Linux中断处理

一、中断流程概述

1. 当设备发出一个中断信号后，由CPU中的中断控制器接收，产生一个 中断；
2. 中断控制器通知CPU此次中断的中断向量号；
3. CPU获得中断后，从IDTR寄存器中读取IDT的基地址；
4. 从中断描述符表（IDT）中取得段选择子与偏移；
5. 用段选择子从GDT中找到相应段的起始地址
6. 根据起始地址+偏移取得中断处理函数
7. 执行中断处理函数。

二、中断初始化

2.1 进入保护模式

统刚启动时，CPU 运行在实模式下，由 BIOS 提供中断服务。 我们进入内核后首先要将CPU从实模式设置为保护模式。一旦进入保护模式，BIOS的中断服务就不再可用，我们需要加载Linux的中断服务。

X86 CPU中，用一个48位的IDTR寄存器存储IDT的基地址（32位）和 IDT的表长度（16位）。因此在进入保护模式之前，需要将IDTR寄存器初始化为0。

```
///arch/x86/boot/main.c 硬件初始化，将 CPU 设置为保护模式
void main(void) {
    ...
    go_to_protected_mode();
}
///arch/x86/boot/pm.c 调用函数初始化 IDT 和 GDT
void go_to_protected_mode(void) {
    ...
    /* Actual transition to protected mode... */
    setup_idt();
    setup_gdt();
    protected_mode_jump(boot_params.hdr.code32_start,
    (u32)&boot_params + (ds() << 4));
}
```

```
struct gdt_ptr {
```

```
    u16 len;
    u32 ptr;
} __attribute__((packed));

//IDTR 寄存器初始化为 0
static void setup_idt(void) {
    static const struct gdt_ptr null_idt = {0, 0};
     asm volatile("lidtl %0" : : "m" (null_idt));
}
```

2.2 第一轮初始化

CPU进入到保护模式之后，先进行内核解压缩，再进入 kernel/head_64.S，初始化IDT、GDT，创建内核页表。

初始化完成后，调用kernel/head64.c中的 x86_64_start_kernel()，该函数中进行一些设定后调用init/main.c中的start_kernel()，正式启动内核。

```
//arch/x86/kernel/head_64.S 初始化进程运行环境
/* CPU 在使用虚拟地址之前的设定 */
    call    startup_64_setup_env
/* Setup and Load IDT （在使用虚拟内存后） */
    pushq %rsi call
    early_setup_idt
    popq %rsi
.Ljump_to_C_code:
// Line 252
    pushq $.Lafter_lret # put return address on stack for unwinder
    xorl    %ebp, %ebp # clear frame pointer
    movq initial_code(%rip), %rax
    pushq $__KERNEL_CS # set correct cs
    pushq %rax    # target address in negative space
    lretq
SYM_DATA(initial_code,.quad x86_64_start_kernel) // Line 339
```

```
//arch/x86/kernel/head64.c 初始化进程运行环境
asmlinkage __visible void __init x86_64_start_kernel(char *
real_mode_data) {
    idt_setup_early_handler();
    x86_64_start_reservations(real_mode_data);
}
void __init x86_64_start_reservations(char *real_mode_data) {
    start_kernel();
```

```
9 }
```

这里有三个与 IDT 的初始化相关的函数：

- startup_64_setup_env
- early_setup_idt
- idt_setup_early_handler

startup_64_setup_env() 和early_setup_idt() 将 bringup_idt_table数组加载到 IDTR。这个数组定义在head64.c中，是最初的IDT。

这个IDT一直要用到x86_64_start_kernel() 中调用 idt_setup_early_handler()，才被 idt_table替换掉。

由于修改 idt_table的代码都在 idt.c中，在CPU 早期启动时不可用， 且此时许多CPU状态都没有设置，所以要用 bringup_idt_table 作为过渡。

中断描述符由gate_desc结构体表示，有GATE_INTERRUPT、GATE_TRAP、GATE_CALL和GATE_TASK几种类型，idt_table就是gate_desc类型的数组。

内核在中断初始化的过程中可以为0x20-0xeb号中断指定具体的中断处理程序，此时由system_vectors位图表示一个中断是否已指定，没有指定的会被自动赋值为默认值，irq_entries_start[i]（见4.2.3小节）。

startup_64_setup_env()：

在/arch/x86/kernel/head64.c中 startup_64_setup_env() 初始化 GDT 和 IDT。 它调用 startup_64_load_idt()，此时内核页表还没初始化完成，仍然是直接寻址模式。 startup_64_load_idt() 将 bringup_idt_table 的物理地址存入IDTR寄存器。

early_setup_idt():

内核切换到虚拟地址模式之后，再调用 early_setup_idt() ，重新将 bringup_idt_table 的虚拟地址存入 IDTR 寄存器。

idt_setup_early_handler():

最后在x86_64_start_kernel() 里，调用 idt.c 中的 idt_setup_early_handler()，我们终于见到了真正的 idt_table。 在这个函数中，为idt_table中的表项赋初始值，以使 idt_table对应初始的中断处理程序。最后load_idt(&idt_descr)，用idt_table替换了bringup_idt_table。

2.3 第二轮初始化

以上是系统早期启动时，在 start_kernel() 之前做的事情，此时 idt_table 中的表项均指向默认的中断处理程序 early_idt_handler_common。在第二遍初始化中，我们要使不同的中断指向不同的处理程

序。

这里与IDT初始化相关的有五个函数：

· idt_setup_early_traps()

· idt_setup_early_pf()

· trap_init()//以上三个函数初始化系统保留的中断

· early_irq_init()

· init_IRQ()//这两个函数初始化外部中断

· idt_setup_early_traps() 初始化IDT中的X86_TRAP_DB（调试）和 X86_TRAP_BP（断点）两项；

· X86-32 下，X86_TRAP_PF（页错误）也会在这里一起初始化，但在 X86-64 下 Linux 内核还需要原有的early_idt_handler_array 中的初 始中断处理函数去初始化早期页表，所以页错误初始化只能单独写成函数 idt_setup_early_pf()；

· 再之后，经过一些必要的系统设置，调用 trap_init() 将其他的系统保留中断对应到入口函数。

2.4 中断请求队列的初始化

专用中断门只用于一种特定的中断源，但通用中断门可以为多个中断源所 共用，不同中断源要执行的中断处理程序也各不相同，在系统运行过程中这个共用的结构也会动态地变化。

因此，在IDT 中，为每个外部中断表项准备一个中断请求队列，让不同中断源的中断服务程序都挂在对应的队列里，这样就形成了一个中断请求队列的数组，这就是 irq_desc 数组，其中 struct irqaction *action 项就对应具体的中断请求队列。

kernel/irq/irqdesc.c

· early_irq_init() 初始化 irq_desc 结构体。

kernel/irqinit.c

· init_IRQ() 调用x86_init.irqs.intr_init() 该函数即native_init_IRQ()

· 其调用 idt.c 中的 idt_setup_apic_and_irq_gates() 将外部中断设置到 IDT中。

注意：kernel/irq/irqdesc.c 中有两个early_irq_init()，分别对应IRQ是否稀疏化。可以用宏 CONFIG_SPARSE_IRQ 切换这两种初始化方式。

中断设备注册中断：

使用中断模式的设备，在使能中断之前必须设置触发方式（电平/边沿触发等）、irq号、处理函数等信息。内核提供了request_irq和request_threaded_irq两个函数可以方便地配置这些信息，前者用于调用，而后者用于实现，具体如下。

· 通过调用 include/linux/interrupt.h中的 request_irq()，可以注册一个外部中断，将其挂到对应的中断请求队列上。
· request_irq() 其实是调用了kernel/irq/manage.c 中的 request_threaded_irq()，这里会设置一个新的action，并调用__setup_irq() 将其链接到队列中。

```
int request_threaded_irq(unsigned int irq, irq_handler_t handler,
            irq_handler_t thread_fn,

            unsigned long flags, const char *name, void *dev);
int request_irq(unsigned int irq, irq_handler_t handler, unsigned long
flags,
            const char *name, void *dev)
```

第二个参数handler表示对中断的第一步处理，thread_fn表示在独立线程中执行的处理函数，request_irq将thread_fn置位NULL，调用request_threaded_irq表示不在独立线程中进行中断处理，flags表示中断触发方式、中断共享等的标志，dev是设备绑定的数据，用作调用handler和thread_fn时传递的参数。

request_threaded_irq函数主要完成以下工作。

（1）首先，根据传递的参数对irqaction对象的字段赋值。

（2）其次，将新的irqaction链接至irq对应的irq_desc。这里涉及中断共享，当多个设备共享同一个irq时，要求每个设备都在flags中设置IRQF_SHARED标志、设置的触发方式一致、IRQF_ONESHOT等的设置也要相同。满足以上条件，新的irqaction会被插入到irq_desc的action指向的链表尾部。

（3）最后，如果thread_fn不等于NULL，则调用setup_irq_thread新建一个线程来处理中断。

中断处理函数handler和thread_fn的编写要遵守以下几个重要原则。

首先，中断处理打断了当前进程的执行，同时需要进行一系列复杂的处理，所以要快速返回，不能在handler中做复杂的操作，如I/O操作等，这就是所谓的中断处理的上半段（Top Half）。如果需要复杂操作，一般有两种常见做法，一种是在函数中启动工作队列或者软中断（如tasklet）等，由工作队列等来完成时间工作；第二种做法是在thread_fn中执行，这就是所谓的中断处理的下半段（Bottom Half）。

其次，handler中不能进行任何sleep的动作，调用sleep，使用信号量、互斥锁等可能导致sleep的机制都不可行。

最后，不要在handler中调用disable_irq这类需要等待当前中断执行完毕的函数，中断处理中调用一个需要等待当前中断结束的函数，会发生“死锁”现象[即两个或两个以上进程（线程）在执行过程中，

因争夺资源而造成的一种互相等待的现象]。实际上，handler执行的时候，一般外部中断依然是在禁止的状态，不需要disable_irq。

request_irq通过将request_threaded_irq的thread_fn参数置为NULL来实现，那么它们究竟有什么区别呢？最直观的，request_irq的handler直接在当前中断上下文中执行，request_threaded_irq的thread_fn在独立的线程中执行。根据前面的第一条原则，handler中不能进行复杂操作，操作由工作队列等进行，工作队列实际上也是进程上下文。二者看似趋于一致，但实际上还是有重要差别的：执行thread_fn进程的优先级比工作队列进程优先级要高。

request_threaded_irq创建新线程时，会调用sched_setscheduler_nocheck(t, SCHED_FIFO,...)将线程设置为实时的。所以，对用户体验影响比较大、要求快速响应的设备的驱动中，采用中断模式的情况下，使用request_threaded_irq有利于提高用户体验；相反，要求不高的设备的驱动中，使用request_irq更合适。

三、中断处理的详细流程

3.1 系统调用

可以使用syscall机器指令来调用系统调用

比如x86_64平台，在执行syscall机器码之前，系统调用的编号要先放到rax寄存器，参数要分别放到rdi、rsi、rdx、r10、r8、r9寄存器中，这样kernel中的代码就会从这些地方取值，然后继续执行逻辑，当kernel部分的逻辑完成之后，结果会再放到rax寄存器中，这样user space的部分就可以从rax寄存器中拿到返回值。

当我们执行syscall机器指令时，MSR_LSTAR寄存器中存放的对应方法就会被执行

```
// arch/x86/kernel/cpu/common.c
void syscall_init(void)
{
        ...
        wrmsrl(MSR_LSTAR, (unsigned long)entry_SYSCALL_64);
        ...
}
```

在上面的方法中，我们可以看到，汇编代码entry_SYSCALL_64被写到了MSR_LSTAR表示的寄存器中。

```
// arch/x86/entry/entry_64.S
ENTRY(entry_SYSCALL_64)
        ...
        call    do_syscall_64           /* returns with IRQs disabled */
```

```
5         ...
```

在entry_SYSCALL_64中调用了do_syscall_64

```
1  // arch/x86/entry/common.c
2  __visible void do_syscall_64(unsigned long nr, struct pt_regs *regs)
3  {
4      ...
5      if (likely(nr < NR_syscalls)) {
6            nr = array_index_nospec(nr, NR_syscalls);
7            regs->ax = sys_call_table[nr](regs);
8      }
9      ...
10 }
```

do_syscall_64根据系统调用的编号，到数组sys_call_table中找到对应方法，然后调用。

那么系统调用函数是如何被填入sys_call_table中的呢?

假设目标系统调用是write，其对应的内核源码为：

```
1  // fs/read_write.c
2  SYSCALL_DEFINE3(write, unsigned int, fd, const char __user *, buf,
3                  size_t, count)
4  {
5          return ksys_write(fd, buf, count);
6  }
```

这里主要看下SYSCALL_DEFINE3这个宏定义：

```
1  // include/linux/syscalls.h
2  #define SYSCALL_DEFINE1(name, ...) SYSCALL_DEFINEx(1, _##name, __VA_ARGS__)
3  #define SYSCALL_DEFINE2(name, ...) SYSCALL_DEFINEx(2, _##name, __VA_ARGS__)
4  #define SYSCALL_DEFINE3(name, ...) SYSCALL_DEFINEx(3, _##name, __VA_ARGS__)
5  #define SYSCALL_DEFINE4(name, ...) SYSCALL_DEFINEx(4, _##name, __VA_ARGS__)
6  #define SYSCALL_DEFINE5(name, ...) SYSCALL_DEFINEx(5, _##name, __VA_ARGS__)
7  #define SYSCALL_DEFINE6(name, ...) SYSCALL_DEFINEx(6, _##name, __VA_ARGS__)
8  ...
9  #define SYSCALL_DEFINEx(x, sname, ...)                          \
10         ...
11         __SYSCALL_DEFINEx(x, sname, __VA_ARGS__)
```

该宏又引用了__SYSCALL_DEFINEx，继续看下：

```
// arch/x86/include/asm/syscall_wrapper.h
#define __SYSCALL_DEFINEx(x, name, ...)                                         \
        asmlinkage long __x64_sys##name(const struct pt_regs *regs);    \
        ...                                                                     \
        static long __se_sys##name(__MAP(x,__SC_LONG,__VA_ARGS__));     \
        static inline long __do_sys##name(__MAP(x,__SC_DECL,__VA_ARGS__));\
        asmlinkage long __x64_sys##name(const struct pt_regs *regs)     \
        {                                                                       \
                return __se_sys##name(SC_X86_64_REGS_TO_ARGS(x,__VA_ARGS__));\
        }                                                                       \
        ...                                                                     \
        static long __se_sys##name(__MAP(x,__SC_LONG,__VA_ARGS__))      \
        {                                                                       \
                long ret = __do_sys##name(__MAP(x,__SC_CAST,__VA_ARGS__));\
                ...                                                             \
                return ret;                                                     \
        }                                                                       \
        static inline long __do_sys##name(__MAP(x,__SC_DECL,__VA_ARGS__))
```

该宏的参数中，x为3，name为_write，...代表的__VA_ARGS__为unsigned int, fd, const char __user *, buf, size_t, count。

接着，在宏的定义中，先声明了三个函数，分别为__x64_sys_write、_se_sys_write、__do_sys_write，紧接着，定义了__x64_sys_write和_se_sys_write的实现，__x64_sys_write内调用_se_sys_write，_se_sys_write内调用__do_sys_write。

__do_sys_write只是一个方法头，它和最开始的write系统调用的方法体构成完整的方法。

由上可以看到，三个方法中，只有__x64_sys_write方法没有static，即只有它是外部可调用的，所以我们看下哪里引用了__x64_sys_write。

```
// arch/x86/entry/syscalls/syscall_64.tbl
#
# 64-bit system call numbers and entry vectors
#
# The format is:
# <number> <abi> <name> <entry point>
#
# The __x64_sys_*() stubs are created on-the-fly for sys_*() system calls
#
# The abi is "common", "64" or "x32" for this file.
#
0       common  read                    __x64_sys_read
```

```
1       common  write                   __x64_sys_write
...
```

我们会在一个非c文件中，找到了对__x64_sys_write方法的引用，但这个文件又是怎么被使用的呢？

根据arch/x86/entry/syscalls/Makefile我们可以知道，是有对应的shell脚本，根据上面的文件来生成c版的头文件，比如下面两个。

kernel内部使用的：

```
// arch/x86/include/generated/asm/syscalls_64.h
#ifdef CONFIG_X86
__SYSCALL_64(0, __x64_sys_read, )
#else /* CONFIG_UML */
__SYSCALL_64(0, sys_read, )
#endif
#ifdef CONFIG_X86
__SYSCALL_64(1, __x64_sys_write, )
#else /* CONFIG_UML */
__SYSCALL_64(1, sys_write, )
#endif
...
```

给用户使用的：

```
// arch/x86/include/generated/uapi/asm/unistd_64.h
#define __NR_read 0
#define __NR_write 1
...
```

那生成的这两个头文件又是给谁使用的呢？看下下面这个文件：

```
// arch/x86/entry/syscall_64.c
#define __SYSCALL_64(nr, sym, qual) [nr] = sym,

asmlinkage const sys_call_ptr_t sys_call_table[__NR_syscall_max+1] = {
        /*
         * Smells like a compiler bug -- it doesn't work
         * when the & below is removed.
         */
        [0 ... __NR_syscall_max] = &sys_ni_syscall,
#include <asm/syscalls_64.h>
```

```
11 };
```

该文件中定义了一个const的数组变量sys_call_table，数组下标为系统调用的编号，值为该编号对应的系统调用方法。

最开始整个数组都初始化为sys_ni_syscall，该方法内会返回错误码ENOSYS，表示对应的方法未实现。

接着用#include <asm/syscalls_64.h>的方式再初始化存在的系统调用。

该include的文件就是上面生成的arch/x86/include/generated/asm/syscalls_64.h，syscalls_64.h文件里调用__SYSCALL_64，为对应的系统下标赋值。

最后，sys_call_table[1] = __x64_sys_write。

自此sys_call_table被填入

3.2 外部中断

中断的入口是用entry_64.S 中的ENTRY(irq_entries_start)，irq_entries_start定义如下：先将vector入栈，然后跳到common_interrupt，重复FIRST_SYSTEM_VECTOR-FIRST_EXTERNAL_VECTOR次（记为n次），可以理解为一次定义了 *n* 个中断处理函数，而irq_entries_start就是一个函数数组。

```
1  ENTRY(irq_entries_start)
2      vector=FIRST_EXTERNAL_VECTOR
3      .rept (FIRST_SYSTEM_VECTOR - FIRST_EXTERNAL_VECTOR)
4      UNWIND_HINT_IRET_REGS
5      pushq   $(~vector+0x80)         /* Note: always in signed byte range */
6      jmp common_interrupt
7      .align  8
8      vector=vector+1
9      .endr
10 END(irq_entries_start)
```

common_interrupt获取中断号取反的值，用SAVE_ALL保存现场，然后调用do_IRQ处理中断，最后执行ret_from_intr返回，代码如下。

```
1  common_interrupt:
2      addq    $-0x80, (%rsp)          /* Adjust vector to [-256, -1] range */
3      call    interrupt_entry
4      UNWIND_HINT_REGS indirect=1
5      call    do_IRQ  /* rdi points to pt_regs */
6      /* 0(%rsp): old RSP */
7  ret_from_intr:
```

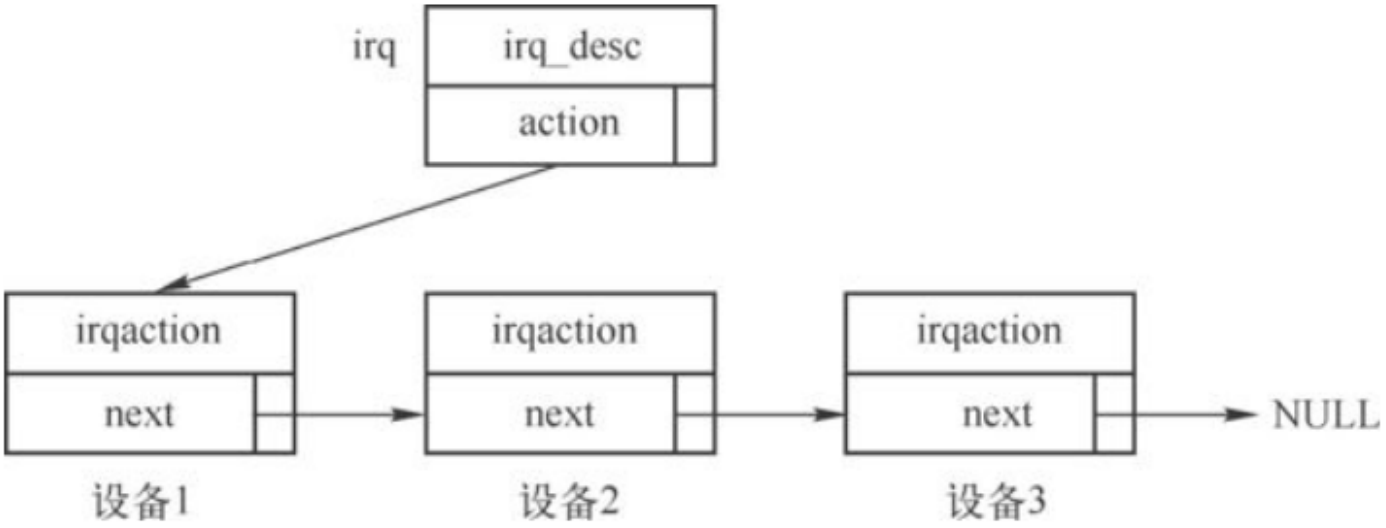
do_IRQ完成中断处理的主要逻辑，如果需要自定义处理函数，可以以它为模板，分为进入处理、处理、退出处理三部分。

```
__visible unsigned int __irq_entry do_IRQ(struct pt_regs *regs)
{
    struct pt_regs *old_regs = set_irq_regs(regs);
    struct irq_desc * desc;
    /* high bit used in ret_from_ code  */
    unsigned vector = ~regs->orig_ax;

    entering_irq();//进入处理

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

    desc = __this_cpu_read(vector_irq[vector]);

    if (!handle_irq(desc, regs)) {//处理
        ack_APIC_irq();

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

    exiting_irq();//退出处理

    set_irq_regs(old_regs);
    return 1;
}
```

entering_irq调用preempt_count_add(HARDIRQ_OFFSET)，表示当前进入硬中断处理。

handle_irq开启了处理之旅，接下来中断服务例程才正式进入视野。

中断服务例程涉及两个关键的结构体，即irq_desc和irqaction，二者是一对多的关系，但并不是每个irq_desc都一定有与之对应的irqaction。Irq_desc与irq号对应，irqaction与一个设备对应，共享同一个irq号的多个设备的irqaction对应同一个irq_desc，如图4-1所示。



irqaction表示设备对中断的处理，主要字段如表4-2所示。

表4-2 irqaction字段表

字　段	类　型	说　明
handler	irq_handler_t	处理中断的函数
thread_fn	irq_handler_t	在独立的线程中执行中断处理时，真正处理中断的函数
thread	task_struct *	对应的线程，不在独立线程中执行中断处理时为 NULL
irq	unsigned int	与 irq_desc 对应的 irq
next	irqaction *	将其链接到链表中

表4-3 irq_desc字段表

字　段	类　型	说　明
action	irqaction *	irqaction 组成的链表的头
Irq _data	Irq _data	芯片相关信息
handle_irq	Irq _flow_handler_t	处理中断的函数

首先，需要引入两个概念。第一个是中断嵌套，顾名思义，一个中断发生的时候，另一个中断到来，需要先处理新中断，处理完毕再返回原中断继续处理。第二个是中断栈，当前内核版本中，默认情况下，中断处理都会优先在中断栈中执行。32位系统中，中断栈分为软中断栈和硬中断栈，分别由每cpu变量hardirq_stack和softirq_stack表示；64位系统中，软硬中断使用同一个中断栈，由每cpu变量irq_stack_ptr表示。另外，软中断也可以在单独的内核线程ksoftirqd中执行，这种情况下就不需要使用软中断栈了。

x86中，中断嵌套时，第一个中断占用了中断栈，那么后面的中断就只能在当前进程的栈中执行了。另一点需要说明的是，x86中，do_IRQ是在当前进程中执行，调用irq_desc的handle_irq时才会切换到中断栈中；x64中，直接切换到中断栈中执行do_IRQ。

handle_irq函数完成溢出检查、找到irq对应的irq_desc对象，并调用它的handle_irq字段定义的回调函数。

两个结构体都有处理中断的函数，中断发生时执行哪个呢？如果对应的irq只有一个设备，且相关的参数都已设置完毕，irq_desc并不一定需要irqaction。实际上，irq_desc的handle_irq是一定会执行的，irqaction的函数一般由handle_irq调用，它们执行与否完全取决于handle_irq的策略。

实例分析：

用一个具体的例子继续分析。键盘和鼠标共享中断引脚，连接到GPIO上，通过GPIO来实现中断，GPIO的中断直接连接到处理器上，假设GPIO的irq号为50，键盘和鼠标的irq号为200，如图4-2所示。

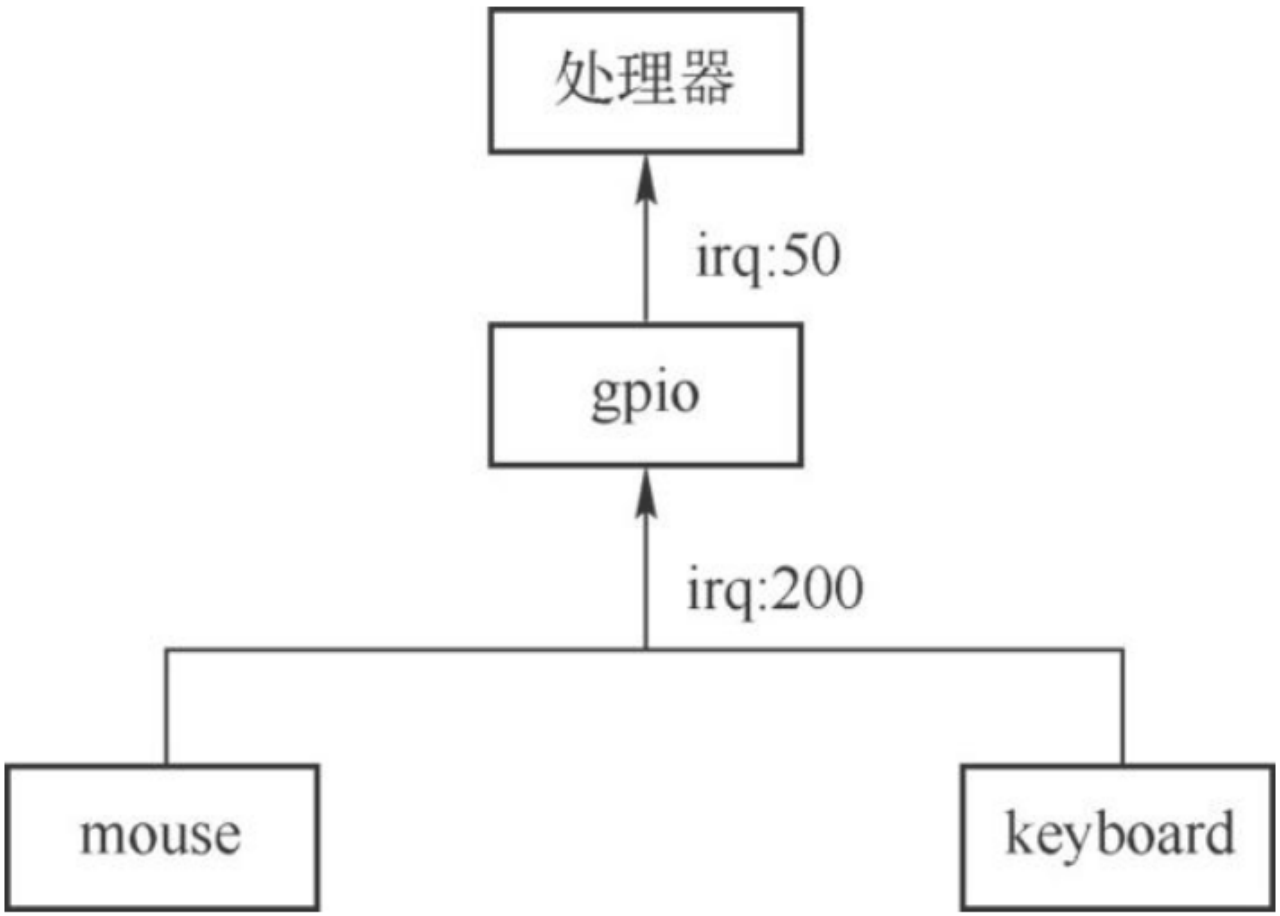


图4-2 中断设备示意图

处理器检测到中断，通过do_IRQ、handle_irq函数调用irq:50对应的this_gpio_handler函数，这是涉及的第一个中断服务例程。

GPIO设置了irq:50的irq_desc的handle_irq字段和irq:200的irq_desc，鼠标和键盘设置了链接到irq:200的irq_desc的irq_action，这个例子的中断处理就可以基本完备了。

处理器检测到中断，通过do_IRQ、handle_irq函数调用irq:50对应的this_gpio_handler函数，这是涉及的第一个中断服务例程。

GPIO在this_gpio_handler中判断是哪个引脚引起了中断，然后将中断处理继续传递至连接到该引脚的设备。generic_handle_irq实际上调用了irq:200的irq_desc的handle_irq字段，也就是GPIO驱动中设置的handle_edge_irq函数。需要说明的是，handle_edge_irq对应的是边沿触发的设备，如果是电平触发，应该是handle_level_irq。

handle_edge_irq有两个重要功能，第一个功能与中断重入有关，如果当前中断处理正在执行，这时候又触发了新一轮的同一个irq的中断，新中断并不会马上得到处理，而是执行desc->istate |= IRQS_PENDING操作。当前中断处理完毕后，会循环检测IRQS_PENDING是否被置位，如果是就继续处理中断。

从另一个角度来讲，即使处理当前中断时多个中断到来，完成当前处理后只会处理一次，这是合理的，毕竟对大多数硬件来讲最新时刻的状态更有意义。不过现实中，这种情况应该深入分析，因为这对某些需要跟踪轨迹设备的用户体验有较大影响。比如鼠标，如果中间几个点丢掉了，光标会从一个位置跳到另一个。

第二个功能就是处理当前中断，我们可以调用handle_irq_event函数实现。该函数会将IRQS_PENDING清零，调用irqd_set将irq_desc的irq_data的IRQD_IRQ_INPROGRESS标记置位，表示正在处理中断。然后调用

handle_irq_event_percpu函数将处理权交给设备，最后清除IRQD_IRQ_INPROGRESS标记。

handle_irq_event_percpu调用__handle_irq_event_percpu，后者遍历irqaction，代码片段如下。

键盘和鼠标的驱动将它们的irqaction链接到了irq_desc，让函数回调它们的handler字段。在我们的例子中，键盘的驱动调用了request_threaded_irq，它的handler参数为NULL，系统默认设置成了irq_default_primary_handler函数，它直接返回IRQ_WAKE_THREAD，所以对于键盘而言，会执行irq_wake_thread唤醒执行中断处理的线程，键盘的handle_keyboard_irq就是在该线程中执行的。而对鼠标而言，驱动调用的是request_irq，handler就是自身的handle_mouse_irq函数，它使用工作队列完成I/O操作和数据报告等任务。

到这里，中断似乎已经得到应有的处理，但内核的工作还没有完成，因为还要退出中断。

exiting_irq调用irq_exit，通过sub_preempt_count(HARDIRQ_OFFSET)减去硬中断标志，如果in_interrupt为false，且当前有软中断需要处理，调用invoke_softirq处理软中断。然后判断是否依然无事可做，如果是，就尝试进入dyntick-idle状态，至此do_IRQ结束。

common_interrupt在调用do_IRQ处理中断后，跳到ret_from_intr，后者判断中断前请求特权等级（RPL），如果是内核态，执行resume_kernel，否则执行resume_userspace，ret_from_exception的实现过程类似于此，代码如下。

中断处理下半部（暂无）：工作队列、tasklet、新建一个线程

3.3 异常

· 异常的入口是用entry_64.S 中的 idtentry等一系列宏定义的。它们共同包含了 idtentry_body宏。

· idtentry_body中首先调用 error_entry 保存环境，再直接调用 cfunc执行中断处理函数，最后用error_return还原寄存器、返回到内核或用户。

· 例如除零错误是这样定义的 /arch/x86/include/asm/idtentry.h

#define DECLARE_IDTENTRY(vector, func) \

idtentry vector asm_##func func has_error_code=0

DECLARE_IDTENTRY(X86_TRAP_DE, exc_divide_error);

· 它通过层层宏定义，最终指向了 exc_divide_error 函数

-> idtentry X86_TRAP_DE asm_exc_divide_error exc_divide_error has_error_code=0

-> idtentry_body exc_divide_error has_error_code=0

-> call exc_divide_error

继续寻找exc_divide_error 函数，发现它定义在 traps.c 中，这里定义了 exc_各种异常形式的函数，它们都指向do_error_trap()；

· 观察 do_trap() 函数，发现它看起来似乎与信号相关。

· 没错，其实就是给对应的中断处理的进程传了个信号。

do_trap()

->force_sig()

->force_sig_info()

->force_sig_info_to_task()

->send_signal()