粤嵌教育

Linux中断处理

www.gec-edu.org

中断的概念

粤嵌教育

中断源——引起中断的原因，或者说发出中断请求的来源叫做中断源。 一般是指由外设发出的中断请求，如：键盘中断、打印机中断等。外部中断是可以屏蔽的中断，也就是说，利用中断控制器可以屏蔽这些外部设备的中断请求。

异常——指因硬件出错（如复位、存储器访问等）或软件出错（如SWI、数据处理、未定义指令等）所引起的中断，也叫异常。异常中断是不可屏蔽的中断。

中断嵌套 -- CPU在处理级别较低的中断过程中，出现了级别较高的中断请求，CPU停止执行低级别中断的处理程序而去优先处理高级别中断，等高级别中断处理完毕后，再接着执行低级别的未处理完的中断处理程序，这种中断处理方式称为中断嵌套。使用中断嵌套可以使高优先级别的中断得到及时的响应和处理。

中断的作用

粤嵌教育

- 中断的作用

-- 提高系统的效率,由于CPU速度远比外设高,轮询的效率非常低下。

-- 中断使得单CPU的系统中可以执行多任务,它是实现现代操作系统的基础。

-- 中断也是产生并发和同步问题的根源。

ARM的中断向量表

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下的Linux中断

- 触发IRQ，CPU jump 到0x18，同时要把irqno传入相应的寄存器，调用一个中断通用处理函数：asm_do_IRQ(unsigned int irqno)。asm_do_IRQ() 这个函数根据irqno 就可以找到对应的中断描述符，然后调用中断描述符里面的handler()了。

- 中断向量表的初始化
- arch/arm/kernel/entry-armv.S ——中断向量表放在这个文件里：
- .globl __vectors_start
- __vectors_start:
- ARM(swi SYS_ERROR0)
- THUMB(svc #0)
- THUMB(nop)
- W(b) vector_und + stubs_offset
- W(ldr) pc, .LCvswi + stubs_offset
- W(b) vector_pabt + stubs_offset
- W(b) vector_dabt + stubs_offset
- W(b) vector_addrexcptn + stubs_offset
- W(b) vector_irq + stubs_offset
- W(b) vector_fiq + stubs_offset
- .globl __vectors_end
- __vectors_end:

ARM下的Linux中断

- ARM linux内核启动时，通过start_kernel()->trap_init()的调用，初始化内核的中断异常向量表。

- CONFIG_VECTORS_BASE是一个宏，用来获取ARM异常向量的地址
- linux/arch/arm/kernel/traps.c

```
//复制异常中断向量表和各个异常中断对应的处理代码
void __init early_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;
        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);
        memcpy((void *)KERN_SIGRETURN_CODE, sigreturn_codes,
              sizeof(sigreturn_codes));

        flush_icache_range(vectors, vectors + PAGE_SIZE);
        modify_domain(DOMAIN_USER, DOMAIN_CLIENT);
}
```

ARM下的Linux中断

- CONFIG_VECTORS_BASE是一个宏，用来获取ARM异常向量的地址，该宏在/arch/arm/include/asm/system.h中定义：

- #define CPU_ARCH_ARMv5 4
- #define CR_V (1 << 13)
- extern unsigned long cr_no_alignment;
- extern unsigned long cr_alignment;

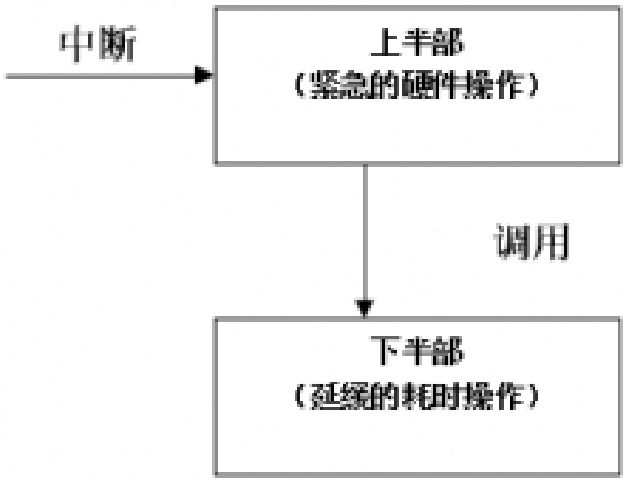
- #if __LINUX_ARM_ARCH__ >= 4
- #define vectors_high() (cr_alignment & CR_V)
- #else
- #define vectors_high() (0)
- #endif

- 对于ARMv4以下的版本，这个地址固定为0；
- ARMv4及其以上的版本，ARM异常向量表的地址受协处理器CP15的c1寄存器中V位(bit[13])控制:
- 如果V=0，则异常向量表的地址为:0x00000000~0x0000001C；
- 如果V=1，则异常向量表的地址为:0xFFFF0000~0xFFFF001C。

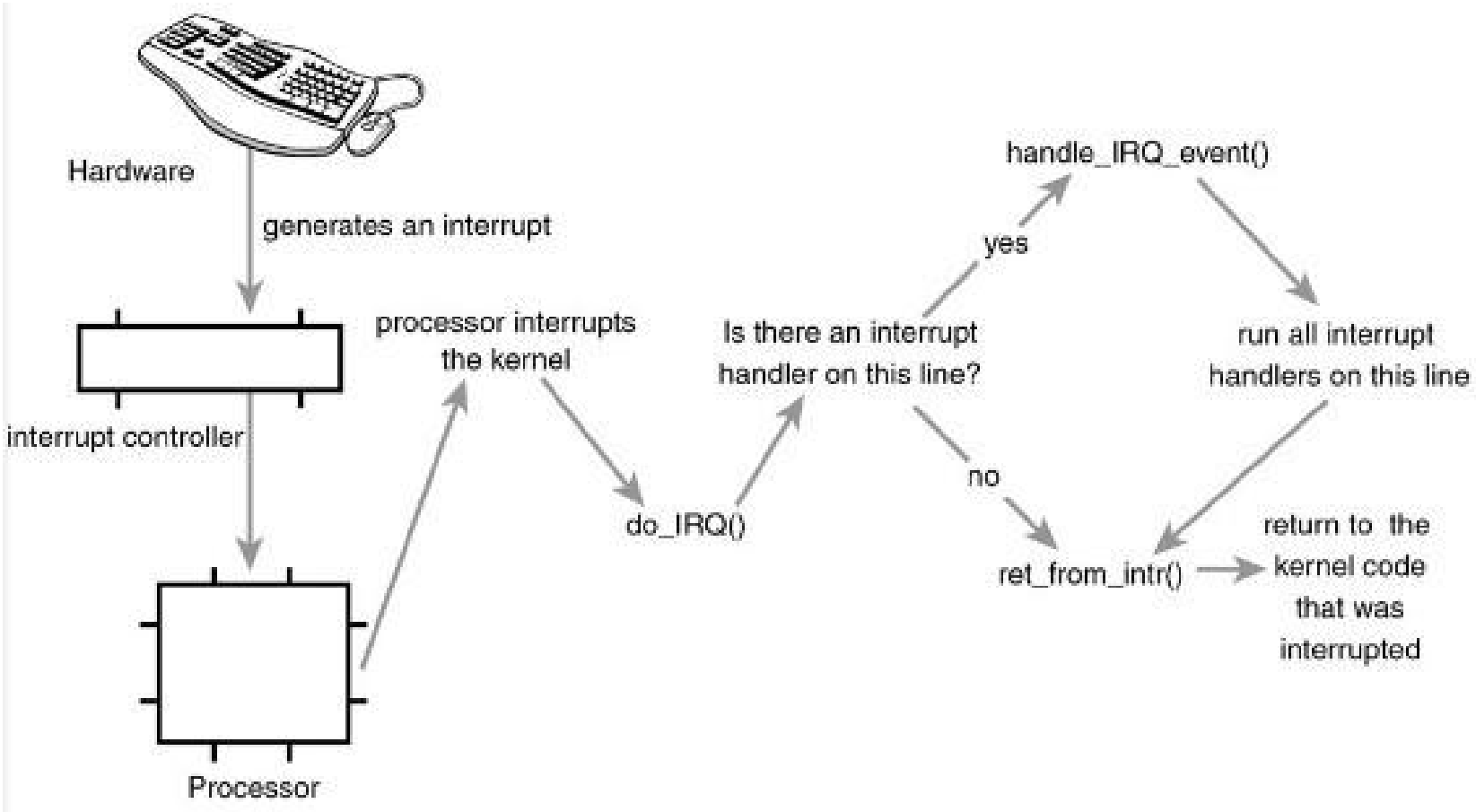
中断处理

粤嵌教育

-- Linux把中断处理分为两半部分，也就是所谓的上下半部的处理。
-- 上半部处理非常紧急的事情，如从硬件读数据，恢复硬件的状态等，这部分要非常快完成，在这部分处理中，对应的中断被屏蔽。
-- 下半部处理指把中断处理中不非常紧急的处理延后到一个合适的时间执行，如把读到的数据放进队列，唤醒等待的进程等。
--上半部与下半部的处理的主要区别在于中断是否被屏蔽，下半部主要由上半部安装调度。



中断处理


Hardware
generates an interrupt
interrupt controller
Processor
processor interrupts
the kernel
do_IRQ()
Is there an interrupt
handler on this line?
yes
handle_IRQ_event()
run all interrupt
handlers on this line
no
ret_from_intr()
return to the
kernel code
that was
interrupted

struct irq_desc


```
struct irq_desc {
        struct irq_data                 irq_data;
        struct timer_rand_state         *timer_rand_state;
        unsigned int __percpu           *kstat_irqs;
        irq_flow_handler_t              handle_irq;
#ifdef CONFIG_IRQ_PREFLOW_FASTEOI
        irq_preflow_handler_t           preflow_handler;
#endif
        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                    irq_count;      /* For detecting broken IRQs */
        unsigned long                   last_unhandled; /* Aging timer for unhandled count */
        unsigned int                    irqs_unhandled;
        raw_spinlock_t                  lock;
#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_PROC_FS
        struct proc_dir_entry           *dir;
#endif
        const char                      *name;
} ____cacheline_internodealigned_in_smp;
```

struct irq_desc

- **在include/linux/irq.h中定义**
- 在内核中，每个中断向量都有相应的有一个irq_desc结构体。所有这样的描述符组织在一起形成irq_desc[NR_IRQS]数组。

- irq_data：主要用来保存中断号irq和chip相关的数据；

- struct irq_data {
- unsigned int irq; //中断号
- unsigned int node;
- unsigned int state_use_accessors;
- struct irq_chip *chip; //底层中断控制器描述符
- void *handler_data;
- void *chip_data;
- struct msi_desc *msi_desc;
- #ifdef CONFIG_SMP
- cpumask_var_t affinity;
- #endif
- };

struct irq_desc

- *kstat_irqs：用于系统中断的统计计数。
- handle_irq：是个函数指针，指向该IRQ线的公共服务程序。
- action：针对一某具体设备的中断处理对象，设备驱动程序会通过request_irq向其中挂载设备特定的中断处理函数。它指向一个单链表，该单链表是由该中断线上所有中断服务程序（对应struct irqaction）所连接起来的。
- status_use_accessors ：描述中断线当前的状态；由一组位掩码组成，如：IRQS_ONESHOT、IRQS_WAITING和IRQS_PENDING等。
- Lock：自旋锁。
- *name： /proc/interrupts 中显示的中断名称；
- depth：中断线被激活时，值为0；其值为正数时，表示被禁止的次数；
- irq_count：记录该中断线发生中断的次数；
- irqs_unhandled：该IRQ线上未处理中断发生的次数；

struct irqaction

- 当多个设备共享一条IRQ线时，因为每个设备都要有各自的ISR。为了能够正确处理此条IRQ线上的中断处理程序（也就是区分每个设备），就需要使用irqaction结构体。在这个结构体中，会有专门的handler字段指向该设备的真正的ISR。共享同一条IRQ线上的多个这样的结构体会连接成了一个单链表，即所谓的中断请求队列。中断产生时，该IRQ线的中断请求队列上所有的ISR都会被依次调用。

- irqaction结构具体的定义如下：
- 在include/linux/interrupt.h中定义
- struct irqaction {
- irq_handler_t handler;
- unsigned long flags;
- void *dev_id;
- struct irqaction *next;
- int irq;
- irq_handler_t thread_fn;
- struct task_struct *thread;
- unsigned long thread_flags;
- unsigned long thread_mask;
- const char *name;
- struct proc_dir_entry *dir;
- } ____cacheline_internodealigned_in_smp;

struct irqaction

- handler：指向一个具体的硬件设备的中断服务例程，可以从此指针的类型发现与前文我们所定义的中断处理函数声明相同；
- flags：对应request_irq函数中所传递的第三个参数，可取IRQF_DISABLED、IRQF_SAMPLE_RANDOM和IRQF_SHARED其中之一；
- name：对应于request_irq函数中所传递的第四个参数，可通过/proc/interrupts文件查看到；
- next：指向下一个irqaction结构体；
- dev_id：对应于request_irq函数中所传递的第五个参数，可取任意值，但必须唯一能够代表发出中断请求的设备，通常取描述该设备的结构体；
- irq：中断号
- 如果一个IRQ线上有中断请求，那么内核将依次次调用在该中断线上注册的每一个中断服务程序，但是并不是所有中断服务程序都被执行。一般硬件设备都会提供一个状态寄存器，以便中断服务程序进行检查是否应该为这个硬件服务。也就是说在整个中断请求队列中，最多会有一个ISR被执行，也就是该ISR对应的那个设备产生了中断请求时；不过当该IRQ线上某个设备未找到匹配的ISR时，那这个中断就不会被处理。此时irq_desc结构中的irqs_unhandled字段就会加1。

struct irq_chip


```
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,const struct cpumask *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);

        /* Currently used only by UML, might disappear one day.*/
#ifdef CONFIG_IRQ_RELEASE_METHOD
        void                            (*release)(unsigned int irq, void *dev_id);
#endif
        /*
         * For compatibility, ->typename is copied into ->name.
         * Will disappear.
         */
        const char      *typename;
};
```

struct irq_chip

粤嵌教育

- struct irq_chip是一个中断控制器的描述符。通常不同的体系结构就有一套自己的中断处理方式。内核为了统一的处理中断，提供了底层的中断处理抽象接口，对于每个平台都需要实现底层的接口函数。这样对于上层的中断通用处理程序就无需任何改动。

- 比如s5pv210的外部中断控制器s5p_irq_eint。
- linux/arch/arm/plat-s5p/irq-eint.c

```
static struct irq_chip s5p_irq_eint = {
        .name           = "s5p-eint",
        .mask           = s5p_irq_eint_mask,
        .unmask         = s5p_irq_eint_unmask,
        .mask_ack       = s5p_irq_eint_maskack,
        .ack            = s5p_irq_eint_ack,
        .set_type       = s5p_irq_eint_set_type,
#ifdef CONFIG_PM
        .set_wake       = s3c_irqext_wake,
#endif
};
```

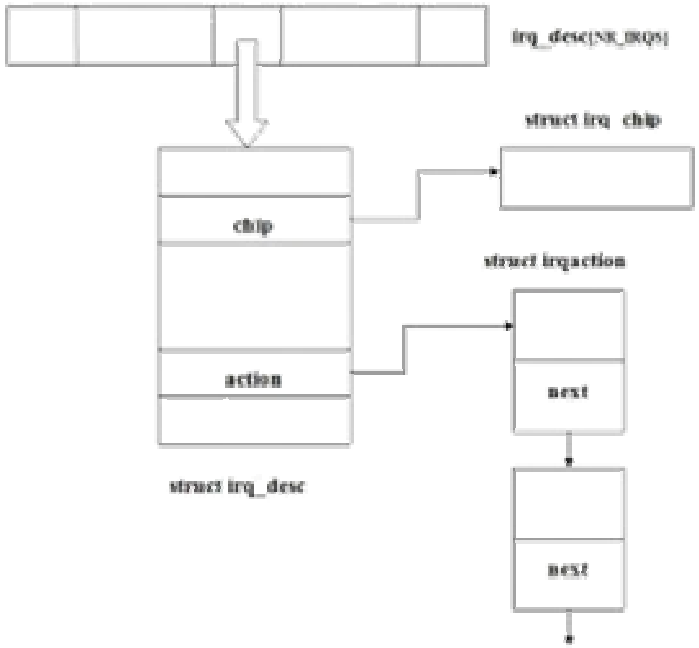
struct irq_chip

粤嵌教育

```
00031: static inline void s5p_irq_eint_mask(struct irq_data *data)
00032: {
00033:     u32 mask;
00034:
00035:     mask = __raw_readl(S5P_EINT_MASK(EINT_REG_NR(data->irq)));
00036:     mask |= eint_irq_to_bit(data->irq);
00037:     __raw_writel(mask, S5P_EINT_MASK(EINT_REG_NR(data->irq)));
00038: }
```

```
00040: static void s5p_irq_eint_unmask(struct irq_data *data)
00041: {
00042:     u32 mask;
00043:
00044:     mask = __raw_readl(S5P_EINT_MASK(EINT_REG_NR(data->irq)));
00045:     mask &= ~(eint_irq_to_bit(data->irq));
00046:     __raw_writel(mask, S5P_EINT_MASK(EINT_REG_NR(data->irq)));
00047: }
```

三个数据结构的关系



中断处理程序的调用过程：在asm_do_IRQ函数中，通过对irq_desc结构体中handler_irq字段的引用，调用handler_irq所指向的公共服务程序；在这个公共服务程序中会调用hand_IRQ_event函数；在hand_IRQ_event函数中，通过对irqaction结构体中handler字段的引用最终调用我们所写的中断处理程序。

struct irq_chip描述了中断最底层的部分；而struct irqacton则描述最上层具体的中断处理函数；而与中断向量所对应的struct irq_desc则类似一个中间层，将中断中的硬件相关的部分和软件相关的部分连接起来。

IRQ中断申请


```
#inclue <linux/interrupt.h>
static inline int __must_check
request_irq(unsigned int irq, irq_handler_t handler, unsigned long flags,
                    const char *name, void * dev_id)
```

- irq----要申请的硬件中断号
- handle----r是向系统登记的中断处理程序(上半部)，是一个回调函数，中断发生时，系统调用它，将dev_id参数传递给它。
- irqflags----中断处理的属性,可以指定中断的触发方式和处理方式。
  - 触发方式:

IRQF_TRIGGER_NONE 、IRQF_TRIGGER_RISING、IRQF_TRIGGER_FALLING 、
IRQF_TRIGGER_HIGH、IRQF_TRIGGER_LOW、IRQ_TYPE_EDGE_BOTH

  - 处理方式:

IRQF_DISABLED - keep irqs disabled when calling the action handler.
IRQF_SAMPLE_RANDOM - irq is used to feed the random generator
IRQF_SHARED - allow sharing the irq among several devices
IRQF_TIMER - Flag to mark this interrupt as timer interrupt
IRQF_NO_SUSPEND - Do not disable this IRQ during suspend
IRQF_NO_THREAD - Interrupt cannot be threaded
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

IRQ中断释放

- name----设置中断名称，在cat /proc/interrupts中可以看到此名称。
- void * dev_id ----用作共享中断线的指针。一般设置为这个设备的设备结构体或者NULL。它是一个独特的标识, 用在当释放中断线时以及可能还被驱动用来指向它自己的私有数据区，来标识哪个设备在中断 。这个参数在真正的驱动程序中一般是指向设备数据结构的指针。在调用中断处理程序的时候它就会传递给中断处理程序的void * dev_id 。如果中断没有被共享， dev_id可以设置为 NULL。

- request_irq的返回值：
  - 为0表示成功，
  - 返回-EINVAL表示中断号无效，
  - 返回-EBUSY表示中断已经被占用，且不能共享

释放IRQ

```
void free_irq(unsigned int irq,void *dev_id);
```

中断处理程序

1、中断处理程序的限制：

在中断时间内运行，不能向用户空间发送或者接收数据。
不能做任何导致休眠的操作。
不能调用schedule函数。
无论快速还是慢速中断处理例程，都应该设计成执行时间尽可能短。

2、上半部的中断处理函数的参数和返回值

irqreturn_t (*handler)(int irq, void *dev_id)

irq----中断号
dev_id----驱动程序可用的数据区，通常可传递指向描述设备的数据结构指针。

typedef irqreturn_t (*irq_handler_t)(int, void *);

```
/**
* enum irqreturn
* @IRQ_NONE              interrupt was not from this device
* @IRQ_HANDLED           interrupt was handled by this device
* @IRQ_WAKE_THREAD       handler requests to wake the handler thread
*/
enum irqreturn {
        IRQ_NONE                = (0 << 0),
        IRQ_HANDLED             = (1 << 0),
        IRQ_WAKE_THREAD         = (1 << 1),};
```

中断信息

- /proc/interrupts反映系统的中断信息
  - 第一列是 IRQ 号
  - 给出每个中断线发生中断的次数。
  - 给出处理中断的可编程中断控制器。
  - 给出在该中断号上注册中断处理例程的设备名称。

- proc/stat记录了几个关于系统活动的底层统计信息, 包括(但不仅限于)自系统启动以来收到的中断数。

中断启动和禁用

粤嵌教育

- 驱动禁止和使能特定的中断
  - #include <asm/irq.h>.
  - void disable_irq(unsigned int irq);
  - irq为中断号，使所选择的中断线无效。使一个中断栈无效。这个函数要等待任何挂起的处理程序在退出之前已经完成。如果你在使用这个函数，同时还持有IRQ处理程序可能需要的一个资源，那么，你就可能死锁。要小心地从IRQ的上下文中调用这个函数。
  - void enable_irq(unsigned int irq);
  - irq为中断号，重新启用这条IRQ线上的中断处理。可以在IRQ的上下文中调用这个函数。
  - void disable_irq_nosync(unsigned int irq);
  - irq为中断号，使所选择的中断线无效。使一个中断栈无效。与disable_ irq不同，这个函数并不确保IRQ处理程序的现有实例在退出前已经完成，将立即关闭中断。可以从IRQ的上下文中调用该函数。

中断启动和禁用

- #include <asm/irq.h>.

- **禁止所有中断**
  - void local_irq_save(unsigned long flags);

    将把当前中断状态保存到flags中，然后禁用当前处理器上的中断发送。注意, flags 被直接传递, 而不是通过指针来传递。
  - void local_irq_disable(void);

    不保存状态而关闭本地处理器上的中断发送; 只有我们知道中断并未在其他地方被禁用的情况下，才能使用这个版本。
- **打开所有中断**
  - void local_irq_restore(unsigned long flags);

    恢复由 local_irq_save 存储于 flags 的状态, 而 local_irq_enable 无条件打开中断.
  - void local_irq_enable(void);

中断号

- **S5PV210外部中断中断号**
- 在头文件： linux/arch/arm/plat-s5p/include/plat/irqs.h
- #define IRQ_EINT(x)　　((x) < 16 ? ((x) + S5P_EINT_BASE1) \
  : ((x) - 16 + S5P_EINT_BASE2))
- x----0~31。

  - #define S5P_EINT_BASE1　　(S5P_IRQ_VIC0(0))
  - #define S5P_IRQ_VIC0(x)　　(S5P_VIC0_BASE + (x))
  - #define S5P_VIC0_BASE　　S5P_IRQ(0)
  - #define S5P_IRQ(x)　　((x) + S5P_IRQ_OFFSET)
  - #define S5P_IRQ_OFFSET　　(32)

  - #define S5P_EINT_BASE2　　(IRQ_VIC_END + 1)
  - #define IRQ_VIC_END　　S5P_IRQ_VIC3(31)
  - #define S5P_IRQ_VIC3(x)　　(S5P_VIC3_BASE + (x))
  - #define S5P_VIC3_BASE　　S5P_IRQ(96)

中断号

粤嵌教育

- **S5PV210 timer中断号**
- 在头文件： linux/arch/arm/plat-s5p/include/plat/irqs.h
- #define IRQ_TIMER0 S5P_TIMER_IRQ(0)
- #define IRQ_TIMER1 S5P_TIMER_IRQ(1)
- #define IRQ_TIMER2 S5P_TIMER_IRQ(2)
- #define IRQ_TIMER3 S5P_TIMER_IRQ(3)
- #define IRQ_TIMER4 S5P_TIMER_IRQ(4)

  - S5P_TIMER_IRQ(x) (11 + (x))

中断号

粤嵌教育

- **S5PV210 UART号**
- 在头文件： linux/arch/arm/plat-s5p/include/plat/irqs.h
- #define IRQ_S3CUART_RX0 IRQ_S5P_UART_RX0
- #define IRQ_S3CUART_RX1 IRQ_S5P_UART_RX1
- #define IRQ_S3CUART_RX2 IRQ_S5P_UART_RX2
- #define IRQ_S3CUART_RX3 IRQ_S5P_UART_RX3

  - #define IRQ_S5P_UART_RX0 (IRQ_S5P_UART_BASE0 +UART_IRQ_RXD)
  - #define IRQ_S5P_UART_BASE0 (16)
  - #define UART_IRQ_RXD (0)

中断号

- **S5PV210 MMC中断号**
- /* linux/arch/arm/mach-s5pv210/include/mach/irqs.h
- #define IRQ_HSMMC0 S5P_IRQ_VIC1(26)
- #define IRQ_HSMMC1 S5P_IRQ_VIC1(27)
- #define IRQ_HSMMC2 S5P_IRQ_VIC1(28)

  - #define S5P_IRQ_VIC1(x) (S5P_VIC1_BASE + (x))
  - #define S5P_VIC1_BASE S5P_IRQ(32)
  - #define S5P_IRQ(x) ((x) + S5P_IRQ_OFFSET)
  - #define S5P_IRQ_OFFSET (32)

中断处理过程

1. 发生中断时，ARM执行异常向量vector_irq的代码。
2. 在vertor_irq内，最终会调用中断处理的总入口函数asm_do_IRQ。
3. asm_do_IRQ根据中断号调用irq_desc数组项的handle_irq。
4. handle_irq会使用chip成员中的函数来设置硬件，如清中断、禁止中断、重新使能中断等。
5. handle_irq调用irq_action中注册的处理函数handler。

中断处理程序的局限性

- 负责对硬件做出迅速响应并完成时间要求很严格的操作
  - 异步方式执行，可能会打断其他重要代码的执行
  - 中断处理程序执行过程中
    - 最好情形下，同级中断被屏蔽
    - 最坏情形下，当前处理器上所有其他中断都会被屏蔽
  - 中断处理程序不在进程上下文中运行，不能被阻塞
- 中断处理程序
  - 上半部：中断处理程序
    - 简单快速，执行时禁止部分或全部中断
  - 下半部：推后工作的机制
    - 执行与中断处理密切相关但中断处理程序本身不执行的工作
  - 执行期间可以响应中断

中断处理

-- Linux把中断处理程序分成了两部分：

- 上半部：实际响应中断的处理程序。
- 下半部：被上半部调用，通过开中断的方式进行。

  - 下半部两种机制实现：
    - Tasklet
    - 工作队列work queue

- 上半部的功能是“登记中断”，当一个中断发生时，它进行相应地硬件读写后就把中断例程的下半部挂到该设备的下半部执行队列中去。因此，上半部执行的速度就会很快，可以服务更多的中断请求。
- 但是，仅有"登记中断"是远远不够的，因为中断的事件可能很复杂。因此，Linux引入了一个下半部，来完成中断事件的绝大多数使命。
- 下半部和上半部最大的不同是下半部是可中断的，而上半部是不可中断的，下半部几乎做了中断处理程序所有的事情，而且可以被新的中断打断！下半部则相对来说并不是非常紧急的，通常还是比较耗时的，因此由系统自行安排运行时机，不在中断服务上下文中执行。

tasklet的实现

tasklet数据结构为struct tasklet_struct，每一个结构体代表一个独立的小任务，在<linux/interrupt.h>定义如下

n

```
struct tasklet_struct
{
    struct tasklet_struct *next;/*指向下一个链表结构*/
    unsigned long state;/*小任务状态*/
    atomic_t count;/*引用计数器*/
    void (*func)(unsigned long);/*小任务的处理函数*/
    unsigned long data;/*传递小任务函数的参数*/
};
```

n state的取值参照下边的枚举型：

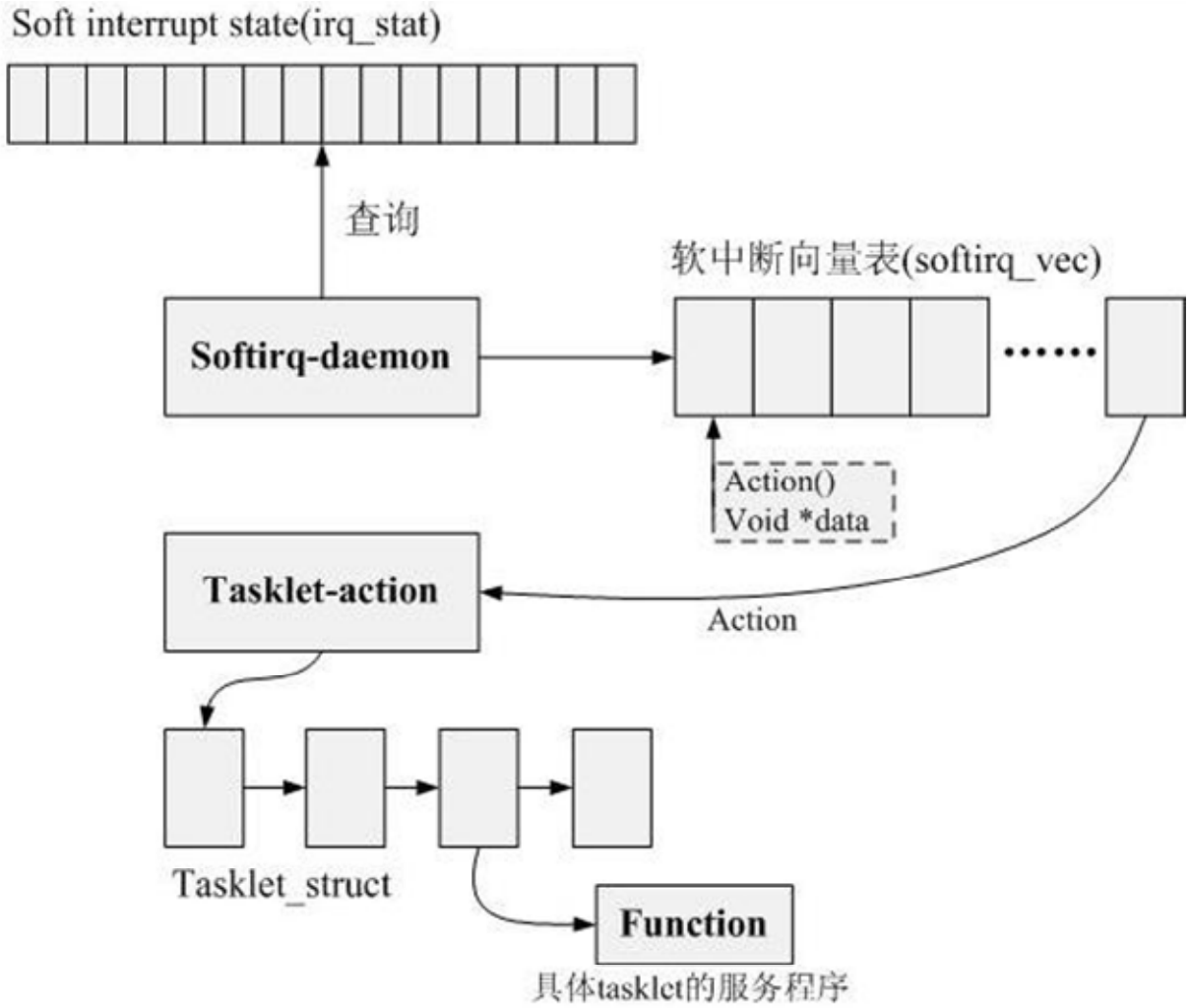
```
enum
{
    TASKLET_STATE_SCHED,    /* 小任务已被调度，等待运行*/
    TASKLET_STATE_RUN    /*正在运行，仅在多处理器上使用*/
};
```

n count域是小任务的引用计数器。只有当它的值为0的时候才能被激活，并其被设置为挂起状态时，才能够被执行，否则为禁止状态。

tasklet运行

- ksoftirqd是一个后台运行的内核线程，它会周期的遍历软中断的向量列表，如果发现哪个软中断向量被挂起了，就执行对应的处理函数。
- tasklet 所对应的处理函数就是tasklet_action，这个处理函数在系统启动时初始化软中断时，就在软中断向量表中注册。
- tasklet_action() 遍历一个全局的 tasklet_vec 链表。链表中的元素为 tasklet_struct结构体。

tasklet运行


Soft interrupt state(irq_stat)
查询
软中断向量表(softirq_vec)
Softirq-daemon
Action()
Void *data
Tasklet-action
Action
Tasklet_struct
Function
具体tasklet的服务程序

tasklet程序设计

#include <linux/interrupt.h>

- 1、声明和使用小任务tasklet
- 静态的创建一个tasklet：
  - #define **DECLARE_TASKLET(name, func, data)** \
    struct tasklet_struct name = { NULL, 0, ATOMIC_INIT(0), func, data }
  - #define **DECLARE_TASKLET_DISABLED**(name, func, data) \
    struct tasklet_struct name = { NULL, 0, ATOMIC_INIT(1), func, data }
- name 是 tasklet 的名字，
- func 是执行 tasklet 的函数；
- data 是 unsigned long 类型的 func 参数。
- 这两个宏的区别在于计数器设置的初始值不同，前者为0，后者为1。为0的表示激活状态，为1的表示禁止状态。

- 动态创建一个tasklet
- void tasklet_init(struct tasklet_struct *t,
  void (*func)(unsigned long), unsigned long data);
- 例：
  - static struct tasklet_struct my_tasklet;
  - tasklet_init(&my_tasklet,tasklet_handler,0);

tasklet程序设计

- 2、小任务处理函数程序
- void tasklet_handler(unsigned long data)

- tasklet应用注意
  - tasklet不能睡眠，不能在tasklet_handler内部使用信号量或其他阻塞式函数。
  - 由于tasklet运行时允许响应中断，如果tasklet与中断处理程序之间共享了某些数据，必须做好预防工作（锁机制）。
  - 不允许两个两个相同类型的tasklet同时执行 。
  - 一个tasklet不会抢占另外一个tasklet 。
  - 索引号小的tasklet在索引号大的tasklet之前执行。
  - tasklet被调度以后，只要有机会它就会尽可能早地运行。在它还没有得到运行机会之前，如果有一个相同的tasklet又被唤醒了，那么它只会运行一次 。

tasklet程序设计

- 3、调度tasklet

- 调度小任务时引用调度函数就能使系统在合适的时候进行调度。函数原型为：
- static inline void tasklet_schedule(struct tasklet_struct *t)

```
{
    if (!test_and_set_bit(TASKLET_STATE_SCHED, &t->state))
        __tasklet_schedule(t);
}
```

  - 调度执行指定的tasklet。
  - 将定义后的 tasklet 挂接到 cpu 的 tasklet_vec 链表。而且会引起一个软 tasklet 的软中断，既把 tasklet 对应的中断向量挂起 (pend) 。
- 调度函数放在中断处理的上半部处理函数中，这样中断申请的时候调用处理函数（即irq_handler_t irq_handler）后，转去执行下半部的小任务。

- 4、删除tasklet
- tasklet_kill(&tasklet);

从挂起的队列中去掉一个tasklet。在处理一个经常重新调度它自身的tasklet的时候，从挂起的队列中移去已调度的tasklet会很有用。这个函数首先等待该tasklet执行完毕，然后再将它移去。由于该函数可能会引起休眠，所以禁止在中断上下文中使用它。

tasklet程序设计

- 使用tasklet作为下半部的处理中断的设备驱动程序模板如下：

- /*中断处理下半部*/
- void my_do_tasklet(unsigned long)

```
{
        ……/*编写处理事件方法*/
}
```

- /*定义tasklet和下半部函数并关联*/
- DECLARE_TASKLET(my_tasklet, my_do_tasklet, 0);

- /*中断处理上半部*/
- irpreturn_t my_interrupt(unsigned int irq,void *dev_id)

```
{
        ……
        /*调度my_tasklet函数，根据声明将去执行my_tasklet_func函数*/
        tasklet_schedule(&my_tasklet)
        ……
}
```

tasklet程序设计

- /*设备驱动的加载函数*/
- int xxx_open(…)

```
{
        ……
        /*申请中断, 转去执行my_interrupt函数并传入参数*/
        result=request_irq(my_irq,my_interrupt,IRQF_DISABLED,"xxx",NULL);
        ……
}
```

- /*设备驱动模块的卸载函数*/
- void   xxx_release(…)

```
{
        ……
        /*释放中断*/
        free_irq(my_irq,my_interrupt);
        ……
}
```

tasklet程序设计

- void tasklet_enable(struct tasklet_struct *t)

  调用tasklet_enable()函数可以激活一个tasklet，要激活DECLARE_TASKLET_DISABLED()创建的tasklet，也要调用这个函数。

- void tasklet_disable(struct tasklet_struct *t)

  tasklet_disable()函数用来禁止某个指定的tasklet。如果该tasklet当前正在执行，这个函数会等到它执行完毕再返回。

- void tasklet_disable_nosync(struct tasklet_struct *t)

  tasklet_disable_nosync()函数也可以用来禁止指定的tasklet，不过它无需在返回前等待tasklet执行完毕。这样做往往不太安全，因为我们无法估计该tasklet是否仍在执行。

- void tasklet_kill(struct tasklet_struct *t)

  通过调用tasklet_kill()函数从挂起的队列中去掉一个tasklet。该函数的参数是一个指向某个tasklet的tasklet_struct的长指针。在处理一个经常重新调度它自身的tasklet的时候，从挂起的队列中移去已调度的tasklet会很有用。这个函数首先等待该tasklet执行完毕，然后再将它移去。由于该函数可能会引起休眠，所以禁止在中断上下文中使用它。

workqueue工作队列

- 工作队列（work queue）是另外一种将中断的部分工作推后的一种方式，它可以实现一些tasklet不能实现的工作，比如工作队列机制可以睡眠。

- 工作队列将推后的工作交给一个称之为工作者线程（worker thread）的内核线程去完成（单核下一般会交给默认的线程events/0）。因此，在该机制中，当内核在执行中断的剩余工作时就处在进程上下文（process context）中。

- 工作队列可以重新调度甚至睡眠。
- 对于tasklet机制（中断处理程序也是如此），内核在执行时处于中断上下文（interrupt context）中。而中断上下文与进程毫无瓜葛，所以在中断上下文中不能睡眠。

- 因此，当推后的那部分中断程序需要睡眠时，工作队列毫无疑问是最佳选择；否则用tasklet。

工作、工作队列和工作者线程

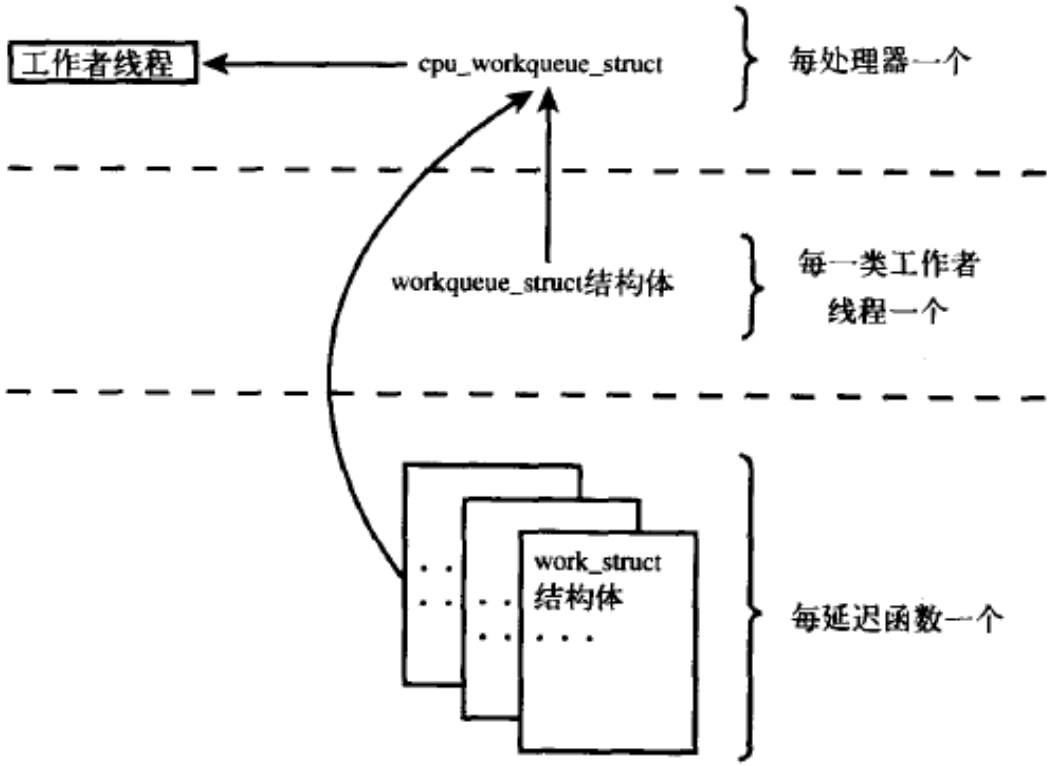
- 我们把推后执行的任务叫做工作（work），描述它的数据结构为work_struct。
- 这些工作以队列结构组织成工作队列（workqueue），其数据结构为workqueue_struct。
- 工作者线程就是负责执行工作队列中的工作。系统默认的工作者线程为events，自己也可以创建自己的工作者线程。

工作队列的实现

- 工作队列子系统是一个用于创建内核线程（工作者线程）的接口，通过它创建的进程负责执行由内核其他部分排队到队列里的任务
- 工作队列子系统提供一个默认的工作者线程（events/n）来处理需推后的工作
  - 其中n是处理器的编号，每个处理器对应一个线程
  - 工作队列最基本的形式
    - 将需要推后执行的任务交给特定的通用线程
- 工作队列可以让驱动程序创建一个专门的工作者线程来处理需要推后的工作
  - 主要适用于处理器密集型和性能要求严格的任务

工作队列的核心组成

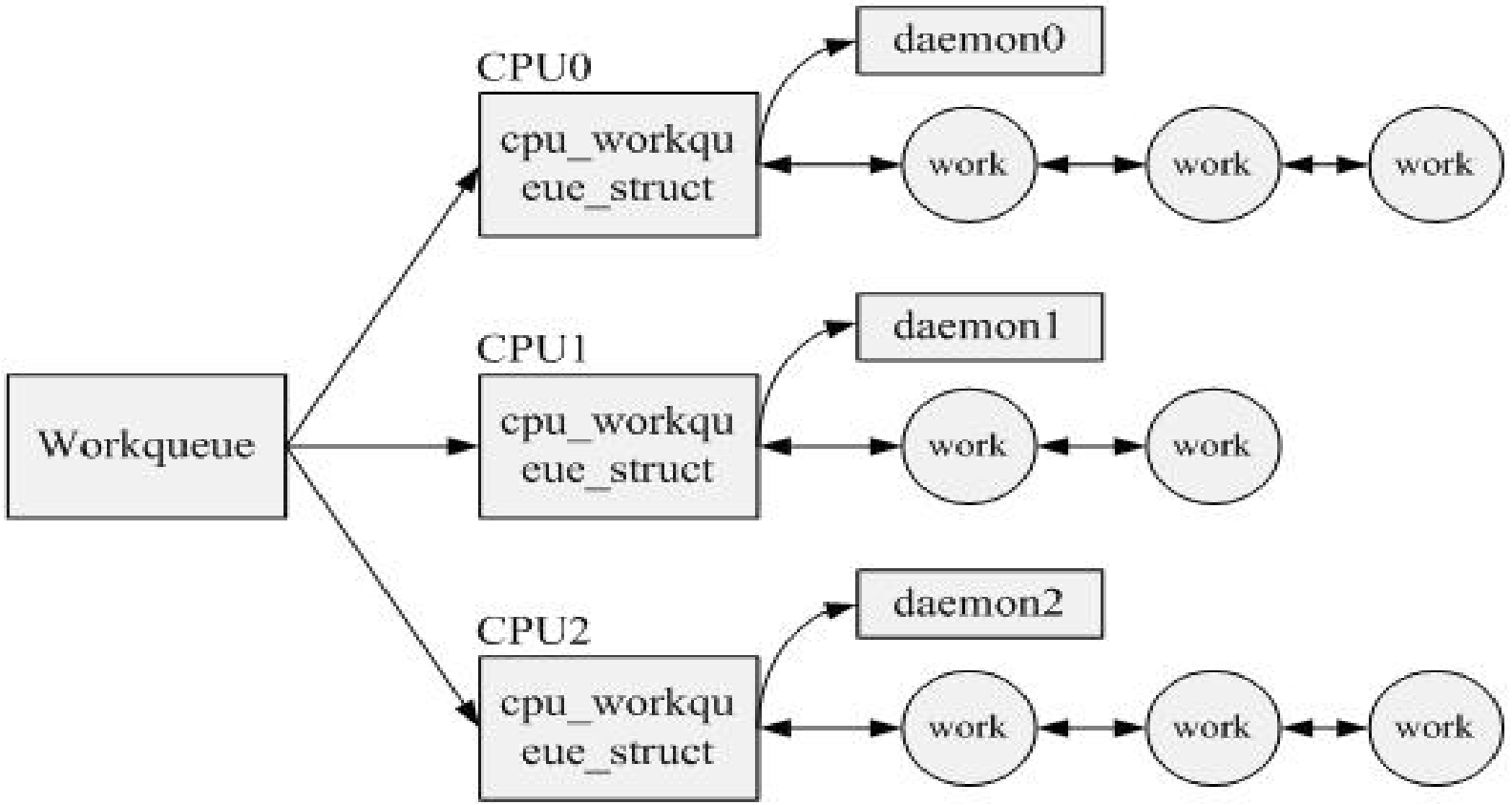
- 工作者线程
  - 工作队列：workqueue_struct
  - CPU相关工作队列：cpu_workqueue_struct
  - 任务：work_struct



工作队列的特点

- 工作队列在进程上下文中执行；
- 可以阻塞；
- 可以被重新调度；
- 使用工作队列的两种形式：
  - 缺省工作者线程
  - 自建的工作者线程
- 在工作队列和内核其他部分之间使用锁机制就像在其他的进程上下文一样；
- 默认允许响应中断；
- 默认不持有任何锁。

workqueue


Workqueue
CPU0
cpu_workqu
eue_struct
daemon0
work
work
work
CPU1
cpu_workqu
eue_struct
daemon1
work
work
CPU2
cpu_workqu
eue_struct
daemon2
work
work
work

workqueue结构体

工作队列work_struct结构体，位于/include/linux/workqueue.h

- typedef void (*work_func_t)　(struct work_struct *work);

- struct work_struct {
  atomic_long_t data; /*传递给处理函数的参数*/
  #define WORK_STRUCT_PENDING 0/*工作是否正在等待处理标志*/
  #define WORK_STRUCT_FLAG_MASK (3UL)
  #define WORK_STRUCT_WQ_DATA_MASK (~WORK_STRUCT_FLAG_MASK)
  struct list_head entry; /* 连接所有工作的链表*/
  work_func_t func; /* 要执行的函数*/
  #ifdef CONFIG_LOCKDEP
  struct lockdep_map lockdep_map;
  #endif
  };

- 这些工作结构被连接成链表，组成工作队列。当一个工作者线程被唤醒时，它会执行它的链表上的所有工作。工作被执行完毕，它就将相应的work_struct对象从链表上移去。当链表上不再有对象的时候，它就会继续休眠。可以通过DECLARE_WORK在编译时静态地创建该结构，以完成推后的工作。

work调度

- #include <linux/workqueue.h>
- 在需要调度工作队列的时候，引用类似tasklet_schedule()函数如：
- int schedule_work(struct work_struct *work)； //put work task in global workqueue
- 如果有时候并不希望工作马上就被执行，而是希望它经过一段延迟以后再执行。在这种情况下，可以调度指定的时间后执行函数：
- int schedule_delayed_work(struct delayed_work *work, unsigned long delay);

- 其中是以delayed_work为结构体的指针，而这个结构体的定义是在work_struct结构体的基础上增加了一项timer_list结构体。
- struct delayed_work {
  struct work_struct work;
  struct timer_list timer; /* 延迟的工作队列所用到的定时器，当不需要延迟时初始化为NULL*/
  };
- 这样，便使预设的工作队列直到delay指定的时钟节拍用完以后才会执行。

- 返回值：
- Returns zero if @work was already on the kernel-global workqueue and * non-zero otherwise.

work程序设计

- #include <linux/workqueue.h>
- 初始化一个work
- INIT_WORK(struct work_struct *work, work_func_t func);
- DECLARE_WORK (struct work_struct *work, work_func_t func);
- DECLARE_DELAYED_WORK (struct work_struct *work, work_func_t func);

work程序设计

- 使用缺省工作队列处理中断下半部的设备驱动程序模板如下：

```
/*中断处理下半部*/
void my_do_work(unsigned long)
{
    ……/*编写自己的处理事件内容*/
}

/*定义工作队列和下半部函数并关联*/
static DECLARE_WORK(my_work, my_do_work);

/*中断处理上半部*/
irpreturn_t my_interrupt(unsigned int irq,void *dev_id)
{
  ……
  schedule_work(&my_work)/*调度my_work 处理函数
  ……
}
```

work程序设计

- /*设备驱动的加载函数*/
- int __init xxx_init(void)

```
{
    ……
    /*申请中断,转去执行my_interrupt函数并传入参数*/
   result=request_irq(my_irq,my_interrupt,IRQF_DISABLED,"xxx",NULL);
    ……
}
```

- /*设备驱动模块的卸载函数*/
- void __exit xxx_exit(void)

```
{
    ……
    /*释放中断*/
    free_irq(my_irq,my_interrupt);
    ……
}
```

粤嵌教育

Thank You !

www.gec-edu.org