

        idtentry_body \cfunc \has_error_code /* 这个是函数实际调用的实体 */

_ASM_NOKPROBE(\asmsym)
SYM_CODE_END(\asmsym)
.endm
/* ... */
/**
```

```
 * idtentry_body - Macro to emit code calling the C function
 * @cfunc:              C function to be called
 * @has_error_code:     Hardware pushed error code on stack
 */
.macro idtentry_body cfunc has_error_code:req

        call    error_entry /* 这里会把剩余的寄存器push stack */
        UNWIND_HINT_REGS

        movq    %rsp, %rdi                      /* pt_regs pointer into 1st argument*/

        .if \has_error_code == 1
                movq    ORIG_RAX(%rsp), %rsi    /* get error code into 2nd argument, 因为后续调用中断处理函数时间，第二个参数是从rsi寄存器得到的，其实就
                movq    $-1, ORIG_RAX(%rsp)     /* no syscall to restart */
        .endif

        call    \cfunc /* 这里实际调用的就是common_interrupt */

        jmp     error_return
.endm
```

```
/* ... */
struct irq_data {
        u32                     mask;
        unsigned int            irq;
        unsigned long           hwirq;
        struct irq_common_data  *common;
        struct irq_chip         *chip;
        struct irq_domain       *domain;
#ifdef  CONFIG_IRQ_DOMAIN_HIERARCHY
        struct irq_data         *parent_data;
#endif
        void                    *chip_data;
};
/* ... */
struct irq_desc {
        struct irq_common_data  irq_common_data;
        struct irq_data         irq_data;
        unsigned int __percpu   *kstat_irqs;
        irq_flow_handler_t      handle_irq;
        struct irqaction        *action;        /* IRQ action list */
        unsigned int            status_use_accessors;
        unsigned int            core_internal_state__do_not_mess_with_it;
        unsigned int            depth;          /* nested irq disables */
        unsigned int            wake_depth;     /* nested wake enables */
        unsigned int            tot_count;
        unsigned int            irq_count;      /* For detecting broken IRQs */
        unsigned long           last_unhandled; /* Aging timer for unhandled count */
        unsigned int            irqs_unhandled;
        atomic_t                threads_handled;
        int                     threads_handled_last;
        raw_spinlock_t          lock;
        struct cpumask          *percpu_enabled;
        const struct cpumask    *percpu_affinity;
#ifdef CONFIG_SMP
        const struct cpumask    *affinity_hint;
        struct irq_affinity_notify *affinity_notify;
#ifdef CONFIG_GENERIC_PENDING_IRQ
        cpumask_var_t           pending_mask;
#endif
#endif
        unsigned long           threads_oneshot;
        atomic_t                threads_active;
        wait_queue_head_t       wait_for_threads;
#ifdef CONFIG_PM_SLEEP
        unsigned int            nr_actions;
        unsigned int            no_suspend_depth;
        unsigned int            cond_suspend_depth;
        unsigned int            force_resume_depth;
#endif
#ifdef CONFIG_PROC_FS
        struct proc_dir_entry   *dir;
#endif
#ifdef CONFIG_GENERIC_IRQ_DEBUGFS
        struct dentry           *debugfs_file;
        const char              *dev_name;
#endif
#ifdef CONFIG_SPARSE_IRQ
        struct rcu_head         rcu;
        struct kobject          kobj;
#endif
        struct mutex            request_mutex;
        int                     parent_irq;
        struct module           *owner;
        const char              *name;
} ____cacheline_internodealigned_in_smp;
/* ... */
static struct irq_chip pci_msi_controller = {
        .name                   = "PCI-MSI",
        .irq_unmask             = pci_msi_unmask_irq,
        .irq_mask               = pci_msi_mask_irq,
        .irq_ack                = irq_chip_ack_parent,
        .irq_retrigger          = irq_chip_retrigger_hierarchy,
        .irq_set_affinity       = msi_set_affinity,
        .flags                  = IRQCHIP_SKIP_SET_WAKE |
                                  IRQCHIP_AFFINITY_PRE_STARTUP,
};
```

```
/* ... */
static struct msi_domain_info pci_msi_domain_info = {
        .flags          = MSI_FLAG_USE_DEF_DOM_OPS | MSI_FLAG_USE_DEF_CHIP_OPS | MSI_FLAG_PCI_MSIX,
        .ops            = &pci_msi_domain_ops,
        .chip           = &pci_msi_controller,
        .handler        = handle_edge_irq,
        .handler_name   = "edge",
};
```

- irq_entries_start
  - asm_common_interrupt
    - __visible noinstr void common_interrupt(struct pt_regs *regs, unsigned long error_code) —— **struct pt_regs**这竟然是ptrace寄存器，这个我多少就有点印象了 —— 看起来是**no instrumentation**，即**不允许插桩**，所以这个函数整体是**不允许插桩的**
      - irqentry_state_t state = irqentry_enter(regs);
      - u32 vector = (u32)(u8)error_code; —— **向量号**就是传入的这个 error_code 哦
      - instrumentation_begin(); —— **以下的代码可以debug以及插桩**
      - kvm_set_cpu_l1tf_flush_l1d();
      - run_irq_on_irqstack_cond(__common_interrupt, regs, vector);
        - irq_enter_rcu();
        - func(regs, vector); —— __common_interrupt —— **寄存器的现场**与**向量号**
          - struct pt_regs *old_regs = set_irq_regs(regs); —— 这是在保存当前的**寄存器现场**么
          - struct irq_desc *desc;
          - desc = __this_cpu_read(vector_irq[vector]); —— 这个**vector**也不是传到中断处理函数中的 irq_num ，但是根据该向量号能够得到**中断描述符**，中断描述符本身就是一个非常重要的数据结构了，他包含了 irq_num 等一系列关键的数据结构，可以理解为**中断处理的核心数据结构**
          - handle_irq(desc, regs); —— **这个是中断真正的处理流程**
            - generic_handle_irq_desc(desc); —— 这是 IS_ENABLED(CONFIG_X86_64) 的**处理流程**
              - desc->handle_irq(desc); —— 一个desc的中断处理函数是自己提供的，本例中是 handle_edge_irq 函数
                - void handle_edge_irq(struct irq_desc *desc)
                  - raw_spin_lock(&desc->lock); —— 这个锁说明什么，说明在**一个计算机系统**中**任一**时刻**同一个中断号**对应的处理函数只能有一个，而且硬件本身也杜绝了同一优先级中断号的嵌套，所以根本不需要考虑中断嵌套的问题 —— 我草，等等，看到下面的时候发现他把这个锁解开了，我去
                  - **分隔符，重点关注在lock中做了什么**
                  - desc->irq_data.chip->irq_ack(&desc->irq_data); —— **Start handling the irq** —— 实际执行的函数是 irq_chip_ack_parent
                    - data = data->parent_data; —— 这尼玛是要开始**套娃**是么
                    - data->chip->irq_ack(data); —— 直接google一下看看，这个与边沿触发是有关系的，这是真tm学习了 —— http://www.wowotech.net/irq_subsystem/High_level_irq_event_handler.html
                      - 实际上上升沿意味着中断，但是一旦触发中断，上升沿会保持，中断控制器将无法检测到新的中断信号（本身是高电平，无法形成上升沿），这个高电平会一直保持到软件ack该中断（调用 irq_chip 的 irq_ack 函数），ack之后中断控制器才有可能继续探测上升沿，**触发下一次中断**
                        - **但是这个和EP硬件本身应该是没有太大的关系的**
                  - handle_irq_event(desc);
                    - irqd_set(&desc->irq_data, IRQD_IRQ_INPROGRESS);
                    - raw_spin_unlock(&desc->lock); —— 我擦，这个锁怎么又解开了
                    - ret = handle_irq_event_percpu(desc);
                      - retval = __handle_irq_event_percpu(desc, &flags);
                        - for_each_action_of_desc(desc, action)
                          - res = action->handler(irq, action->dev_id); —— 例如在nvme中就是nvme_irq
                            - 整个软件通路上确实很轻薄，没有什么太重开销的地方，重点看一下那个**ack**是在做什么呢
                          - switch (res) {
                          - case IRQ_WAKE_THREAD:
                            - __irq_wake_thread(desc, action);
                              - if (test_and_set_bit(IRQTF_RUNTHREAD, &action->thread_flags)) —— **Wake up the handler thread for this action. If the RUNTHREAD bit is already set, nothing to do**
                                - return;

- atomic_inc(&desc->threads_active);
- wake_up_process(action->thread);
- case IRQ_HANDLED:
- *flags |= action->flags;
- break;
- default:
- break;
- }
- add_interrupt_randomness(desc->irq_data.irq, flags);
- raw_spin_lock(&desc->lock); —— 然后这个锁又锁上了
- irqd_clear(&desc->irq_data, IRQD_IRQ_INPROGRESS);
- **分隔符，重点关注在lock中做了什么**
- raw_spin_unlock(&desc->lock);
- set_irq_regs(old_regs); —— **恢复寄存器现场**
- current_thread_info()->regs = new_regs;
- irq_exit_rcu();
- instrumentation_end(); —— **以上的代码可以debug以及插桩**
- irqentry_exit(regs, state);

补充一个instrumentation的知识点

- ref
  - https://lwn.net/Articles/877229/
- Non-instrumentable code - noinstr —— 禁止**插桩**的代码
  - 有noinstr标记意味着这部分代码会被放到一个特殊的位置，该位置**禁止instrumentation and debug facilities**，其实这部分规则很简单，描述如下
    - noinstr void entry(void)
      - handle_entry(); —— <-- must be 'noinstr' or '__always_inline'
      - ...
      - instrumentation_begin();
      - handle_context(); —— <-- instrumentable code —— **相当于暂时解开了限制**
      - instrumentation_end();
      - ...
      - handle_exit(); —— <-- must be 'noinstr' or '__always_inline'

在这个过程中发现了一些初始化的端倪，所以觉得先看一下中断门的初始化 —— idt_setup_apic_and_irq_gates

```
struct x86_init_ops x86_init __initdata = {
        .resources = {
                .probe_roms             = probe_roms,
        /* ... */
        },
        .mpparse = {
                .setup_ioapic_ids       = x86_init_noop,
        /* ... */
        },
        .irqs = {
                .pre_vector_init        = init_ISA_irqs,
                .intr_init              = native_init_IRQ,
                .intr_mode_select       = apic_intr_mode_select,
                .intr_mode_init         = apic_intr_mode_init,
                .create_pci_msi_domain  = native_create_pci_msi_domain,
        },
    /* ... */
        .acpi = {
                .set_root_pointer       = x86_default_set_root_pointer,
                .get_root_pointer       = x86_default_get_root_pointer,
                .reduced_hw_early_init  = acpi_generic_reduced_hw_init,
        },
};
```

- `asmlinkage __visible void __init __no_sanitize_address start_kernel(void)` —— 直接干到了 `start_kernel`，**真棒**
  - `init_IRQ();`
    - `for (i = 0; i < nr_legacy_irqs(); i++)`
      - `per_cpu(vector_irq, 0)[ISA_IRQ_VECTOR(i)] = irq_to_desc(i);`
    - `x86_init.irqs.intr_init();` —— 实际就是这个函数 `native_init_IRQ`
      - `void __init native_init_IRQ(void)`
        - `int i = FIRST_EXTERNAL_VECTOR;` —— **32**
        - `x86_init.irqs.pre_vector_init();`
        - `idt_setup_apic_and_irq_gates();` —— `static gate_desc idt_table[IDT_ENTRIES] __page_aligned_bss;` —— 看起来就是在初始化中断描述符表啊，**IDT_ENTRIES=256**
          - `idt_setup_from_table(idt_table, apic_idts, ARRAY_SIZE(apic_idts), true);`
            - `gate_desc desc;` —— 这个是**中断门**的描述符
            - `for (; size > 0; t++, size--)`
              - `idt_init_desc(&desc, t);`
              - `write_idt_entry(idt, t->vector, &desc);`
              - `if (sys)`
                - `set_bit(t->vector, system_vectors);`
          - `for_each_clear_bit_from(i, system_vectors, FIRST_SYSTEM_VECTOR)`
            - `entry = irq_entries_start + 8 * (i - FIRST_EXTERNAL_VECTOR);`
            - `set_intr_gate(i, entry);`
            - `idt_map_in_cea();` —— **Map IDT into CPU entry area and reload it**
            - `load_idt(&idt_descr);` —— **将中断描述符表的地址丢到寄存IDTR器中**
              - 在SMP架构中，目前这里仅仅是在**引导处理器**上，后续会在每一个**应用处理器**上处理
            - `set_memory_ro((unsigned long)&idt_table, 1);` —— **Make the IDT table read only**
        - `lapic_assign_system_vectors();`
- 差不多吧，具体细节干了点啥，其实我也没有很清楚

中断最核心的自问自答环节 —— 这一套流程走完中断基本就了了

- 这些问题的源头都是炜杰问我的一句话，硬件应该如何去发中断呢，能无脑一直发么，**真的是把我给难住了我草**，随后就有了一系列的探索

某些驱动中的清中断或禁中断

- 某些驱动，例如**rudy sas，hisi sas**，这些驱动在中断处理函数的第一步总是与硬件做一次交互，一般都称为**清中断**，简单理解就是硬件发出中断后，需要等待软件ack之后才能发起下一次中断
- 还有一些逻辑，比如**pm8001，xdma**等驱动，他们的中断处理流程基本都是，**禁中断——逻辑处理——开中断**，与**hisi sas**不完全相同，但是在这种类型情况下，**在中断发出到中断处理的过程中，硬件是否还能够继续发起下一个中断呢**
- qdma是什么逻辑呢
  - qdma的文档描述了一个重要的info
    - **Internal Mode Writeback and Interrupts (AXI MM and H2C ST)**
      - An interrupt will not be generated until the irq_arm bit has been set by software. Once an interrupt has been sent the irq_arm bit is cleared by hardware. Should an interrupt be needed when the irq_arm bit is not set, the interrupt will be held in a pending state until the irq_arm bit is set.
      - 最后代码改变的位置在这里
        - `int qdma_queue_pidx_update(void *dev_hndl, uint8_t is_vf, uint16_t qid, uint8_t is_c2h, const struct qdma_q_pidx_reg_info *reg`
  - qdma的上半部什么都没有做，而是在处理完本次中断后会更新一个寄存器的某一个bit，**该步骤完成之后才能够发起下一次中断**
    - 相当于硬件发出中断后，把自己阻塞了，直到软件ack本次中断并处理完本次中断后才会继续发起下一次的中断
- 反观**nvme/mpt3sas**等，无论是上半部还是下半部都不会有与**中断开关**相关的操作

边沿触发水平触发等等

- ref

- https://example61560.wordpress.com/2016/06/30/pcipcie-总线概述6/
  - 有介绍一些字段的细节**Message Address字段和Message Data字段的格式**
- https://zhuanlan.zhihu.com/p/598620943 && https://zhuanlan.zhihu.com/p/90074320
  - 这两个都比较宏观，重点是中断控制器与CPU的交互以及**handle_level_irq函数**的一些详细解析
  - 第二个帖子描述边沿触发很清晰
    - 边缘触发描述的很清楚，他说这个过程**handle_edge_irq**
      - 硬件上也会有类似的机制
    - 中断是不排队的，能pending几个，但是再多就一定会丢，那问题又来了，**硬件不应该放开发啊**
    - 对边缘触发，假设是拉高，电平被拉高后会保持（相当于**锁存**），直到ACK应答
- https://github.com/yifengyou/linux-0.12/blob/master/docs/中断.md
  - **中断讲的还可以**

- **MSI/MSI-X**一定是**边沿触发**，尽管其控制位可以选择触发方式
  - **边沿/电平**触发，触发就是再说，信号来了，什么时候采集它，最好的办法就是利用**时钟采样**
    - 边沿触发 —— 上升沿或下降沿触发，信号在该过程中变化很小，便于采集
    - 电平触发 —— 高电平或低电平的状态采样，采样后需要立刻mask该中断，否则下次采样又是一次中断
- msgdata中会有关于中断方式的描述
  - Trigger Mode字段
    - 为0b0x时，PCIe设备使用边沿触发方式申请中断
      - 为0b10时使用低电平触发方式；为0b11时使用高电平触发方式
  - **MSI/MSI-X中断请求使用边沿触发方式**
    - 但是FSB Interrupt Message总线事务还支持Legacy INTx中断请求方式，**因此在Message Data字段中仍然支持电平触发方式。但是对于PCIe设备而言，该字段为0b0x**

core-api/genericirq.rst

- 主要就让我更强化的理解了一个模型，**ep与驱动**都是各家自己的，中间无论是软件还是硬件都有一部分公共部分，比如硬件的rc，中断控制器，软件上的中断处理的 **公共部分，公共抽象**等等

https://kernel.0voice.com/forum.php?mod=viewthread&tid=1931 —— 解惑帖

- **电平触发**
  - 为什么上来就要mask中断
    - Level type interrupts are active as long as the hardware line has the active level. This may require to mask the interrupt and unmask it after the associated handler has acknowledged the device, so the interrupt line is back to inactive
      - **当中断线电平达到激活电平，中断一直会被激活。所以需要刚进中断处理函数就要屏蔽掉中断，等handler处理完后再打开中断(unmask)**
  - 为什么要**先**屏蔽中断，**后**确认呢（**mask_and_ack**）
    - 因为cpu**周期性**采样中断线，当发现电平有效时，触发中断，因此外部设备为了保证cpu可以检测到有效电平，需要将电平保持一段时间。当cpu检测到有效电平后，cpu引脚被置位，如果不先屏蔽中断，直接确认，当cpu引脚复位时，如果此时电平依然有效，那么同一个中断又被触发，此为不正确的
      - **解释的太tm的牛皮了**
    - 注：cpu确认是用于复位cpu引脚，**对外设的应答在注册的irq_action由特定的驱动程序进行** —— 这是我苦苦寻觅的内容
      - **如果说外设的每一个中断都是需要应答的，那么一切就显得很合理了，但是还是有一些不同**

- **边沿触发**
  - 之前疑惑的**锁**就有解释了，实际上在处理一个中断之前，同样的中断可能已经到了（因为在处理前ack了该中断，并且释放了对应的锁）。这个时候第二个中断必须pending
    - 如果有pending，那就应该一直调用下去
    - 同样的中断是有可能去到不同的核的，如果是这样的话，当前这个核就不应该处理这个中断
    - 即使有屏蔽的，每次也是只会多一个，不会冲进来
      - 但在这种情况下，IRQ已经被屏蔽（在设置IRQ_PENDING时同时也调用了mask_ack_irq函数将该中断屏蔽掉了）。因而必须用unmask_irq解除IRQ的屏蔽，并清除IRQ_MASKED标志，这确保在**handle_irq_event**执行期间只能发生一个中断 （被屏蔽后将不会在收到同一个编号的中断）
      - **那这种情况下，有可能有同一个中断继续到来么，会如何呢，这个阶段太多了，我擦**
    - 正常情况下，为什么不屏蔽中断，因为如果中断被屏蔽，很容易漏信号，屏蔽中断的时刻应该是越短越好
  - 边沿触发为什么不需要先屏蔽，后ack

        - 因为边沿型中断在电平的变化沿才有效，所以外设为触发中断，会在中断线上放一个边沿脉冲，一个边沿脉冲只能触发一次中断，所以不需要防止像电平型中断，同一个中断请求的高电平触发多次中断而需先屏蔽中断的情况

MSI-X在不设置亲和性的情况下真的能给所有CPU都发么？

- 先利用mpt3sas做了一些实验的说
  - 驱动模块有一个参数，**smp_affinity_enable**，这个东西可以置1，可以置0，默认是1 ——
    `MODULE_PARM_DESC(smp_affinity_enable, "SMP affinity feature enable/disable Default: enable(1)");`
    - 这个值是1的时候，每一个中断号与一个CPU相互绑定，且中断只会出现在那个核上，就是最常见的很规整的那种斜线
    - 这个值是0的时候，**中断在CPU上的分布就比较乱了**
      - 一个中断号可能会出现在不同的CPU上
      - 一个CPU上可能会有多个中断
    - 顺便比较了一下性能，**32核火力全开**，队列深度4，4k随机小写 —— **115K IOPS**
      - 重新把affinity打开看看，iops几乎没有区别，性能没有什么变化，**可能中断并不是影响它的主要因素**
  - 队列数分配的少一些试试看，现在默认分配的是CPU的数量，**现在降低试试看**，修改硬件读取回来的这个值**MaxMSIxVectors**最方便了（**insmod的过程直接挂了我草**）
    - 可以变参数选择最大msi-x的数量的，**舍近求远了**
      - 给8个msi-x，带affinity enable
        - 确实是在8个core上，绑定好了的，不会瞎跑的
      - 现在8个msi-x，然后affinity disable
        - 和32个的没啥区别看起来
    - 改成1试试啊，最大是1
      - 带affinity enable
        - 所有的中断都落到了同一个core上，这个core不会发生变化
      - 不带affinity enable
        - 是有可能在不同的核上所执行的，但是也并没有乱的很夸张，**就是分布到了2个核上**
- 但是后期我知道原因了，在设置affinity之后，无法再次修改中断号与CPU的对应关系，而不设置affinity时，**软件是能够修改该对应关系的**
- 结合 `proc/irq` 做一些实验的说 —— **mpt3sas不开启亲和性**
  - 通过观察mpt3sas的中断分布，通过fio测试发现**smp_affinity/smp_affinity_list**是会发生改变的
    - **smp_affinity是bitmap，smp_affinity_list说白了就是cpuid**，例如 `smp_affinity=0x20, smp_affinity_list=5`
    - 然后我就发现系统还有这么个东西 —— `service irqbalance status` —— irq的balance服务默认是开启的在服务器上，我擦我擦我擦，有趣
      - **基本确定就是这个东西做的irq均衡，是系统自带的**
  - 如果指定了亲和性，**那么这个是不能被修改的**
- 此外，**关于亲和性的修改并非立即生效，而是有一个pending的过程**，直到下一次中断到来时完成切换，嘉正给出的文档中也有描述，并且挂一个我看到的调用栈
  - `irq_affinity_proc_write`
    - `write_irq_affinity`
      - `irq_set_affinity`
        - `__irq_set_affinity`
          - `irq_set_affinity_locked`
            - `if (irq_can_move_pcntxt(data) && !irqd_is_setaffinity_pending(data))`
              - `ret = irq_try_set_affinity(data, mask, force);`
            - `else`
              - `irqd_set_move_pending(data);` —— 我自己实际上是走到了这里，但是这里是直接pending了，参考嘉正的文档可知，**新的中断到来的过程中可能会完成set**
              - `irq_copy_pending(desc, mask);`
- **affinity_hint** —— https://patchwork.ozlabs.org/project/netdev/patch/20100430202343.4591.66240.stgit@ppwaskie-hc2.jf.intel.com/
  - 没看太明白，可以先摆在这里，以后再找个人慢慢消化
    - This affinity_hint handle for each interrupt can be used by underlying drivers that need a better mechanism to control interrupt affinity. The underlying driver can register a cpumask for the interrupt, which will allow the driver to provide the CPU mask for the interrupt to anything that requests it. The intent is to extend the userspace daemon, irqbalance, to help hint to it a preferred CPU mask to balance the interrupt into
- **effective_affinity/effective_affinity_list**
  - The effective IRQ affinity on SMP as some irq chips do not allow multi CPU destinations. A subset of affinity.
    - 这个解释说的很清楚

MSI-X不分配中断时，不指定亲和性，分配给哪个CPU呢

- 不会主动把所有的bitmap都置1，而是选一个当前best的CPU
- 自己后期修改的时候可以写**全1**，但是有一个pending的过程，不确定再次写到msi-x表的时候是全1还是选best，**但是结合下文这里大概率还是选择的best**

在qemu中跑nvme跑出来的结果 —— 收获颇丰

- 这个nvme太解燃眉之急了，太有趣了，其实最主要的是，我之前以为在**pci_alloc_irq_vectors_affinity**执行完毕之后，msi-x表就不会改了。但是这里确实仅仅只是确定了cpu msk，之后在申请中断处理函数的时候依然需要做进一步的修改
  - 在这个函数中 `request_threaded_irq` 会做**进一步的修改**
    - 以nvme为例，这里的irq已经带上了描述符，也就是带上了cpu msk，一个管理队列，4个io队列，申请了5个中断，在4核机器上，他们的cpu msk分别是
      - `msk=0xffffffff` —— 看起来管理队列的中断可以去任何一个地方，至少在确定分配之前是这样得 —— **这个msk在前一个阶段就已经是这样的了**
      - `msk=0x1` —— `0b1`
      - `msk=0x2` —— `0b10`
      - `msk=0x4` —— `0b100`
      - `msk=0x8` —— `0x1000`
- 至少这是一个分界线，但是 —— 这3个问题解决了，就可以去check硬件到底能不能连续发射了，还是需要软件ack
  - 如果这里的msk是全1，下面写msi-x表的时候应该如何实现？
    - `/* Try the full online mask */`
    - `return assign_vector_locked(irqd, cpu_online_mask);` —— **全0xf会有一些类似这样的调用**
      - `vector = irq_matrix_alloc(vector_matrix, dest, resvd, &cpu);` —— **Allocate a regular interrupt in a CPU map**，在**指定的CPU**上找**最好**的
        - `cpu = matrix_find_best_cpu(m, msk);` —— 说白了这里只会找**一个最好的CPU，只有一个**
  - 申请中断的时候不指定affinity，那么这里的表现如何呢？**就拿nvme的管理队列与io队列做对比呗**
    - 先从 `pci_alloc_irq_vectors_affinity(pdev, 1, irq_queues, PCI_IRQ_ALL_TYPES | PCI_IRQ_AFFINITY, &affd);` 修改成 `pci_alloc_irq_vectors_affinity(pdev, 1, irq_queues, PCI_IRQ_ALL_TYPES, NULL);`
      - 最后的cpu是 `0/1/2/3/0` —— **第一个是管理队列**，所有中断发来mask的时候都是 `0xffffffff` ，含义是你可以将其丢到**任何一个CPU上**
      - 有affinity的时候cpu是 `0/0/1/2/3` ，只有管理队列的mask是全f，其它则会有规整的平均分配，**这就是最大的差别**
    - 管理队列是如何操作的呢，这个问题还没有解决
      - 第一个管理队列的申请是这样写的， `result = pci_alloc_irq_vectors(pdev, 1, 1, PCI_IRQ_ALL_TYPES);` 确实没有care亲和性的问题
        - 那这个意思就是，管理队列，**随便选一个cpu回来就好**
    - 如果不指定亲和性，**每一个都会找当前的best cpu**
      - **所以说同一个CPU上可能有一个以上的中断**
      - 但是妈的一个中断不应该出现在不同的CPU上啊，没见有调整的啊 —— smp_affinity是会发生变化的，说明是没有绑核的，那这个就清楚了，**一切都对得上**，**我草刺激**
        - 不不不，只是这个过程中没看到，但是这种情况完全是允许的，只需要修改smp_affinity即可，直接修改的是msix表

msi-x表中的目标cpu id只有8bit，这是不是有点小的说

- 这个有点复杂，还有逻辑，物理id的说法，Cluster ID，具体参考一下内容，也需要总结出来的说
  - https://example61560.wordpress.com/2016/06/30/pcipcie-总线概述6/

博通或nvme的中断处理流程中有没有与硬件参与的部分是在悄悄告诉硬件可以继续发中断了？

- 尝试注释过mpt3sas以及nvme驱动中断处理函数中的逻辑，后续的中断不受任何影响

中断控制器收到中断后（边沿触发），如果不ack中断会发生什么？

- **无法收到任何一个中断**

继续怀疑，中断控制器的ack能作用到pcie的ep么，算是能告诉指定硬件继续发中断的信号么 —— 顿悟之后发现确实是这样的

- 这个猜测挨到边了，参考顿悟系列

中断控制器这边中断是否入队呢?

- 显然中断不入队，否则就没怎么多屁事了

脑袋灵光一闪，应该check一下qemu是如何模拟nvme的 —— 我其实就是在qemu这里找到的答案，qemu真牛皮啊

```
static void nvme_irq_assert(NvmeCtrl *n, NvmeCQueue *cq)
{
    PCIDevice *pci = PCI_DEVICE(n);

    if (cq->irq_enabled) {
        if (msix_enabled(pci)) {
            trace_pci_nvme_irq_msix(cq->vector);
            msix_notify(pci, cq->vector); /* 关键就是这句话 */
        } else {
            trace_pci_nvme_irq_pin();
            assert(cq->vector < 32);
            n->irq_status |= 1 << cq->vector;
            nvme_irq_check(n);
        }
    } else {
        trace_pci_nvme_irq_masked();
    }
}
```

- 我好想顿悟了，顿悟了，顿悟了 —— **msix_notify** —— **还得是qemu的源码牛逼**
  - `void msix_notify(PCIDevice *dev, unsigned vector)`
    - `if (msix_is_masked(dev, vector))` —— 这里会检查是否pending，并不是无脑发出的，所以联想到中断处理部分的的mask and ack，这个mask就很有意思了
      - `msix_set_pending(dev, vector);`
      - `return;`
    - `msg = msix_get_message(dev, vector);`
    - `msi_send_message(dev, msg);`
- `pci_msi_mask_irq` —— 这是linux内核中PCIe自己的mask方法
  - `__pci_msi_mask_desc(desc, BIT(data->irq - desc->irq));`
    - `if (desc->msi_attrib.is_msix)`
      - `pci_msix_mask(desc);`
        - `pci_msix_write_vector_ctrl(desc, desc->msix_ctrl);`
        - `/* Flush write to device */`
        - `readl(desc->mask_base);`
    - `else if (desc->msi_attrib.maskbit)`
      - `pci_msi_mask(desc, mask);`

msi-x是否会影响mpt3sas的性能?

- `fio -filename=/dev/sda -direct=1 -iodepth=1 -thread -rw=randwrite -ioengine=libaio -bs=4k -size=100G -numjobs=32 -runtime=30 -group_reporting -`
- 结果如下
  - 只有一个中断，且开启affinity —— 113K左右
  - 只有一个中断，关闭affinity —— 113k左右，但是本次测试中断的CPU发生了改变
  - 32个中断打开，且开启affinity —— 113k，几乎没有本质的差别
  - 32个中断打开，且关闭affinity —— 还是113k的样子

中断不入队

能走到这里，这个就算是结束了，我草，真刺激啊

还是有东西没有搞清楚，比如/proc/interrupts以及PCIe设备的IRQ到底是在描述什么呢

/proc/interrupts

```
zc@zc-System-Product-Name:/proc/irq/264$ cat smp_affinity
08000000
```

- 在目录 `/proc/irq/xx中断号` 下会有一些额外的关于中断的info，比较重要的可以关注smp_affinity，他告诉我们这个中断是被哪个CPU来处理的的，一般对于NVMe/mpt3sas等外设的reply的队列，每一个队列的中断都会给到一个CPU上
- `cat /proc/interrupts` 最左侧的起始是virq，即虚拟irq，这个是系统全局唯一的，这个值可以非常大，virq是相对于hwirq（**硬件irq，每个local apic有256个**）而言的，是软件层面，全局的irq，**分配代码如下所示**
  - `int __ref __irq_alloc_descs(int irq, unsigned int from, unsigned int cnt, int node, struct module *owner, const struct irq_affinity_desc *a`
    - `...`
    - `start = bitmap_find_next_zero_area(allocated_irqs, IRQ_BITMAP_BITS, from, cnt, 0);`
      - 所有的virq都是从全局bitmap allocated_irqs 中分配出来的，所以全局唯一
      - 该bitmap的size可以很大的说
    - `...`

下面就是驱动的初始化过程中常用的中断申请函数了

- 我先把常见的接口**摆一下**看看
  - `pci_request_irq()`
  - `pci_irq_vector()`
  - `request_threaded_irq()`
  - `disable_irq()`
  - `enable_irq()`

```
/* hisi_sas */
vectors = pci_alloc_irq_vectors_affinity(hisi_hba->pci_dev, min_msi, max_msi, PCI_IRQ_MSI | PCI_IRQ_AFFINITY, &desc); /* 这种或多或少都与MSI/MSI-X有
rc = devm_request_irq(dev, pci_irq_vector(pdev, 1), int_phy_up_down_bcast_v3_hw, 0, DRV_NAME " phy", hisi_hba);
/* nvme */
pci_alloc_irq_vectors_affinity(pdev, 1, irq_queues, PCI_IRQ_ALL_TYPES | PCI_IRQ_AFFINITY, &affd);
pci_request_irq(pdev, nvmeq->cq_vector, nvme_irq_check, nvme_irq, nvmeq, "nvme%dq%d", nr, nvmeq->qid);
/* cetc52_sas */
rc = request_threaded_irq(pdev->irq, cetc52_hard_isr, cetc52_soft_isr, IRQF_ONESHOT, CETC52_SAS_DRIVER_NAME, hpriv);
/* mpt3sas */
r = request_irq(pci_irq_vector(pdev, index), _base_interrupt, IRQF_SHARED, reply_q->name, reply_q);
```

- 然后最常见的接口其实是 `pci_request_irq()` ，上文简单描述了最近接触的各种驱动类型的中断注册流程，典型的4种基本都看到了，其实基本都大同小异，下文仔细分析一下，这里分配与申请的究竟是什么，还是研究研究每一个函数吧

int pci_request_irq(struct pci_dev *dev, unsigned int nr, irq_handler_t handler, irq_handler_t thread_fn, void *dev_id, const char *fmt, ...) —— nvme用的是这个接口，nvme驱动自己提供了nr

- This call allocates interrupt resources and enables the interrupt line and IRQ handling
  - 这个描述与函数 `request_threaded_irq` 完全一样，不过确实也是，该函数内部调用的就是 `request_threaded_irq` ，主要是把参数理解利索

- 代码分析
  - int pci_request_irq(struct pci_dev *dev, unsigned int nr, irq_handler_t handler, irq_handler_t thread_fn, void *dev_id, const char *fmt, ..
    - ret = request_threaded_irq(pci_irq_vector(dev, nr), handler, thread_fn, irqflags, devname, dev_id); —— **详见下文**

```
/**
 * pci_irq_vector - return Linux IRQ number of a device vector
 * @dev: PCI device to operate on
 * @nr: device-relative interrupt vector index (0-based).
 */
int pci_irq_vector(struct pci_dev *dev, unsigned int nr)
{
        if (dev->msix_enabled) {
                struct msi_desc *entry;
                int i = 0;

                for_each_pci_msi_entry(entry, dev) {
                        if (i == nr) return entry->irq;
                        i++;
                }
                WARN_ON_ONCE(1);
                return -EINVAL;
        }

        if (dev->msi_enabled) {
                struct msi_desc *entry = first_pci_msi_entry(dev);

                if (WARN_ON_ONCE(nr >= entry->nvec_used)) return -EINVAL;
        } else {
                if (WARN_ON_ONCE(nr > 0)) return -EINVAL;
        }

        return dev->irq + nr;
}
EXPORT_SYMBOL(pci_irq_vector);
```

- 参数解析
  - nr —— device-relative interrupt vector index (0-based)
    - 其实之前一直理解有问题的地方就是这个**nr号**，但是看了pci_irq_vector函数的实现之后，基本能get到了，虽然原理还是不是很清晰，但是心里有数了，其代码如上，分析如下
      1. 首先，不论是 `msi-x/msi/legacy` ，这个接口都是可以调用的
      2. 如果是msi-x，因为msi-x允许中断号不连续，所以它的写法是判断**i是否等于nr**，而msi直接**累加**上去就可以了（**msi的中断号必须是连续的**）
      3. 如果是mxi，排除掉错误情况之后，直接**累加即可**
      4. 如果else，即legacy，legacy的情况下，只有1个中断号，所以 `nr>0` 将直接返回报错
         - 我也清楚为啥在leapiosas只有legacy中断的情况下，我可以直接调用 `request_threaded_irq` 接口，第一个参数就是 `pdev->irq` ，这本身就是可以支持legacy的

pci_alloc_irq_vectors_affinity —— 分析一下博通mpt3sas中断向量分配的过程 —— 先申请中断号，再将中断号与中断处理函数关联在一起 —— IOMMU未开启，在开启IOMMU的情况下，事情开始变得不一样了 —— request_irq的时候也可能会修改，一次结合mpt3sas总结了

- mpt3sas_base_attach
  - mpt3sas_base_map_resources(ioc);
    - _base_enable_msix(ioc);
      - _base_alloc_irq_vectors(ioc);
        - if (ioc->smp_affinity_enable)
          - irq_flags |= PCI_IRQ_AFFINITY | PCI_IRQ_ALL_TYPES; —— **带亲和性的**
        - else
          - descp = NULL; —— **不带亲和性的**
        - pci_alloc_irq_vectors_affinity(ioc->pdev, ioc->high_iops_queues, nr_msix_vectors, irq_flags, descp); —— allocate msix vectors，其实这里要清楚，**要去写PCIe设备的MSIX表了**
          - struct irq_affinity msi_default_affd = {0};
            - if (flags & PCI_IRQ_AFFINITY) {
              - if (!affd)
                - affd = &msi_default_affd;
            - } else {

- if (WARN_ON(affd)) —— **如果没有这个PCI_IRQ_AFFINITY但是传了affd，那么就会打印一个警告的stack出来，mpt3sas确实也是遵守了这个的**
  - affd = NULL; —— 如果mpt3sas驱动没有指定PCI_IRQ_AFFINITY，那么affd就是NULL，**我就想看指定与没指定亲和性有什么区别的说**
- }
- nvecs = __pci_enable_msix_range(dev, NULL, min_vecs, max_vecs, affd, flags);
- if (affd) —— 如果存在affd，那么有下面这个**额外**的操作
  - nvec = irq_calc_affinity_vectors(minvec, nvec, affd);
- rc = __pci_enable_msix(dev, entries, nvec, affd, flags);
  - nr_entries = pci_msix_vec_count(dev); —— **这是中断向量的个数**
  - return msix_capability_init(dev, entries, nvec, affd);
    - base = msix_map_region(dev, tsize); —— 映射**MSI-X Table**到CPU的虚拟地址空间，**以便后期软件初始化该表**
      - pci_read_config_dword(dev, dev->msix_cap + PCI_MSIX_TABLE, &table_offset);
      - return ioremap(phys_addr, nr_entries * PCI_MSIX_ENTRY_SIZE);
    - msix_setup_entries(dev, base, entries, nvec, affd); —— **更新内存数据结构**
    - pci_msi_setup_msi_irqs(dev, nvec, PCI_CAP_ID_MSIX);
      - domain = dev_get_msi_domain(&dev->dev); —— **这就是熟悉的位置了vector**
      - if (domain && irq_domain_is_hierarchy(domain))
        - return msi_domain_alloc_irqs(domain, &dev->dev, nvec);
          - return ops->domain_alloc_irqs(domain, dev, nvec); —— **msi_domain_ops_default && __msi_domain_alloc_irqs**
            - irq_domain_activate_irq(irq_data, can_reserve);
              - __irq_domain_activate_irq(irq_data, reserve);
                - domain->ops->activate(domain, irqd, reserve); —— **msi_domain_ops && msi_domain_activate**
                  - irq_chip_compose_msi_msg(irq_data, msg);
                    - pos->chip->irq_compose_msi_msg(pos, msg); —— **lapic_controller && x86_vector_msi_compose_msg**
                      - __irq_msi_compose_msg(irqd_cfg(data), msg, false);
                      - msg->arch_addr_lo.base_address = X86_MSI_BASE_ADDRESS_LOW;
                      - #define X86_MSI_BASE_ADDRESS_LOW (0xfee00000 >> 20)
                      - 这个值很重要，因为FSB Interrupts存储器空间的基地址正是0xFEE
                      - 当PCIe设备对0xFEEX-XXXX这段**PCI总线域**的地址空间进行写操作时，MCH/ICH将会首先进行**PCI总线域**到**存储器域**的地址转换，之后将这个写操作翻译为FSB总线的Interrupt Message总线事务，从而向CPU内核提交中断请求
                      - msg->arch_addr_lo.dest_mode_logical = apic->dest_mode_logical;
                      - msg->arch_addr_lo.destid_0_7 = cfg->dest_apicid & 0xFF;
                      - **该irq的目标lapic的ID**，说明该irq产生的TLP写事务会直接发往irq绑定的CPU的lapic（local apic）
                      - msg->arch_data.delivery_mode = APIC_DELIVERY_MODE_FIXED;
                      - msg->arch_data.vector = cfg->vector;
                      - msg->address_hi = X86_MSI_BASE_ADDRESS_HIGH;
                  - irq_chip_write_msi_msg(irq_data, msg);
      - return arch_setup_msi_irqs(dev, nvec, type);
    - pci_intx_for_msi(dev, 0); —— **Set MSI-X enabled bits and unmask the function**

- r = _base_request_irq(ioc, i); —— 上一步的作用是分配中断向量，现在将中断向量与对应的中断处理函数绑定在一起，在该过程中的某些条件下也会有msi-x表的修改
  - r = request_irq(pci_irq_vector(pdev, index), _base_interrupt, IRQF_SHARED, reply_q->name, reply_q);
    - return request_threaded_irq(irq, handler, NULL, flags, name, dev);
      - retval = __setup_irq(irq, desc, action);
        - ret = irq_activate(desc);
          - if (!irqd_affinity_is_managed(d))
            - return irq_domain_activate_irq(d, false);
        - irq_startup(desc, IRQ_RESEND, IRQ_START_COND); —— 在某些条件下会执行

自己是如何利用mpt3sas为plda书写msi-x中断代码的

```
diff --git a/leapio_mod.c b/leapio_mod.c
index 38b4603..4b9b597 100644
--- a/leapio_mod.c
+++ b/leapio_mod.c
@@ -664,6 +664,19 @@ static void leapio_channnels_destroy(plda_board_t * pBoard)
        }
 }

+struct msixmust {
+       u8                              msix_index;
+       char                    name[32];
+};
+
+static struct msixmust msixmusts[32];
+
+static irqreturn_t
+_base_interrupt(int irq, void *bus_id)
+{
+       pr_err("%s 中断is comming!\n", __func__);
+       return IRQ_HANDLED;
+}

 static int leapio_plda_probe(struct pci_dev *pdev,
                             const struct pci_device_id *id)
@@ -766,8 +779,35 @@ static int leapio_plda_probe(struct pci_dev *pdev,
                }
        }

-       return 0;
+       /* 检查一下这个设备的msi-x中断 */
+       int base;
+       u16 message_control;
+       int irq_flags = PCI_IRQ_MSIX;
+       int ret;
+       int i;

+       base = pci_find_capability(pdev, PCI_CAP_ID_MSIX);
+       if (!base)
+               pr_err("%s msix not supported!\n", __func__);
+       else
+               pr_err("%s msix supported!\n", __func__);
+
+       pci_read_config_word(pdev, base + 2, &message_control);
+       pr_err("%s msix count=%d!\n", __func__, ((message_control & 0x3FF) + 1));
+
+       irq_flags |= PCI_IRQ_AFFINITY | PCI_IRQ_ALL_TYPES;
+       ret = pci_alloc_irq_vectors_affinity(pdev, 0, 32, irq_flags, NULL);
+       pr_err("%s 返回的数量=%d\n", __func__, ret);
+
+       for(i=0; i<32; i++) {
+               msixmusts[i].msix_index = i;
+               snprintf(msixmusts[i].name, 32, "plda-msix%d", i);
+               ret = request_irq(pci_irq_vector(pdev, i), _base_interrupt, IRQF_SHARED, msixmusts[i].name, (void*)(&(msixmusts[i])));
+               if (ret)
+                       pr_alert("%s request_irq gg simida i=%d\n", __func__, i);
+       }
+
+       return 0;

 disable_msi:
        pci_disable_msi(pBoard->pdev);
@@ -813,6 +853,7 @@ static void leapio_unmap_bars(plda_board_t * pBoard, struct pci_dev *pdev)
 void leapio_plda_remove(struct pci_dev *pdev)
 {
        int ch_idx;
+       int i;
        plda_board_t *pBoard = (plda_board_t *) pci_get_drvdata(pdev);

        dev_info(&pdev->dev, "Remove: Releasing board %04x:%04x.\n",
@@ -845,6 +886,13 @@ void leapio_plda_remove(struct pci_dev *pdev)
                // deassociate device from board to be picked up by char device
                pBoard->pdev = NULL;

+               // 中断需在在这个地方被释放
```

```
+               for(i=0; i<32; i++)
+                       free_irq(pci_irq_vector(pdev, msixmusts[i].msix_index), (void*)(&(msixmusts[i])));
+
+               pci_free_irq_vectors(pdev);
+
+

                leapio_unmap_bars(pBoard, pdev);

                pci_release_regions(pdev);
```

static inline int __must_check request_irq(unsigned int irq, irq_handler_t handler, unsigned long flags, const char *name, void *dev)

```
static inline int __must_check
request_irq(unsigned int irq, irq_handler_t handler, unsigned long flags,
            const char *name, void *dev)
{
        return request_threaded_irq(irq, handler, NULL, flags, name, dev);
}
```

- 分析详见下文

int request_threaded_irq(unsigned int irq, irq_handler_t handler, irq_handler_t thread_fn, unsigned long irqflags, const char *devname, void *dev_id)

- ref
  - https://lwn.net/Articles/302043/
    - Processing interrupts from the hardware is a **major source of latency** in the kernel, because other interrupts **are blocked** while doing that processing
      - 即**中断线程化**
    - `request_threaded_irq()` will create a thread for the interrupt and put a pointer to it in the `struct irqaction`
- This call **allocates interrupt resources** and **enables the interrupt line and IRQ handling**
- Since your handler function must clear any interrupt the board raises, you must take care both to initialise your hardware and to set up the interrupt handler in the right order
  - 必须unmask掉硬件的寄存器，否则硬件是发不出来中断的
  - 与之对应是更靠前的一个步骤，**ack操作**
    - 必须ack，否则谁的中断都进不来
- **handler位于中断上下文**
- **thread_fn位于进程上下文，这里是完全创建了一个新的线程**
- 代码分析
  - `int request_threaded_irq(unsigned int irq, irq_handler_t handler, irq_handler_t thread_fn, unsigned long irqflags, const char *devname, voi` —— 其实关键就是irq的获取方式， `pdev->irq` /大部分是 `pci_irq_vector(dev, nr)` —— **nr是基于0的index，即0对应pdev->irq，如果涉及到超过一个的中断向量，那么是MSI/MSI-X**
    - `struct irqaction *action;`
    - `struct irq_desc *desc;`
    - `if (((irqflags & IRQF_SHARED) && !dev_id) || ((irqflags & IRQF_SHARED) && (irqflags & IRQF_NO_AUTOEN)) || (!(irqflags & IRQF_SHARED) &&`
      - `return -EINVAL;` —— 有一些flag是相互冲突的，如果是shared，则必须传入一个**真实的dev-ID**
    - `desc = irq_to_desc(irq);` —— 对于PCIe设备而言， `irq` 号**lspci**的时候就看得到
      - `return radix_tree_lookup(&irq_desc_tree, irq);`
    - `if (!handler)`
      - `if (!thread_fn) return -EINVAL;` —— 两个handler都没有给到那就得**报错**
      - `handler = irq_default_primary_handler;`
        - `static irqreturn_t irq_default_primary_handler(int irq, void *dev_id)`
          - `return IRQ_WAKE_THREAD;` —— 这其实就是**唤醒**，自己写的也是这个
    - `action = kzalloc(sizeof(struct irqaction), GFP_KERNEL);` —— 这句话本身可能**睡眠**
    - `action->handler = handler;` —— **中断上下文**
    - `action->thread_fn = thread_fn;` —— **进程上下文**
    - `action->flags = irqflags;`
    - `action->name = devname;`

        - `action->dev_id = dev_id;`
        - `irq_chip_pm_get(&desc->irq_data);` —— **Enable power for an IRQ chip**
        - `retval = __setup_irq(irq, desc, action);` —— **Internal function to register an irqaction**
            - `new->irq = irq;`
            - `nested = irq_settings_is_nested_thread(desc);` —— **Check whether the interrupt nests into another interrupt thread**
            - `if (new->thread_fn && !nested)`
                - `ret = setup_irq_thread(new, irq, false);` —— **Create a handler thread when a thread function is supplied and the interrupt does not nest into another interrupt thread**
                    - `t = kthread_create(irq_thread, new, "irq/%d-%s", irq, new->name);` —— 这里是真的有创建一个**新线程**的
                    - `sched_set_fifo(t);`
                    - `new->thread = get_task_struct(t);`
                    - `set_bit(IRQTF_AFFINITY, &new->thread_flags);`
            - 后续的代码量其实还蛮大的，暂时先到这个位置
        - `return retval;`
- tips
    - 使用 `request_thread_irq` 来申请中断时，通常的处理是使用NULL来利用系统默认的**primary_handler（就是唤醒thread_fn）**，这个时候**IRQF_ONESHOT**标识必不可少，否则中断会申请失败。**IRQF_ONESHOT**的作用是保证thread_handler函数**执行完整**，才会接受下一个中断信号
    - 驱动里有时候会增加延迟，这个的主要目的是为了**消抖**，例如耳机插入手机的一个过程，因为有信号的干扰，并不是一有信号就意味着耳机真的插入了，所以耳机驱动一般在第一次感受到信号的中断后，会一定时间后去检查信号是否稳定，确定稳定后才断定有耳机插入
        - **所以延迟主要是为了消除抖动的问题**
    - irqchip
        - 随着linux kernel的发展，将interrupt controller抽象成irqchip这个概念越来越流行
    - irq domain —— `linux/Documentation/translations/zh_CN/core-api/irq/irq-domain.rst`
        - 随着系统复杂度加大，外设中断数据增加，实际上系统可以需要多个中断控制器进行级联，面对这样的趋势，linux kernel工程师如何应对？答案就是irq domain这个概念
        - 系统中所有的interrupt controller会形成树状结构，对于每个interrupt controller都可以连接若干个外设的中断请求（我们称之interrupt source）
        - interrupt controller会对连接其上的interrupt source（根据其在Interrupt controller中物理特性）进行编号（也就是HW interrupt ID了）
        - 但这个编号仅仅限制在本interrupt controller范围内（也就是所谓的irq_domain内）

重点研究一下request_threaded_irq的flags参数

```
// linux/include/linux/interrupt.h
/*
 * These flags used only by the kernel as part of the
 * irq handling routines.
 *
 * IRQF_SHARED - allow sharing the irq among several devices
 * IRQF_PROBE_SHARED - set by callers when they expect sharing mismatches to occur
 * IRQF_TIMER - Flag to mark this interrupt as timer interrupt
 * IRQF_PERCPU - Interrupt is per cpu
 * IRQF_NOBALANCING - Flag to exclude this interrupt from irq balancing
 * IRQF_IRQPOLL - Interrupt is used for polling (only the interrupt that is
 *                registered first in a shared interrupt is considered for
 *                performance reasons)
 * IRQF_ONESHOT - Interrupt is not reenabled after the hardirq handler finished.
 *                Used by threaded interrupts which need to keep the
 *                irq line disabled until the threaded handler has been run.
 * IRQF_NO_SUSPEND - Do not disable this IRQ during suspend.  Does not guarantee
 *                   that this interrupt will wake the system from a suspended
 *                   state.  See Documentation/power/suspend-and-interrupts.rst
 * IRQF_FORCE_RESUME - Force enable it on resume even if IRQF_NO_SUSPEND is set
 * IRQF_NO_THREAD - Interrupt cannot be threaded
 * IRQF_EARLY_RESUME - Resume IRQ early during syscore instead of at device
 *                resume time.
 * IRQF_COND_SUSPEND - If the IRQ is shared with a NO_SUSPEND user, execute this
 *                interrupt handler after suspending interrupts. For system
 *                wakeup devices users need to implement wakeup detection in
 *                their interrupt handlers.
 * IRQF_NO_AUTOEN - Don't enable IRQ or NMI automatically when users request it.
 *                Users will enable it explicitly by enable_irq() or enable_nmi()
 *                later.
 * IRQF_NO_DEBUG - Exclude from runnaway detection for IPI and similar handlers,
 *                 depends on IRQF_PERCPU.
 */
/*
 * 在中断到来时，会遍历执行共享此中断的所有中断处理程序，直到某一个程序返回 IRQ_HANDLED ；故在中断处理函数中，应根据硬件寄存器的信息比对dev_id参数，迅速判断是
 * mpt3sas的驱动在申请中断时就有使用该flag
 * 但是实际上，如果中断控制器能够支持足够多的中断源，那么share是不推荐的，毕竟还是有额外的开销
 */
#define IRQF_SHARED                0x00000080  /* 多设备间可以共享中断 */
#define IRQF_PROBE_SHARED       0x00000100  /* caller其实可以预见sharing mismatche的发生 */
#define __IRQF_TIMER            0x00000200  /* 没见过 */
/*
 * 在smp架构下，中断有两种mode，一种是共享的，一种是per cpu的
 *  对于共享的来说，当一个CPU ack了一个中断之后，所有的CPU都会看到，其它CPU就不用处理了
 *  与之相对的典型例子就是本地CPU的时钟中断了，每一个CPU都有各自的一套寄存器
 */
#define IRQF_PERCPU                0x00000400  /* 字面意思 */
/*
 * 这也是和multi-processor相关的一个flag。对于那些可以在多个CPU之间共享的中断，具体送达哪一个processor是有策略的，我们可以在多个CPU之间进行平衡
 * 如果你不想让你的中断参与到irq balancing的过程中那么就设定这个flag
 */
#define IRQF_NOBALANCING        0x00000800  /* 如上 */
#define IRQF_IRQPOLL            0x00001000  /* 这个中断是用于polling的 */
/*
 * 这个内容有点意思哦，关于中断嵌套的问题
 *  首先，中断的primary handler显然是不会嵌套的，众多机制有所保证，但是对于threaded interrupt handler，情况可能是不太一样的，毕竟位于进程上下文，什*  么都不
 *
 *  如果在中断的底半部，中断寄存器如果我mask掉，那么这个时候是不会有相同的中断被触发的，所以中断一定不会嵌套执行，但是假如我在下半部的thread中unmask了中*  断，
 *
 *  那确实oneshot彻底的保证了中断不会重入，完全是串行化的
 */
#define IRQF_ONESHOT            0x00002000  /* 一次性触发，不会嵌套 */
#define IRQF_NO_SUSPEND         0x00004000  /**/
#define IRQF_FORCE_RESUME       0x00008000  /**/
#define IRQF_NO_THREAD          0x00010000  /**/
#define IRQF_EARLY_RESUME       0x00020000  /**/
#define IRQF_COND_SUSPEND       0x00040000  /**/
#define IRQF_NO_AUTOEN          0x00080000  /**/
#define IRQF_NO_DEBUG           0x00100000  /**/
```

有一个文档说的不错，专门来记录一下它关于中断的描述

- ref
  - https://blog.csdn.net/kris_fei/article/details/77435718
    - 笔记分享[中断]中断申请释放以及上下半部
- 参数IRQF_ONESHOT表示中断线程处理函数执行完毕之后才可以开启中断，oneshot与shared不能共用，因为shared不能关中断
- **当中断一直被触发时，中断线程处理函数会得不到运行，此时flag可以加上irqf_oneshot**
- 中断上下文
  - 在中断上下文，无法与current建立联系，不能sleep(何时唤醒呢，如何唤醒呢)，可能引起休眠的函数printk不能用在中断上下文使用

底半部应该如何处理呢3类中断呢

- 首先在顶半部，**CPU中断全关**，这个是**CPU+现代linux内核**决定的
  - 就是什么类型的其它中断都进不来，**所有的中断都进不来**
- 但是在所有的底半部，中断都是开启的，其它类型中断随便进，主要是这个时期，CPU的EFLAGS寄存器已经把对应的bit消除了
- 顶半部**大大减少了**系统关中断的时间
  - 其实顶半部，底半部的处理简单来说就是**顶半部记录，底半部处理**
- ref
  - https://www.cnblogs.com/SsoZhNO-1/p/12559506.html
    - 中断上下文不是线程，是一个更**轻**的**内核控制路径**
    - 中断上下文 —— 对于中断而言，是硬件通过触发信号，导致内核调用中断处理程序，进入内核空间，这个过程中硬件的一些变量和参数也要传递给内核，这些**参数和被中断的进程的上下文**就是**中断上下文**
  - **其它部分打散糅下去了**

softirq —— 中断上下文

- 处理软中断的过程中，中断本身是开启的。本地CPU的软中断被禁止了，但是其他CPU的软中断并未被禁止，所以程序员需要小心的处理多核上中断处理函数的**并行**执行。为了极致的性能，必须如此，普通驱动程序完全没有必要使用softirq，性能最好用到tasklet即可
- softirq过程中是允许中断的，但是抢占应该关闭
- 在softirq中，只是本地软中断被禁止，所以在单个core上无需考虑重入的问题，但是tasklet是串行的，程序员可以放心的使用他们

tasklet —— 中断上下文

- tasklet的同一个中断的中断处理函数不会在两个CPU上同时执行。中断处理函数如果需要休眠就选workqueue否则tasklet即可
- tasklet是基于softirq的
  - **softirq倾向于性能，而tasklet倾向于易用性**
- 每一个CPU都维护了一个list来记录自己处理的tasklet，总归是串行的

workqueue —— 进程上下文

with BKF 由于中断所延伸出来的一些杂事

SMP系统引导

- ref
  - https://frankjkl.github.io/2019/03/10/Linux内核-SMP系统的引导/

- 系统的引导和初始化阶段是个特例，因为在这个阶段里系统中只有**一个上下文**，只能由一个处理器来处理。在这个阶段里，也就是在系统刚加电或reset之后，系统中**暂时只有一个处理器运行**
  - 这个处理器称之为**引导处理器BP**
  - 其余的处理器则处于暂停状态，称为**应用处理器AP**
- **引导处理器**完成整个系统的引导和初始化，并创建起多个进程，从而可以由多个处理器同时参与处理时，才启动所有的**应用处理器**，让他们完成自身的初始化以后，投入运行，我们在这里关心的是**引导处理器**怎样为各个**应用处理器**做好准备，然后启动其运行的过程
- 在初始化阶段，引导处理器先完成自身的初始化，进入保护模式并开启页式存储管理机制，再完成系统特别是内存的初始化，然后从 `start_kernel()->rest_init()->kernel_init()->smp_init()` 进行**SMP系统**的初始化
  - 由于此时APs处于暂停状态，所以BP需要通过 `smp_init()->cpu_up()->native_cpu_up()->do_boot_cpu()->wakeup_secondary_cpu_via_init()` 发送IPI中断唤醒APs，这样APs就开始了正常的运行过程，拥有和BP一样的地位

IDT的进一步研究

- 不管多少cpu，**linux内核都维护一个IDT表**，不同CPU的IDTR寄存器都保存**同样的IDT表地址**

linux下的一个宏 ASSEMBLY

- 某些常量宏会同时被C和asm引用，而C与asm在对立即数符号的处理上是不同的。asm中通过指令来区分其操作数是有符号还是无符号的，而不是通过操作数。而C中是通过变量的属性，而不是通过操作符。C中如果要指明常量有无符号，必须为常量添加后缀，而asm则通过使用不同的指令来指明。如此，当一个常量被C和asm同时包含时，必须做不同的处理。故KBUILD_AFLAGS中将添加一项D__ASSEMBLY__，来告知预处理器此时是asm

汇编的一些语法

- `.byte expressions`
  - 将表达式汇编成一个字节存入下一个地址，可以有多个参数
- `.rept count`
  - `.rept` 和 `.endr` 之间的语句count次

tasklet可以指定cpu加载模块，然后再module_init中能够获取当前cpu现场

关于中断的一些tips

- 由于**不同CPU中相同的中断向量对应的虚拟中断号irq需要不同**（这是irq这一虚拟中断层出现的原因，也可以认为这是一个**实锤**的地方），所以，需要每个CPU在系统初始化的时候，将虚拟中断号irq与其本地的中断向量进行一个映射
- 然后，cpu会根据中断向量携带的相关信息，中断号，设备号等，先找到对应的整个内核都可以看得到的irq号（这部分映射主要是通过硬件传过来的硬件中断以及事先建立好的中断向量与irq号映射关系获得）
  - 获得irq号之后，就会有一个与之对应的irq_desc结构
- 这里的action是一串链表，表示当前irq号对应的一些操作，这时候就会遍历这个链表找到对应的硬件设备信息来执行相关的函数了。这些相关的函数，**其实就是在驱动程序向系统注册相关irq的时候挂在irqaction链表后面的**
- ack表示我收到了中断，EOI表示中断处理结束