Linux操作系统分析
Chapter 10 中断和异常

陈香兰（xlanchen@ustc.edu.cn）

计算机应用教研室@计算机学院
嵌入式系统实验室@苏州研究院
中国科学技术大学
Fall 2014

December 30, 2014

Outline

为什么会有中断？

- 内核的一个主要功能就是处理硬件外设I/O
  - 处理器速度一般比外设快很多
  - 轮询方式效率不高
- 内核应当处理其它任务，只有当外设真正完成/准备好了时才转过来处理外设IO
- 中断机制就是满足上述条件的一种解决办法。
  （回忆IO方式：轮询、中断、DMA等）

查看主机系统中断信息

- cat /proc/interrupts

```
/proc/interrupts:
    to display every IRQ vector in use by the system
```

Outline

中断和异常（ARM版）

- 中断（广义）会改变处理器执行指令的顺序，通常与CPU芯片内部或外部硬件电路产生的电信号相对应
  - 中断——异步的：（狭义）
    由硬件随机产生，在程序执行的任何时候可能出现
  - 异常——同步的：（狭义）
    在（特殊的或出错的）指令执行时由CPU控制单元产生

ARM用“异常（exception）”来统称，我们仍然使用”中断信号“

中断信号的作用

- 中断信号提供了一种特殊的方式，使得CPU转去运行正常程序之外的代码
  - 比如一个外设采集到一些数据，发出一个中断信号，CPU必须立刻响应这个信号，否则数据可能丢失
- 当一个中断信号到达时，CPU必须停止它当前正在做的事，并切换到一个新的活动以响应这个中断信号
- 为了做到这这一点，
  在进程的内核态堆栈中保存程序计数器的当前值(即pc寄存器)
  以便处理完中断的时候能正确返回到中断点，并把与中断信号相关的一个地址放入进程序计数器，从而进入中断的处理

中断信号的处理原则

1. 快！
   - 当内核正在做一些别的事情的时候，中断会随时到来。
     无辜的正在运行的代码被打断
   - 中断处理程序在run的时候可能禁止了同级中断
   - 中断处理程序对硬件操作，一般硬件对时间也是非常敏感的
   - 内核的目标就是让中断尽可能快的处理完，尽其所能把更多的处理向后推迟
     - ★ 上半部分(top half)和下半部分(bottom half)
2. 允许不同类型中断的嵌套发生，
   这样能使更多的I/O设备处于忙状态
3. 尽管内核在处理一个中断时可以接受一个新的中断，
   但在内核代码中还在存在一些临界区，
   在临界区中，中断必须被禁止

中断上下文与进程上下文

- 程序的运行必然有上下文，中断处理程序亦然。
- 中断上下文不同于进程上下文
  - 中断或异常处理程序执行的代码不是一个进程
  - 它是一个内核控制路径，执行内核代码，在中断发生时正在运行的进程上下文中执行
  - 作为一个内核控制路径，中断处理程序比一个进程要“轻”(中断上下文只包含了很有限的几个寄存器，建立和终止中断上下文所需要的时间很少)
- 假设：

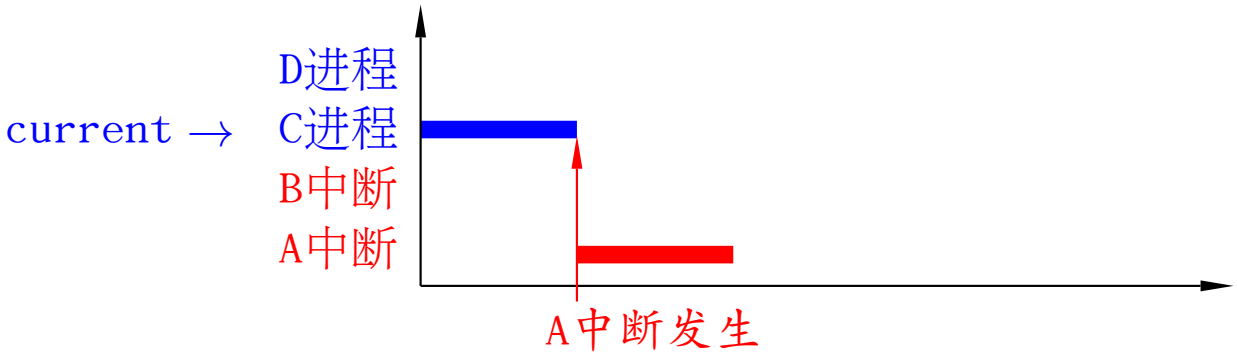
  2个interrupt context，记为A和B

  2个process，记为C和D

  分析A,B,C,D在互相抢占上的关系

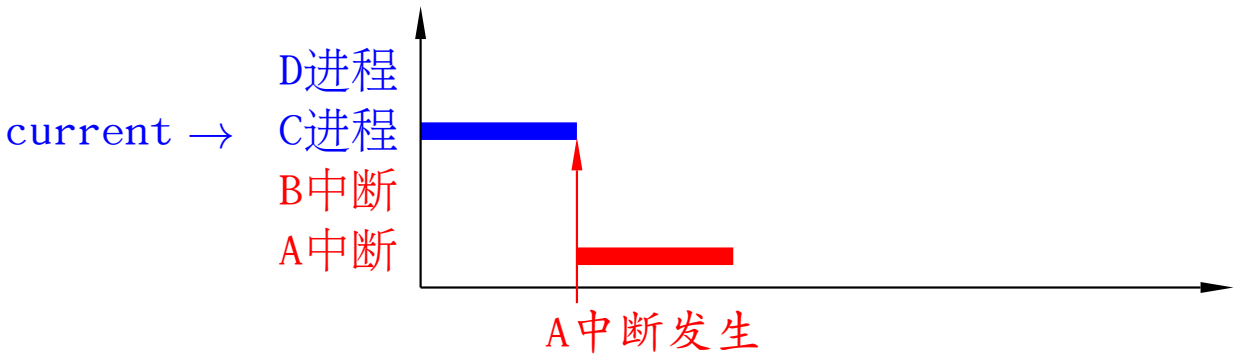
中断上下文与进程上下文

1. 假设某个时刻C占用CPU运行，此时A中断发生，
   则C被A抢占，A得以在CPU上执行。
   由于Linux不为中断处理程序设置process context，A只能使用C的kernel stack作为自己的运行栈



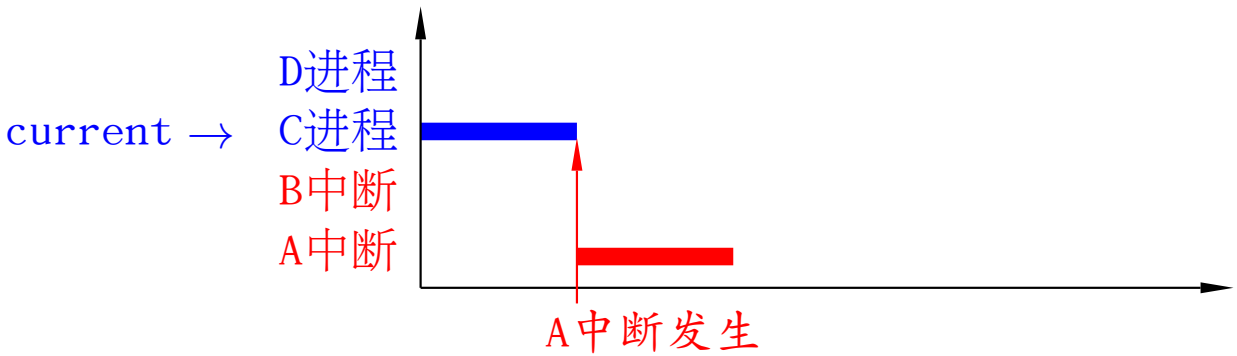
中断上下文与进程上下文

2. 无论如何，Linux的interrupt context A绝对不会被某个进程C或者D抢占！！
   这是由于所有已经启动的interrupt contexts，不管是interrupt contexts之间切换，还是在某个interrupt context中执行代码的过程，决不可能插入scheduler调度例程的调用。
   除非interrupt context主动或者被动阻塞进入睡眠，唤起scheduler，但这是必须避免的，危险性见第3点说明。



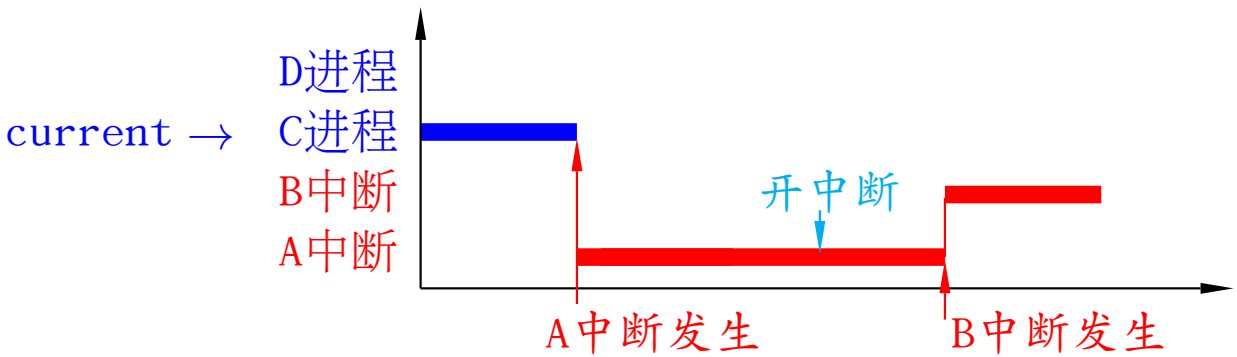
中断上下文与进程上下文

3. 关于第2点的解释：
首先，interrupt context没有process context，A中断是“借”了C的进程上下文运行的，若允许A“阻塞”或“睡眠”，则C将被迫阻塞或睡眠，仅当A被“唤醒”C才被唤醒；而“唤醒”后，A将按照C在就绪队列中的顺序被调度。这既损害了A的利益也污染了C的kernel stack。
其次，如果interrupt context A由于阻塞或是其他原因睡眠，外界对系统的响应能力将变得不可忍受



中断上下文与进程上下文

4. 那么interrupt context A和B的关系又如何呢？
   由于可能在interrupt context的某个步骤打开了CPU的IF flag标志，这使得在A过程中，B的irq line已经触发了PIC，进而触发了CPU IRQ pin，使得CPU执行中断B的interrupt context，这是中断上下文的嵌套过程。
5. 通常Linux不对不同的interrupt contexts设置优先级，这种任意的嵌套是允许的。
   当然可能某个实时Linux的patch会不允许低优先级的interrupt context抢占高优先级的interrupt context



ARM的异常（中断）机制

- 在ARM中，中断／异常／系统调用统称为异常
- ARM的异常有7种，下面是向量表

Table 2-3 Exception processing modes

Exception type	Mode	Normal address	High vector address
Reset	Supervisor	0x00000000	0xFFFF0000
Undefined instructions	Undefined	0x00000004	0xFFFF0004
Software interrupt (SWI)	Supervisor	0x00000008	0xFFFF0008
Prefetch Abort (instruction fetch memory abort)	Abort	0x0000000C	0xFFFF000C
Data Abort (data access memory abort)	Abort	0x00000010	0xFFFF0010
IRQ (interrupt)	IRQ	0x00000018	0xFFFF0018
FIQ (fast interrupt)	FIQ	0x0000001C	0xFFFF001C

ARM的寄存器和模式

Modes						
	Privileged modes					
		Exception modes				
User	System	Supervisor	Abort	Undefined	Interrupt	Fast interrupt
R0	R0	R0	R0	R0	R0	R0
R1	R1	R1	R1	R1	R1	R1
R2	R2	R2	R2	R2	R2	R2
R3	R3	R3	R3	R3	R3	R3
R4	R4	R4	R4	R4	R4	R4
R5	R5	R5	R5	R5	R5	R5
R6	R6	R6	R6	R6	R6	R6
R7	R7	R7	R7	R7	R7	R7
R8	R8	R8	R8	R8	R8	R8_fiq
R9	R9	R9	R9	R9	R9	R9_fiq
R10	R10	R10	R10	R10	R10	R10_fiq
R11	R11	R11	R11	R11	R11	R11_fiq
R12	R12	R12	R12	R12	R12	R12_fiq
R13	R13	R13_svc	R13_abt	R13_und	R13_irq	R13_fiq
R14	R14	R14_svc	R14_abt	R14_und	R14_irq	R14_fiq
PC	PC	PC	PC	PC	PC	PC
CPSR	CPSR	CPSR	CPSR	CPSR	CPSR	CPSR
		SPSR_svc	SPSR_abt	SPSR_und	SPSR_irq	SPSR_fiq

indicates that the normal register used by User or System mode has been replaced by an alternative register specific to the exception mode

ARM异常的优先级

Table 2-4 Exception priorities

Priority		Exception
Highest	1	Reset
	2	Data Abort
	3	FIQ
	4	IRQ
	5	Prefetch Abort
Lowest	6	Undefined instruction SWI

ARM的异常（中断）机制

- 发生异常时

```
R14_<exception_mode> = return link
SPSR_<exception_mode> = CPSR
CPSR[4:0] = exception mode number
CPSR[5] = 0                          /*Execute in ARM state */
if <exception_mode> == Reset or FIQ then
    CPSR[6] = 1                      /* Disable fast interrupts */
/* else CPSR[6] is unchanged */
CPSR[7] = 1                          /* Disable normal interrupt */
PC = exception vector address
```

- 异常返回：

```
R14_<exception_mode> → PC
SPSR_<exception_mode> → CPSR
```

开关中断

```
/*
 * Enable and disable interrupts
 */
#if __LINUX_ARM_ARCH__ >= 6
        .macro disable_irq
        cpsid i
        .endm

        .macro enable_irq
        cpsie i
        .endm
#else
        .macro disable_irq
        msr cpsr_c, #PSR_I_BIT | SVC_MODE
        .endm

        .macro enable_irq
        msr cpsr_c, #SVC_MODE
        .endm
#endif
```

Interrupts are disabled when the I bit in the CPSR is set. If the I bit is clear, ARM checks for an IRQ at instruction boundaries.

Outline

中断和异常处理程序的嵌套执行

- 当内核处理一个中断或异常时，就开始了一个新的内核控制路径
- 当CPU正在执行一个与中断相关的内核控制路径时，linux不允许进程切换。
  不过，一个中断处理程序可以被另外一个中断处理程序中断，这就是中断的嵌套执行
- 抢占原则
  - 普通进程可以被中断或异常处理程序打断
  - 异常处理程序可以被中断程序打断
  - 中断程序只可能被其他的中断程序打断
- Linux允许中断嵌套的原因
  - 提高可编程中断控制器和设备控制器的吞吐量
  - 实现了一种没有优先级的中断模型

Outline

初始化ARM的向量表

- 内核启动中断前，必须初始化ARM的向量表

arch/arm/kernel/entry-armv.S, 每个向量长度为4个字节

```
        .equ stubs_offset, __vectors_start + 0x200 - __stubs_start

        .globl __vectors_start
__vectors_start:
        swi SYS_ERROR0
        b vector_und + stubs_offset
        ldr pc, .LCvswi + stubs_offset
        b vector_pabt + stubs_offset
        b vector_dabt + stubs_offset
        b vector_addrexcptn + stubs_offset
        b vector_irq + stubs_offset
        b vector_fiq + stubs_offset

        .globl __vectors_end
__vectors_end:
```

向量表的初始化

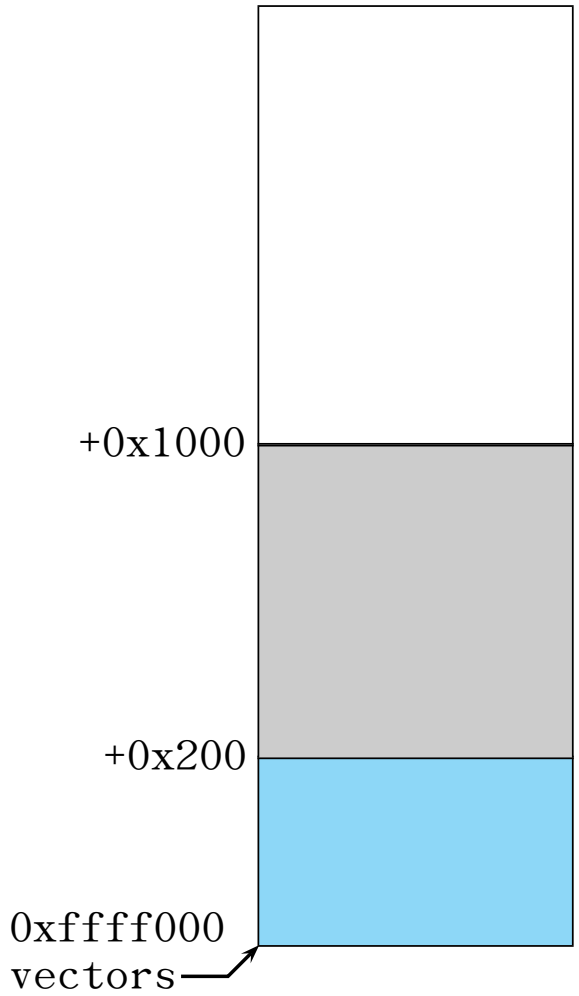
- trap_init@arch/arm/kernel/traps.c中
  （trap_init由start_kernel调用）

```
void __init trap_init(void) {
    unsigned long vectors = CONFIG_VECTORS_BASE;
    extern char __stubs_start[], __stubs_end[];
    extern char __vectors_start[], __vectors_end[];
    extern char __kuser_helper_start[], __kuser_helper_end[];
    int kuser_sz = __kuser_helper_end - __kuser_helper_start;

    /*
     * Copy the vectors, stubs and kuser helpers (in entry-armv.S)
     * into the vector page, mapped at 0xffff0000, and ensure these
     * are visible to the instruction stream.
     */
    memcpy((void *)vectors, __vectors_start, __vectors_end - __vectors_start);
    memcpy((void *)vectors + 0x200, __stubs_start, __stubs_end - __stubs_start);
    memcpy((void *)vectors + 0x1000 - kuser_sz, __kuser_helper_start, kuser_sz);
    ...
```

其中，CONFIG_VECTORS_BASE=0xffff0000

向量表的初始化



Outline

中断的处理vector_irq：普通的中断

- 在ARM手册中：
  1. IRQ为处理器外部中断，IRQ输入; IRQ优先级低于FIQ
  2. Interrupts are disabled when the I bit in the CPSR is set.
  3. If I bit is clear, ARM checks for an IRQ at intruction boundaries.
  4. 检测到有中断发生时：

```
R14_irq = address of next instruction to be executed +4
SPSR_irq = CPSR
CPSR[4:0] = 0b10010      /* Enter IRQ mode */
CPSR[5] = 0              /* Execute in ARM state */
/* CPSR[6] is unchanged */
CPSR[7] = 1              /* Disable normal interrupt */
if high vectors configured then
    PC = 0xFFFF0018
else
    PC = 0x00000018
```

  5. 返回时：

```
subs PC，R14, #4
```

vector_irq的定义

```
        .globl __stubs_start
__stubs_start:
/*
 * Interrupt dispatcher
 */
        vector_stub irq, IRQ_MODE, 4

        .long __irq_usr          @ 0 (USR_26 / USR_32)
        .long __irq_invalid      @ 1 (FIQ_26 / FIQ_32)
        .long __irq_invalid      @ 2 (IRQ_26 / IRQ_32)
        .long __irq_svc          @ 3 (SVC_26 / SVC_32)
        .long __irq_invalid      @ 4
        .long __irq_invalid      @ 5
        .long __irq_invalid      @ 6
        .long __irq_invalid      @ 7
        .long __irq_invalid      @ 8
        .long __irq_invalid      @ 9
        .long __irq_invalid      @ a
        .long __irq_invalid      @ b
        .long __irq_invalid      @ c
        .long __irq_invalid      @ d
        .long __irq_invalid      @ e
        .long __irq_invalid      @ f
```

vector_stub的宏定义 I

```
/*
 * Vector stubs.
 *
 * This code is copied to 0xffff0200 so we can use branches in the
 * vectors, rather than ldr's. Note that this code must not
 * exceed 0x300 bytes.
 *
 * Common stub entry macro:
 * Enter in IRQ mode, spsr = SVC/USR CPSR, lr = SVC/USR PC
 *
 * SP points to a minimal amount of processor-private memory, the address
 * of which is copied into r0 for the mode specific abort handler.
 */
        .macro vector_stub, name, mode, correction=0
        .align 5

vector_\name:
        .if \correction
        sub lr, lr, #\correction
        .endif
```

vector_stub的宏定义 II

```
@
@ Save r0, lr_<exception> (parent PC) and spsr_<exception>
@ (parent CPSR)
@
stmia sp, {r0, lr}      @ save r0, lr
mrs lr, spsr
str lr, [sp, #8]        @ save spsr

@
@ Prepare for SVC32 mode.  IRQs remain disabled.
@
mrs r0, cpsr
eor r0, r0, #(\mode ^ SVC_MODE)
msr spsr_cxsf, r0

@
@ the branch table must immediately follow this code
@
and lr, lr, #0x0f
mov r0, sp
ldr lr, [pc, lr, lsl #2]
movs pc, lr             @ branch to handler in SVC mode
.endm
```

__irq_user

```
        .align 5
__irq_usr:
        usr_entry
        kuser_cmpxchg_check
        ...
        get_thread_info tsk
#ifdef CONFIG_PREEMPT
         ldr r8, [tsk, #TI_PREEMPT] @ get preempt count
        add r7, r8, #1 @ increment it
        str r7, [tsk, #TI_PREEMPT]
#endif

        irq_handler
#ifdef CONFIG_PREEMPT
        ldr r0, [tsk, #TI_PREEMPT]
        str r8, [tsk, #TI_PREEMPT]
        teq r0, r7
        strne r0, [r0, -r0]
#endif
        ...
        mov why, #0
        b ret_to_user
```

ret_to_user

```
/*
 * "slow" syscall return path. "why" tells us if this was a real syscall.
 */
ENTRY(ret_to_user)
ret_slow_syscall:
    disable_irq             @ disable interrupts
    ldr r1, [tsk, #TI_FLAGS]
    tst r1, #_TIF_WORK_MASK
    bne work_pending
no_work_pending:
    /* perform architecture specific actions before user return */
    arch_ret_to_user r1, lr

@ slow_restore_user_regs
    ldr r1, [sp, #S_PSR]            @ get calling cpsr
    ldr lr, [sp, #S_PC]!            @ get pc
    msr spsr_cxsf, r1               @ save in spsr_svc
    ldmdb sp, {r0 - lr}^            @ get calling r0 - lr
    mov r0, r0
    add sp, sp, #S_FRAME_SIZE - S_PC
    movs pc, lr                 @ return & move spsr_svc into cpsr
```

irq_handler

```
/*
 * Interrupt handling. Preserves r7, r8, r9
 */
        .macro irq_handler
        get_irqnr_preamble r5, lr
1:      get_irqnr_and_base r0, r6, r5,
        lr movne r1, sp
        @
        @ routine called with r0 = irq number, r1 = struct pt_regs *
        @
        adrne lr, 1b
        bne asm_do_IRQ
        ...
        .endm
```

__irq_svc

```
        .align 5
__irq_svc:
        svc_entry
        ...
#ifdef CONFIG_PREEMPT
        get_thread_info tsk
        ldr r8, [tsk, #TI_PREEMPT] @ get preempt count
        add r7, r8, #1 @ increment it
        str r7, [tsk, #TI_PREEMPT]
#endif

        irq_handler
#ifdef CONFIG_PREEMPT
        str r8, [tsk, #TI_PREEMPT] @ restore preempt count
        ldr r0, [tsk, #TI_FLAGS] @ get flags
        teq r8, #0 @ if preempt count != 0
        movne r0, #0 @ force flags to 0
        tst r0, #_TIF_NEED_RESCHED
        blne svc_preempt
#endif
        ldr r0, [sp, #S_PSR] @ irqs are already disabled
        msr spsr_cxsf, r0
        ...
        ldmia sp, {r0 - pc}^ @ load r0 - pc, cpsr
```

Outline

中断的处理vector_fiq：快速中断

- 在ARM手册中：
  1. FIQ为处理器外部中断，FIQ输入;
  2. 引入FIQ：降低上下文切换代价，方法是提供足够多的专用寄存器
  3. Fast interrupts are disabled when the F bit in the CPSR is set.
  4. If the F bit is clear, ARM checks for an FIQ at instruction boundaries.
  5. 若检测到FIQ：

```
R14_fiq = address of next instruction to be executed +4
SPSR_fiq = CPSR
CPSR[4:0] = 0b10001     /* Enter IRQ mode */
CPSR[5] = 0              /* Execute in ARM state */
CPSR[6] = 1              /* Disable fast interrupts */
CPSR[7] = 1              /* Disable normal interrupts */
if high vectors configured then
    PC = 0xFFFF001C
else
    PC = 0x0000001C
```

  6. 返回时：

```
subs PC，R14, #4
```

```
/*=============================================================================
 * Undefined FIQs
 *-----------------------------------------------------------------------------
 * Enter in FIQ mode, spsr = ANY CPSR, lr = ANY PC
 * MUST PRESERVE SVC SPSR, but need to switch to SVC mode to show our msg.
 * Basically to switch modes, we *HAVE* to clobber one register... brain
 * damage alert! I don't think that we can execute any code in here in any
 * other mode than FIQ... Ok you can switch to another mode, but you can't
 * get out of that mode without clobbering one register.
 */
vector_fiq:
        disable_fiq
        subs pc, lr, #4
```

set_fiq_handler()

- 在arch/arm/kernel/fiq.c中：

```
void set_fiq_handler(void *start, unsigned int length) {
    memcpy((void *)0xffff001c, start, length);
    flush_icache_range(0xffff001c, 0xffff001c + length);
    if (!vectors_high())
        flush_icache_range(0x1c, 0x1c + length);
}
```

Outline

异常处理1：未定义指令异常

- 未定义指令异常，第2个向量
  - 对于协处理器的指令，arm处理器将等待协处理器来说明是否可以处理，若无响应，则产生未定义指令异常
  - 对于其他未定义指令，也产生该异常。

__und_usr

```
        .align 5
__und_usr:
        usr_entry

        @
        @ fall through to the emulation code, which returns using r9 if
        @ it has emulated the instruction, or the more conventional lr
        @ if we are to treat this as a real undefined instruction
        @
        @ r0 - instruction
        @
        adr r9, ret_from_exception
        adr lr, __und_usr_unknown
        tst r3, #PSR_T_BIT @ Thumb mode?
        subeq r4, r2, #4 @ ARM instr at LR - 4
        subne r4, r2, #2 @ Thumb instr at LR - 2
1:      ldreqt r0, [r4]
        beq call_fpe
        @ Thumb instruction
#if __LINUX_ARM_ARCH__ >= 7
2:      ldrht r5, [r4], #2
        and r0, r5, #0xf800 @ mask bits 111x x... .... ....
        cmp r0, #0xe800 @ 32bit instruction if xx != 0
        blo __und_usr_unknown
3:      ldrht r0, [r4]
        add r2, r2, #2 @ r2 is PC + 2, make it PC + 4
        orr r0, r0, r5, lsl #16
#else
        b __und_usr_unknown
#endif
```

__und_svc

```
        .align 5
__und_svc:
#ifdef CONFIG_KPROBES
        @ If a kprobe is about to simulate a "stmdb sp..." instruction,
        @ it obviously needs free stack space which then will belong to
        @ the saved context.
        svc_entry 64
#else
        svc_entry
#endif
        @
        @ call emulation code, which returns using r9 if it has emulated
        @ the instruction, or the more conventional lr if we are to treat
        @ this as a real undefined instruction
        @
        @ r0 - instruction
        @
        ldr r0, [r2, #-4]
        adr r9, 1f
        bl call_fpe

        mov r0, sp @ struct pt_regs *regs
        bl do_undefinstr
        @
        @ IRQs off again before pulling preserved data off the stack
        @
1:      disable_irq
        @
        @ restore SPSR and restart the instruction
        @
        ldr lr, [sp, #S_PSR] @ Get SVC cpsr
        msr spsr_cxsf, lr
        ldmia sp, {r0 - pc}^ @ Restore SVC registers
```

Outline

中断处理

- 中断跟异常不同，它并不是表示程序出错，而是硬件设备有所动作，所以不是简单地往当前进程发送一个信号就OK的
- 主要有三种类型的中断：
  - I/O设备发出中断请求 √
  - 时钟中断 √
  - 处理器间中断(SMP, Symmetric Multiprocessor)

I/O中断处理

- I/O中断处理程序必须足够灵活以给多个设备同时提供服务
  - 比如几个设备可以共享同一个IRQ线
    (2个8259级联也只能提供15根IRQ线，
    所以外设共享IRQ线是很正常的)

  这就意味着仅仅中断向量解决不了全部问题
- 灵活性以两种不同的方式达到
  1. IRQ共享：
     中断处理程序执行多个中断服务例程(interrupt service routines, ISRs)。
     每个ISR是一个与单独设备(共享IRQ线)相关的函数
  2. IRQ动态分配：
     一条IRQ线在可能的最后时刻才与一个设备相关联

Linux的中断处理原则

- 为了保证系统对外部的响应，一个中断处理程序必须被尽快的完成。因此，把所有的操作都放在中断处理程序中并不合适

Linux中把紧随中断要执行的操作分为三类

1. 紧急的(critical)
   一般关中断运行。
   诸如对PIC应答中断，对PIC或是硬件控制器重新编程，
   或者修改由设备和处理器同时访问的数据

Linux的中断处理原则

2. 非紧急的(noncritical)
   如修改那些只有处理器才会访问的数据结构(例如按下一个键后读扫描码)，这些也要很快完成，因此由中断处理程序立即执行，不过一般在开中断的情况下

3. 非紧急可延迟的(noncritical deferrable)
   如把缓冲区内容拷贝到某个进程的地址空间(例如把键盘缓冲区内容发送到终端处理程序进程)。这些操作可以被延迟较长的时间间隔而不影响内核操作，有兴趣的进程将会等待数据。
   内核用下半部分这样一个机制来在一个更为合适的时机用独立的函数来执行这些操作

Linux的中断处理原则

- 不管引起中断的设备是什么，所有的I/O中断处理程序都执行相同的基本操作
  1. 若有必要，进入核心态
  2. 在内核态堆栈保存上下文（用户态／核心态）
  3. 调用asm_do_IRQ
  4. 恢复上下文
  5. 若有必要，返回用户态

中断处理：asm_do_IRQ()函数

```
/*
 * do_IRQ handles all hardware IRQ's. Decoded IRQs should not
 * come via this function. Instead, they should provide their
 * own 'handler'
 */
asmlinkage void __exception asm_do_IRQ(unsigned int irq, struct pt_regs *regs) {
    struct pt_regs *old_regs = set_irq_regs(regs);
    struct irq_desc *desc = irq_desc + irq;

    /*
     * Some hardware gives randomly wrong interrupts. Rather
     * than crashing, do something sensible.
     */
    if (irq >= NR_IRQS)
        desc = &bad_irq_desc;

    irq_enter();

    desc_handle_irq(irq, desc);

    /* AT91 specific workaround */
    irq_finish(irq);

    irq_exit();
    set_irq_regs(old_regs);
}
```

中断处理：asm_do_IRQ()函数

include/asm-arm/mach/irq.h

```
/*
 * Obsolete inline function for calling irq descriptor handlers.
 */
static inline void desc_handle_irq(unsigned int irq, struct irq_desc *desc) {
        desc->handle_irq(irq, desc);
}
```

Outline

- asm_do_IRQ@arch/arm/kernel/irq.c使用
  与体系结构无关的中断描述符表irq_desc[]

```
/*
 * do_IRQ handles all hardware IRQ's. Decoded IRQs should not
 * come via this function. Instead, they should provide their
 * own 'handler'
 */
asmlinkage void __exception asm_do_IRQ(unsigned int irq, struct pt_regs *regs) {
    struct pt_regs *old_regs = set_irq_regs(regs);
    struct irq_desc *desc = irq_desc + irq;
    ...
```

从Linux中断描述符中找到irq号对应的中断描述符desc

Linux中断描述符表：irq_desc数组

- irq_desc数组包含了NR_IRQS(体系相关)个irq_desc_t描述符。在include/linux/irq.h中声明：

```
extern struct irq_desc irq_desc[NR_IRQS];
```

  - 中断描述符表irq_desc[]的定义和最初的初始化，参见kernel/irq/handle.c

```
struct irq_desc irq_desc[NR_IRQS] __cacheline_aligned_in_smp = {
    [0 ... NR_IRQS-1] = {
        .status = IRQ_DISABLED,
        .chip = &no_irq_chip,
        .handle_irq = handle_bad_irq,
        .depth = 1,
        .lock = __SPIN_LOCK_UNLOCKED(irq_desc->lock),
#ifdef CONFIG_SMP
        .affinity = CPU_MASK_ALL
#endif
    }
};
```

Linux中断描述符表：irq_desc数组

- 中断描述符数据结构irq_desc在include/linux/irq.h中定义

```
/**
 * struct irq_desc - interrupt descriptor
 *
 * @handle_irq: highlevel irq-events handler [if NULL, __do_IRQ()]
 * @chip: low level interrupt hardware access
 * @msi_desc: MSI descriptor
 * @handler_data: per-IRQ data for the irq_chip methods
 * @chip_data: platform-specific per-chip private data for the chip
 * methods, to allow shared chip implementations
 * @action: the irq action chain
 * @status: status information
 * @depth: disable-depth, for nested irq_disable() calls
 * @wake_depth: enable depth, for multiple set_irq_wake() callers
 * @irq_count: stats field to detect stalled irqs
 * @irqs_unhandled: stats field for spurious unhandled interrupts
 * @last_unhandled: aging timer for unhandled count
 * @lock: locking for SMP
 * @affinity: IRQ affinity on SMP
 * @cpu: cpu index useful for balancing
 * @pending_mask: pending rebalanced interrupts
 * @dir: /proc/irq/ procfs entry
 * @affinity_entry: /proc/irq/smp_affinity procfs entry on SMP
 * @name: flow handler name for /proc/interrupts output
 */
```

Linux中断描述符表：irq_desc数组

```
struct irq_desc {
    irq_flow_handler_t handle_irq;
    struct irq_chip *chip;
    struct msi_desc *msi_desc;
    void *handler_data;
    void *chip_data;
    struct irqaction *action; /* IRQ action list */
    unsigned int status; /* IRQ status */
    unsigned int depth; /* nested irq disables */
    unsigned int wake_depth; /* nested wake enables */
    unsigned int irq_count; /* For detecting broken IRQs */
    unsigned int irqs_unhandled;
    unsigned long last_unhandled; /* Aging timer for unhandled count */
    spinlock_t lock;
    ...
    const char *name;
} ____cacheline_internodealigned_in_smp;
```

Linux中断描述符表：irq_desc数组

- Linux中断描述符表irq_desc[]的几个关键数据结构和类型
  1. irqaction数据结构
  2. irq_chip
  3. irq_flow_handler_t

  - 查看相关数据结构
  - 查看irq_desc数组的定义和最初的初始化

irqaction数据结构

- irqaction数据结构，参见include/linux/interrupt.h
  - 用来实现IRQ的共享，维护共享irq的特定设备和特定中断，所有共享一个irq的链接在一个action表中，由中断描述符中的action指针指向

```
struct irqaction {
    irq_handler_t handler;
    unsigned long flags;
    cpumask_t mask;
    const char *name;
    void *dev_id;
    struct irqaction *next;
    int irq;
    struct proc_dir_entry *dir;
};
```

  - 设置irqaction的函数：kernel/irq/manage.c::setup_irq

irq_chip数据结构

- 为特定PIC编写的低级I/O例程，参见include/linux/irq.h

```
/**
 * struct irq_chip - hardware interrupt chip descriptor
 *
 * @name: name for /proc/interrupts
 * @startup: start up the interrupt (defaults to ->enable if NULL)
 * @shutdown: shut down the interrupt (defaults to ->disable if NULL)
 * @enable: enable the interrupt (defaults to chip->unmask if NULL)
 * @disable: disable the interrupt (defaults to chip->mask if NULL)
 * @ack: start of a new interrupt
 * @mask: mask an interrupt source
 * @mask_ack: ack and mask an interrupt source
 * @unmask: unmask an interrupt source
 * @eoi: end of interrupt - chip level
 * @end: end of interrupt - flow level
 * @set_affinity: set the CPU affinity on SMP machines
 * @retrigger: resend an IRQ to the CPU
 * @set_type: set the flow type (IRQ_TYPE_LEVEL/etc.) of an IRQ
 * @set_wake: enable/disable power-management wake-on of an IRQ
 *
 * @release: release function solely used by UML
 * @typename: obsoleted by name, kept as migration helper
 */
```

irq_chip数据结构

- 为特定PIC编写的低级I/O例程，参见include/linux/irq.h

```
struct irq_chip {
    const char *name;
    unsigned int (*startup)(unsigned int irq);
    void (*shutdown)(unsigned int irq);
    void (*enable)(unsigned int irq);
    void (*disable)(unsigned int irq);
    void (*ack)(unsigned int irq);
    void (*mask)(unsigned int irq);
    void (*mask_ack)(unsigned int irq);
    void (*unmask)(unsigned int irq);
    void (*eoi)(unsigned int irq);
    void (*end)(unsigned int irq);
    void (*set_affinity)(unsigned int irq, cpumask_t dest);
    int  (*retrigger)(unsigned int irq);
    int  (*set_type)(unsigned int irq, unsigned int flow_type);
    int  (*set_wake)(unsigned int irq, unsigned int on);
    ...
};
```

irq_chip数据结构

- 例如arch/arm/plat-s3c24xx/irq.c中

```
struct irq_chip s3c_irq_level_chip = {
        .name = "s3c-level",
        .ack = s3c_irq_maskack,
        .mask = s3c_irq_mask,
        .unmask = s3c_irq_unmask,
        .set_wake = s3c_irq_wake
};

struct irq_chip s3c_irq_chip = {
        .name = "s3c",
        .ack = s3c_irq_ack,
        .mask = s3c_irq_mask,
        .unmask = s3c_irq_unmask,
        .set_wake = s3c_irq_wake
};
```

irq_chip数据结构

- 为一个中断设置irq_chip的函数有
  - set_irq_chip_and_handler_name
    - set_irq_chip
    - 等

例如，Init_IRQ的中断初始化

```
void __init init_IRQ(void) {
    int irq;

    for (irq = 0; irq < NR_IRQS; irq++)
        irq_desc[irq].status |= IRQ_NOREQUEST | IRQ_NOPROBE;

#ifdef CONFIG_SMP
    bad_irq_desc.affinity = CPU_MASK_ALL;
    bad_irq_desc.cpu = smp_processor_id();
#endif
    init_arch_irq();
}
```

grep -r init_arch_irq arch/arm/ -I

```
arch/arm/kernel/setup.c:        init_arch_irq = mdesc->init_irq;
arch/arm/kernel/irq.c:void (*init_arch_irq)(void) __initdata = NULL;
arch/arm/kernel/irq.c:  init_arch_irq();
```

例如，Init_IRQ的中断初始化

- 在ach/arm/kernel/setup.c中

```
void __init setup_arch(char **cmdline_p) {
    struct tag *tags = (struct tag *)&init_tags;
    struct machine_desc *mdesc;
    char *from = default_command_line;

    setup_processor();
    mdesc = setup_machine(machine_arch_type);
    machine_name = mdesc->name;
    ...
    /*
     * Set up various architecture-specific pointers
     */
    init_arch_irq = mdesc->init_irq;
    system_timer = mdesc->timer;
    init_machine = mdesc->init_machine;
    ...
}
```

例如，Init_IRQ的中断初始化

```
static struct machine_desc * __init
setup_machine(unsigned int nr) {
    struct machine_desc *list;

    /*
     * locate machine in the list of supported machines.
     */
    list = lookup_machine_type(nr);
    if (!list) {
        printk(" Machine configuration botched (nr %d), unable "
            " to continue.\n" , nr);
        while (1);
    }

    printk(" Machine: %s\n" , list->name);

    return list;
}
```

关于machine_arch_type

- 以s3c2410为例

在include/asm-arm/mach-types.h中：

```
...
...
#define MACH_TYPE_S3C2410 182
...
...
#ifdef CONFIG_ARCH_S3C2410
# ifdef machine_arch_type
# undef machine_arch_type
# define machine_arch_type __machine_arch_type
# else
# define machine_arch_type MACH_TYPE_S3C2410
# endif
# define machine_is_s3c2410() (machine_arch_type == MACH_TYPE_S3C2410)
#else
# define machine_is_s3c2410() (0)
#endif
```

关于machine_desc

include/asm-arm/mach/arch.h

```
struct machine_desc {
    /*
     * Note! The first four elements are used
     * by assembler code in head.S, head-common.S
     */
    unsigned int        nr;             /* architecture number  */
    unsigned int        phys_io;        /* start of physical io */
    unsigned int        io_pg_offst;    /* byte offset for io
                                         * page tabe entry      */

    const char          *name;          /* architecture name    */
    unsigned long       boot_params;    /* tagged list          */

    unsigned int        video_start;    /* start of video RAM   */
    unsigned int        video_end;      /* end of video RAM     */

    unsigned int        reserve_lp0 :1; /* never has lp0        */
    unsigned int        reserve_lp1 :1; /* never has lp1        */
    unsigned int        reserve_lp2 :1; /* never has lp2        */
    unsigned int        soft_reboot :1; /* soft reboot          */
    void                (*fixup)(struct machine_desc *,
                                 struct tag *, char **,
                                 struct meminfo *);
    void                (*map_io)(void);/* IO mapping function  */
    void                (*init_irq)(void);
    struct sys_timer    *timer;         /* system tick timer    */
    void                (*init_machine)(void);
};
```

关于machine_desc

```
/*
 * Set of macros to define architecture features. This is built into
 * a table by the linker.
 */
#define MACHINE_START(_type,_name)                          \
static const struct machine_desc __mach_desc_##_type       \
 __used                                                     \
 __attribute__((__section__(" .arch.info.init" ))) = {      \
        .nr = MACH_TYPE_##_type,                            \
        .name = _name,

#define MACHINE_END                 \
};
```

```
grep -r " MACHINE_START"  |grep s3c2410
```

```
arch/arm/mach-s3c2410/mach-bast.c:MACHINE_START(BAST, " Simtec-BAST" )
arch/arm/mach-s3c2410/mach-tct_hammer.c:MACHINE_START(TCT_HAMMER, " TCT_HAMMER" )
arch/arm/mach-s3c2410/mach-qt2410.c:MACHINE_START(QT2410, " QT2410" )
arch/arm/mach-s3c2410/mach-n30.c:MACHINE_START(N30, " Acer-N30" )
arch/arm/mach-s3c2410/mach-vr1000.c:MACHINE_START(VR1000, " Thorcom-VR1000" )
arch/arm/mach-s3c2410/mach-h1940.c:MACHINE_START(H1940, " IPAQ-H1940" )
arch/arm/mach-s3c2410/mach-smdk2410.c:MACHINE_START(SMDK2410, " SMDK2410" ) /* @TODO:
request a new identifier and switch
arch/arm/mach-s3c2410/mach-otom.c:MACHINE_START(OTOM, " Nex Vision - Otom 1.1" )
arch/arm/mach-s3c2410/mach-am1m5900.c:MACHINE_START(AML_M5900, " AML_M5900" )
```

以smdk2410为例

arch/arm/mach-s3c2410/mach-smdk2410.c

```
MACHINE_START(SMDK2410, "SMDK2410") /* @TODO: request a new identifier and switch
                                     * to SMDK2410 */
    /* Maintainer: Jonas Dietsche */
    .phys_io = S3C2410_PA_UART,
    .io_pg_offst = (((u32)S3C24XX_VA_UART) >> 18) & 0xfffc,
    .boot_params = S3C2410_SDRAM_PA + 0x100,
    .map_io = smdk2410_map_io,
    .init_irq = s3c24xx_init_irq,
    .init_machine = smdk2410_init,
    .timer = &s3c24xx_timer,
MACHINE_END
```

- 关于s3c24xx_init_irq，阅读arch/arm/plat-s3c24xx/irq.c

irq_desc的第一个数据项：irq_flow_handler_t handle_irq

- __set_irq_handler设置handle_irq数据项
- 具体函数由kernel/irq/chip.c定义
  - handle_simple_irq
  - handle_level_irq
  - handle_fasteoi_irq
  - handle_edge_irq
  - handle_percpu_irq

irq_desc的第一个数据项：irq_flow_handler_t handle_irq

- 以s3c24xx_init_irq为例：

```
switch (irqno) {
/* deal with the special IRQs (cascaded) */
case IRQ_EINT4t7:
case IRQ_EINT8t23:
case IRQ_UART0:
case IRQ_UART1:
case IRQ_UART2:
case IRQ_ADCPARENT:
    set_irq_chip(irqno, &s3c_irq_level_chip);
    set_irq_handler(irqno, handle_level_irq);
    break;
case IRQ_RESERVED6:
case IRQ_RESERVED24:
    /* no IRQ here */
    break;
default:
    //irqdbf(" registering irq %d (s3c irq)\n" , irqno);
    set_irq_chip(irqno, &s3c_irq_chip);
    set_irq_handler(irqno, handle_edge_irq);
    set_irq_flags(irqno, IRQF_VALID);
}
```

设备相关的中断处理例程：action->handle

- 在setup_irq时，给定
- 例如，在arch/arm/plat-s3c24xx/time.c中

```
/*
 * IRQ handler for the timer
 */
static irqreturn_t s3c2410_timer_interrupt(int irq, void *dev_id) {
    timer_tick();
    return IRQ_HANDLED;
}

static struct irqaction s3c2410_timer_irq = {
    .name = "S3C2410 Timer Tick",
    .flags = IRQF_DISABLED | IRQF_TIMER | IRQF_IRQPOLL,
    .handler = s3c2410_timer_interrupt,
};
...
static void __init s3c2410_timer_init (void) {
    s3c2410_timer_setup();
    setup_irq(IRQ_TIMER4, &s3c2410_timer_irq);
}
```

中断处理关键流程

1. asm_do_IRQ()函数调用desc_handle_irq()
2. desc_handle_irq()调用desc->handle_irq()

- 几个handle_irq方法共同调用的关键函数
  - handle_IRQ_event()

kernel/irq/handle.c::handle_IRQ_evnet()和中断服务例程

- 一个中断服务例程实现一种特定设备的操作，handle_IRQ_event()函数依次调用这些设备例程
  - 这个函数的核心代码：

```
do {
    ret = action->handler(irq, action->dev_id);
    if (ret == IRQ_HANDLED)
        status |= action->flags;
    retval |= ret;
    action = action->next;
} while (action);
```

小结：中断处理过程

- 根据需要进入核心态
- 保存上下文
- 调用asm_do_IRQ
  - 根据中断号，找到中断处理函数处理
- 恢复上下文
- 根据需要返回用户态

Outline

软中断、tasklet以及下半部分

- 对内核来讲，可延迟中断不是很紧急，可以将它们从中断处理例程中抽取出来，保证较短的中断响应时间
- Linux2.6提供了三种方法来执行可延迟操作（函数）：
  1. 软中断
     - ★ 软中断由内核静态分配（编译时确定）
  2. tasklet
     - ★ Tasklet在软中断之上实现
     - ★ 一般原则：在同一个CPU上软中断/tasklet不嵌套
     - ★ Tasklet可以在运行时分配和初始化（例如装入一个内核模块时）
  3. 工作队列（ work queues ）

软中断、tasklet以及下半部分

- 一般而言，可延迟函数上可以执行4种操作
  1. 初始化：
     定义一个新的可延迟函数，通常在内核初始化时进行
  2. 激活：
     设置可延迟函数在下一轮处理中执行
  3. 屏蔽：
     有选择的屏蔽一个可延迟函数，这样即使被激活也不会被运行
  4. 执行：
     在特定的时间执行可延迟函数

Outline

软中断softirq管理的关键数据结构

- Linux2.6.26使用有限个软中断，
  具体参见include/linux/interrupt.h

```
enum
{
    HI_SOFTIRQ=0,
    TIMER_SOFTIRQ,
    NET_TX_SOFTIRQ,
    NET_RX_SOFTIRQ,
    BLOCK_SOFTIRQ,
    TASKLET_SOFTIRQ,
    SCHED_SOFTIRQ,
#ifdef CONFIG_HIGH_RES_TIMERS
    HRTIMER_SOFTIRQ,
#endif
    RCU_SOFTIRQ, /* Preferable RCU should always be the last softirq */
};
```

  - 软中断的编号越小，优先级越高

软中断softirq管理的关键数据结构

- 软中断的向量表softirq_vec定义，参见kernel/softirq.c

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

- 每一个软中断向量由数据结构softirq_action定义，参见include/linux/interrupt.h

```
struct softirq_action {
    void (*action)(struct softirq_action *);
    void *data;
};
```

软中断的主要操作

- 软中断的关键操作包括
  1. 软中断向量的初始化
  2. 软中断的触发
  3. 软中断的屏蔽和激活？？local_bh_enable/disable
  4. 软中断的检查和处理

1、软中断向量的初始化

- 初始化函数为kernel/softirq.c::open_softirq

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

例如

在kernel/softirq.c::softirq_init中：

```
open_softirq(TASKLET_SOFTIRQ, tasklet_action, NULL);
open_softirq(HI_SOFTIRQ, tasklet_hi_action, NULL);
```

在net/core/dev.c::net_dev_init中：

```
open_softirq(NET_TX_SOFTIRQ, net_tx_action, NULL);
open_softirq(NET_RX_SOFTIRQ, net_rx_action, NULL);
```

2、软中断的触发

- 软中断触发函数为raise_softirq，它对所属CPU的irq_stat::__softirq_pending中的对应位置1

irq_stat定义，参见include/asm－x86/hardirq_32.h

```
typedef struct {
    unsigned int __softirq_pending;
...
} ____cacheline_aligned irq_cpustat_t;

DECLARE_PER_CPU(irq_cpustat_t, irq_stat);
```

  - raise_softirq
    →raise_softirq_irqoff
    →include/linux/interrupt.h::__raise_softirq_irqoff

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

4、软中断的检查和处理 I

- 宏local_softirq_pending()用于访问所属CPU的irq_stat::__softirq_pending域
- 在某些特定的时机，会读取该域的值以检查是否有软中断被挂起，若有软中断被挂起，就调用do_softirq 。
  这种时机，称为检查点，例如
  1. 调用local_bh_enable重新激活软中断时

4、软中断的检查和处理 II

2. 当asm_do_IRQ完成了I/O中断的处理时调用irq_exit

kernel/softirq.c

```
#ifdef __ARCH_IRQ_EXIT_IRQS_DISABLED
# define invoke_softirq() __do_softirq()
#else
# define invoke_softirq() do_softirq()
#endif

/*
 * Exit an interrupt context. Process softirqs if needed and possible:
 */
void irq_exit(void) {
    ...
    if (!in_interrupt() && local_softirq_pending())
        invoke_softirq();
    ...
```

3. 当那个特定的进程ksoftirqd被唤醒时
4. ...

- 在每个检查点，若有软中断被挂起，就调用do_softirq

4、软中断的检查和处理 III

- 软中断处理的核心代码是__do_softirq

- 在处理完一轮软中断之后，若发现又有新的软中断被激活，则再次执行软中断处理主流程；
  重复执行超过一定次数，就唤醒ksoftirqd进程继续处理

4、软中断的检查和处理 IV

```
asmlinkage void __do_softirq(void) {
    ...
restart: /* Reset the pending bitmask before enabling irqs */
    set_softirq_pending(0);
    local_irq_enable();
    h = softirq_vec;
    do {
        if (pending & 1) {
            h->action(h);
            rcu_bh_qsctr_inc(cpu);
        }
        h++;
        pending >>= 1;
    } while (pending);
    local_irq_disable();
    pending = local_softirq_pending();
    if (pending && --max_restart)
        goto restart;
    if (pending) wakeup_softirqd();
    ...
}
```

Ksoftirqd内核线程

- Linux在初始化时调用kernel/softirq.c::spawn_ksoftirqd()为每一个逻辑CPU创建一个ksoftirqd内核线程
  static DEFINE_PER_CPU(struct task_struct *, ksoftirqd);
- 创建ksoftirqd的关键代码为：

```
p = kthread_create(ksoftirqd, hcpu, "ksoftirqd/%d", hotcpu);
...
kthread_bind(p, hotcpu);
per_cpu(ksoftirqd, hotcpu) = p;
```

Ksoftirqd内核线程

- 必要时，ksoftirqd内核线程会被唤醒

```
/*
 * we cannot loop indefinitely here to avoid userspace starvation,
 * but we also don't want to introduce a worst case 1/HZ latency
 * to the pending events, so lets the scheduler to balance
 * the softirq load for us.
 */
static inline void wakeup_softirqd(void) {
    /* Interrupts are disabled: no need to stop preemption */
    struct task_struct *tsk = __get_cpu_var(ksoftirqd);
    if (tsk && tsk->state != TASK_RUNNING)
        wake_up_process(tsk);
}
```

- 例如，在do_softifq()开中断处理完若干次pending的软中断之后，若发现又有新的软中断pending，则唤醒ksoftirqd内核线程

```
if (pending)
    wakeup_softirqd();
```

Ksoftirqd内核线程

- ksoftirqd内核线程的核心工作是：

```
while (local_softirq_pending()) {
    ...
    do_softirq();
    ...
}
```

- 使用ps -lef|grep
  soft查看ksoftirqd的信息，例如进程号、优先级

Outline

Tasklet

- Tasklet是I/O驱动程序中实现可延迟函数的首选方法。
  在软中断之上实现。

1. tasklet的数据结构及其组织
2. tasklet的使用及相关接口
3. tasklet的处理

1、tasklet的数据结构及其组织

1. tasklet的数据结构：tasklet_struct
   参见include/linux/interrupt.h

```
struct tasklet_struct {
    struct tasklet_struct *next;
    unsigned long state;
    atomic_t count;
    void (*func)(unsigned long);
    unsigned long data;
};
```

- state值：tasklet的状态

```
enum {
    TASKLET_STATE_SCHED, /* Tasklet is scheduled for execution */
    TASKLET_STATE_RUN /* Tasklet is running (SMP only) */
};
```

- count值：tasklet的屏蔽情况
  - ★ >0：被屏蔽
  - ★ =0：没有被屏蔽

1、tasklet的数据结构及其组织

2. tasklet的组织：Tasklet和高优先级的tasklet
   - 分别存放在每个CPU的tasklet_vec和tasklet_hi_vec链表中
   - 链表头定义如下，参见kernel/softirq.c

```
/* Tasklets */
struct tasklet_head {
    struct tasklet_struct *head;
    struct tasklet_struct **tail;
};
/* Some compilers disobey section attribute on statics when not initialized — RR */
static DEFINE_PER_CPU(struct tasklet_head, tasklet_vec) = { NULL };
static DEFINE_PER_CPU(struct tasklet_head, tasklet_hi_vec) = { NULL };
```

   - 这两个percpu链表在kernel/softirq.c::softirq_init中进行初始化。

2、Tasklet的使用及相关接口

- 当需要使用tasklet时，可以按照如下方法进行
  1. 分配一个tasklet的数据结构，并初始化
     ====相当于声明（定义）一个tasklet
  2. 可以禁止/允许这个tasklet
     ====相当于定义了一个是否允许使用tasklet的窗口
  3. 可以激活这个tasklet
     ====这个tasklet被插入tasklet_vec或者tasklet_hi_vec的相应CPU的链表上，将在合适的时机得到处理

1. tasklet的初始化：kernel/softirq::tasklet_init

2、Tasklet的使用及相关接口

2. tasklet的禁止和允许：屏蔽情况
   - tasklet_enable
     - ★ tasklet_struct::count减1
   - tasklet_disable和tasklet_disable_nosync
     - ★ tasklet_struct::count加1
   - 参见include/linux/interrupt.h，对于一个已经被激活的tasklet
     - ★ count值大于0，表示tasklet被屏蔽，不能处理它
     - ★ count值等于0，处理

2、Tasklet的使用及相关接口

3. tasklet的激活
   - 调用tasklet_schedule或者tasklet_hi_schedule将tasket插入到对应的链表中
   - 原则1：已经在链表中的tasklet，即状态为TASKLET_STATE_SCHED，不再重复插入
   - 原则2：激活一个tasklet时，还需要触发相关软中断，使得tasklet有机会得到处理

3、tasklet的处理

- tasklet分别在软中断TASKLET_SOFTIRQ和HI_SOFTIRQ中得到处理
- tasklet_action和tasklet_hi_action分别是tasklet_vec和tasklet_hi_vec链表中的tasklet的处理函数
  - 找到CPU对应的那个项，遍历执行
- 它们分别被注册为软中断的软中断处理函数，参见kernel/softirq.c::softirq_init

```
open_softirq(TASKLET_SOFTIRQ, tasklet_action, NULL);
open_softirq(HI_SOFTIRQ, tasklet_hi_action, NULL);
```

Outline

工作队列和工作线程

- 相关数据结构
  - 参见kernel/workqueue.c:
    - ★ workqueue_struct；cpu_workqueue_struct
  - 参见include/linux/workqueue.h
    - ★ work_struct；delayed_work
- 入列，参见kernel/workqueue.c
  - queue_work；queue_delayed_work
- 工作队列的处理
  - run_workqueue
  - worker_thread
- create_workqueue用来创建一个工作者队列
  - 查看Linux-2.6.26中，工作者队列的创建情况
  - 进一步查看工作者队列的实例“events”的使用情况

从中断和异常返回（阅读源码）

- 中断和异常的终止目的很清楚，即恢复某个程序的执行，但是还有几个
  1. 内核控制路径是否嵌套
     - ★ 如果仅仅只有一条内核控制路径，那CPU必须切换到用户态
  2. 挂起进程的切换请求
     - ★ 如果有任何请求，必须调度；否则，当前进程得以运行
  3. 挂起的信号
     - ★ 如果一个信号发送到进程，那必须处理它
  4. 等等

Thanks !

The end.