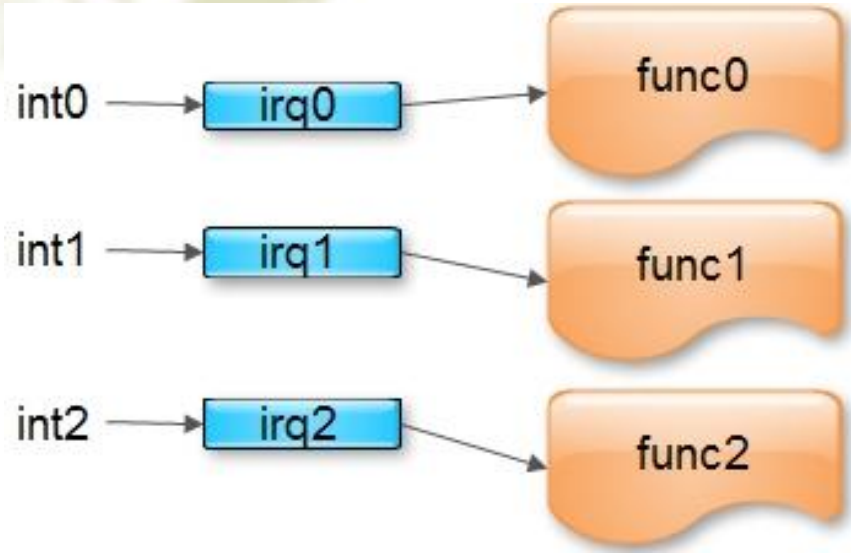
5.3 中断下半部处理机制

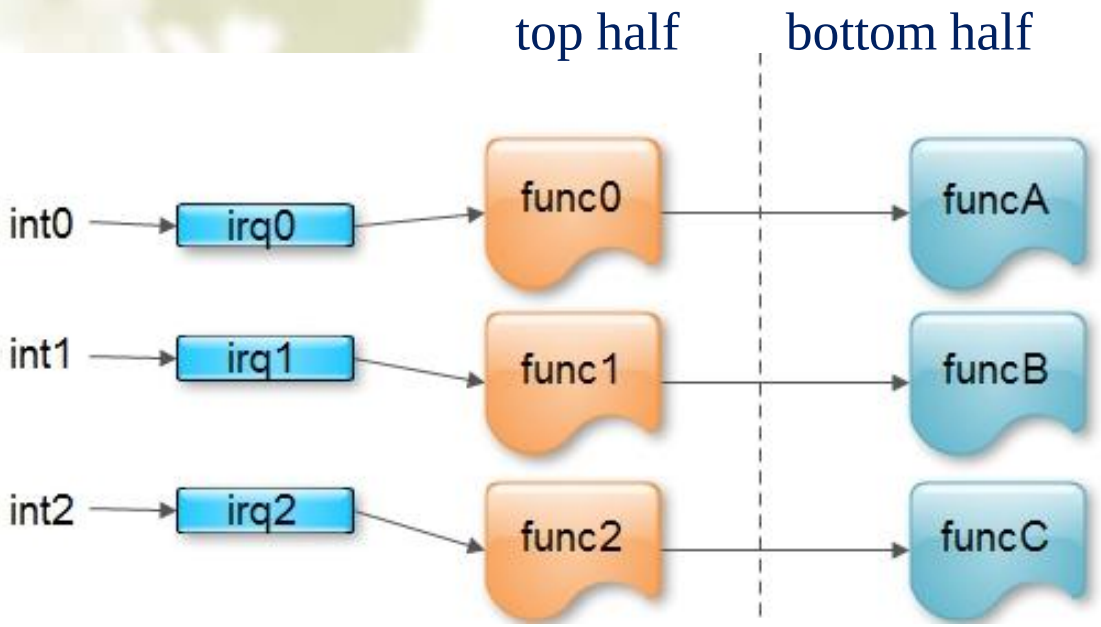
西安邮电大学

中断的基本机制



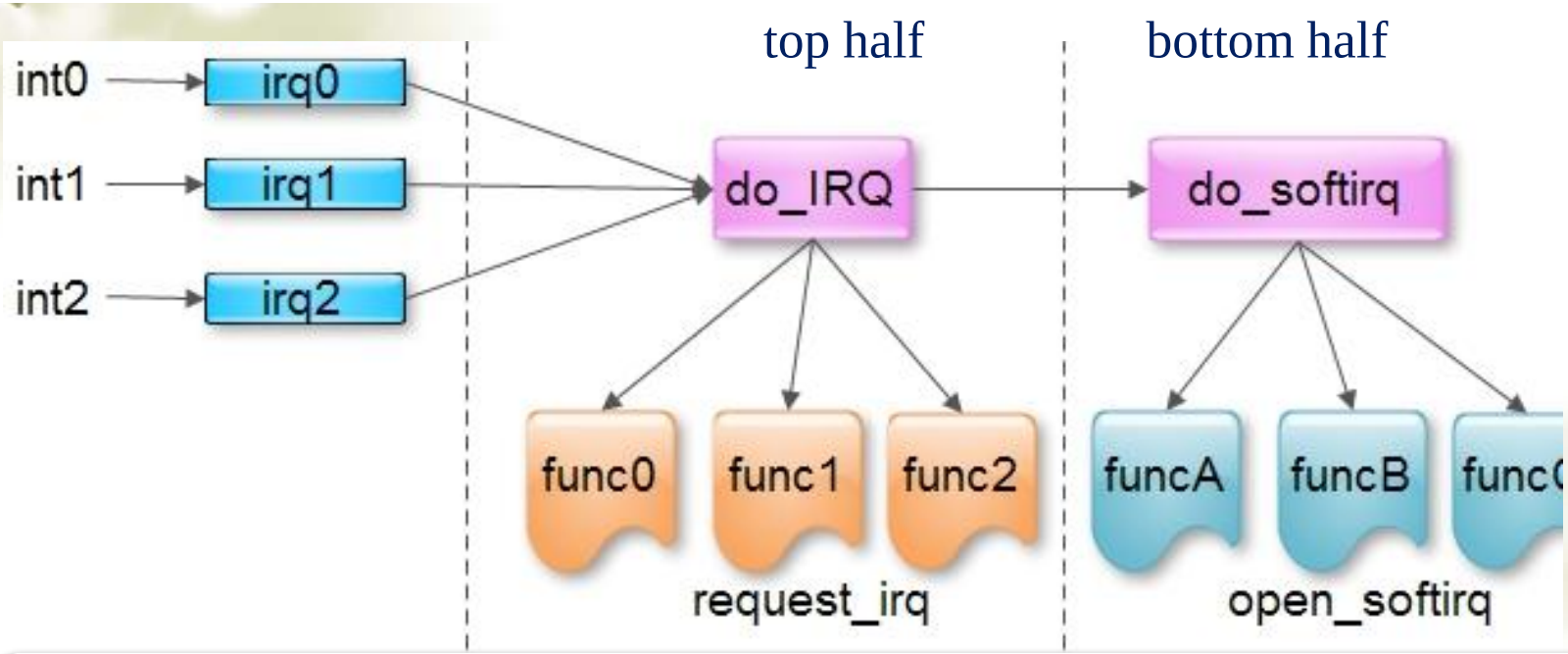
中断服务程序一般都是在中断请求关闭的条件下执行的，以避免嵌套而使中断控制复杂化。但是，中断是一个随机事件，它随时会到来，如果关中断的时间太长，CPU就不能及时响应其他的中断请求，从而造成中断的丢失。因此，内核的目标就是尽可能快的处理完中断请求，尽其所能把更多的处理向后推迟。如图是中断的基本模型，在中断向量表中填入中断处理程序的入口地址，然后跳到该程序执行。

中断的下半部



随着系统的不断复杂，中断处理函数要做的事情也越来越多，多到都来不及接收新的中断了。于是发生了中断丢失，这显然不行，于是，内核把中断处理分为两部分：上半部（top half）不可中断和下半部（bottom half）可中断，上半部（也就是中断服务程序）内核立即执行，而下半部（就是一些内核函数）留着稍后处理

软中断机制



不管是中断的上半部，还是下半部，都是一种概念，实际上它们都是内核中的一个函数，这些函数写好后，什么时候执行，如何执行，这是内核必须统一管理的。在上一讲的中断机制中，我们介绍了中断注册函数request_irq()把中断服务例程添加到中断请求队列中。

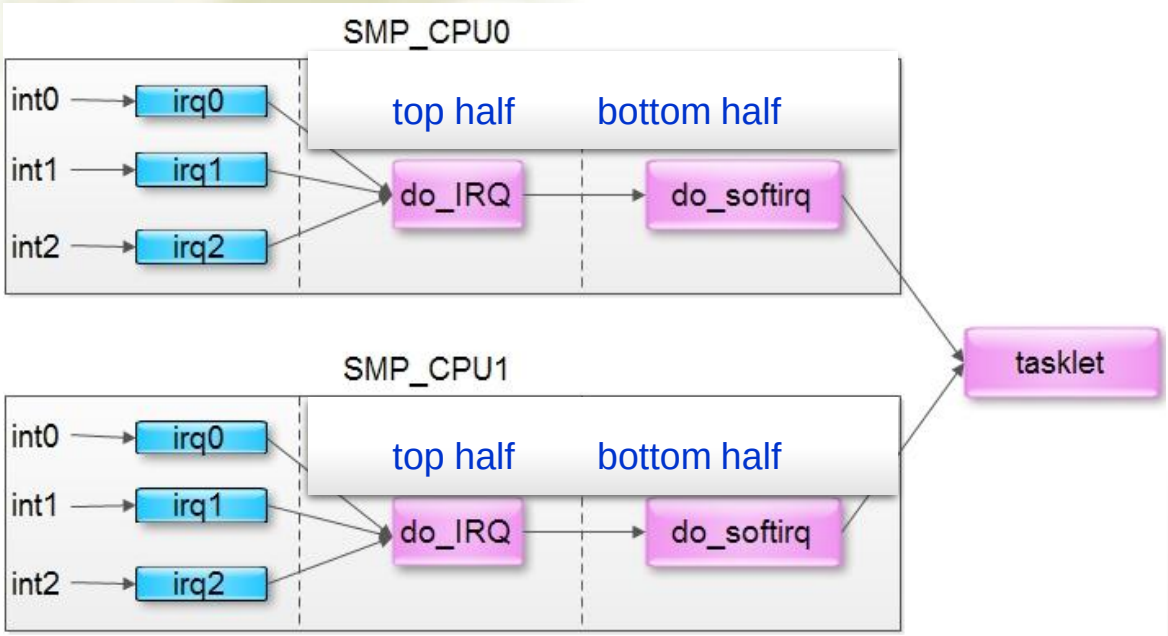
其执行是由do_IRQ完成的。与之对应，通过open_softirq添加下半部对应的处理函数。而对其执行则是通过do_softirq，也就是软中断机制完成的。

软中断类型

```
enum
{
    HI_SOFTIRQ=0,                   /* 高优先级tasklet */  /* 优先级最高 */
    TIMER_SOFTIRQ,                  /* 时钟相关的软中断 */
    NET_TX_SOFTIRQ,                 /* 将数据包传送到网卡 */
    NET_RX_SOFTIRQ,                 /* 从网卡接收数据包 */
    BLOCK_SOFTIRQ,                  /* 块设备的软中断 */
    BLOCK_IOPOLL_SOFTIRQ,           /* 支持IO轮询的块设备软中断 */
    TASKLET_SOFTIRQ,                /* 常规tasklet */
    SCHED_SOFTIRQ,                  /* 调度程序软中断 */
    HRTIMER_SOFTIRQ,                /* 高精度计时器软中断 */
    RCU_SOFTIRQ,                    /* RCU锁软中断，该软中断总是最后一个软中断 */
    NR_SOFTIRQS                     /* 软中断数，为10 */
};
```

下半部的处理方式主要有soft_irq,tasklet,workqueue三种，他们在使用方式和适用情况上各有不同。soft_irq用在对下半部执行时间要求比较紧急的场合，在中断上下文执行。tasklet和work queue在普通的驱动程序中用的相对较多，主要区别是tasklet是在中断上下文执行,而workqueue是在进程上下文，因此可以执行可能睡眠的操作。每个软中断在内核中以softirq_action表示，内核目前实现了10中软中断，定义在linux/interrupt.h中，如图所示。

小任务（tasklet）机制



小任务（tasklet）机制是I/O驱动程序中实现可延迟函数的首选方法；

小任务和工作队列是延期执行工作的机制，其实现基于软中断，但他们更易于使用，因而更适合于设备驱动程。，所谓小任务，就是执行一些迷你任务。

小任务数据结构

```
struct tasklet_struct {
    struct tasklet_struct *next; /*指向链表中的下一个结构*/
        unsigned long state;        /* 小任务的状态 */
        atomic_t count;             /* 引用计数器 */
        void (*func) (unsigned long); /* 要调用的函数 */
        unsigned long data;           /* 传递给函数的参数
*/
};
```

结构中的func域就是下半部中要推迟执行的函数，data是它唯一的参数。

Count域是小任务的引用计数器。如果它不为0，则小任务被禁止，不允许执行；只有当它为零，小任务才被激活，并且在被设置为挂起时，小任务才能够执行。

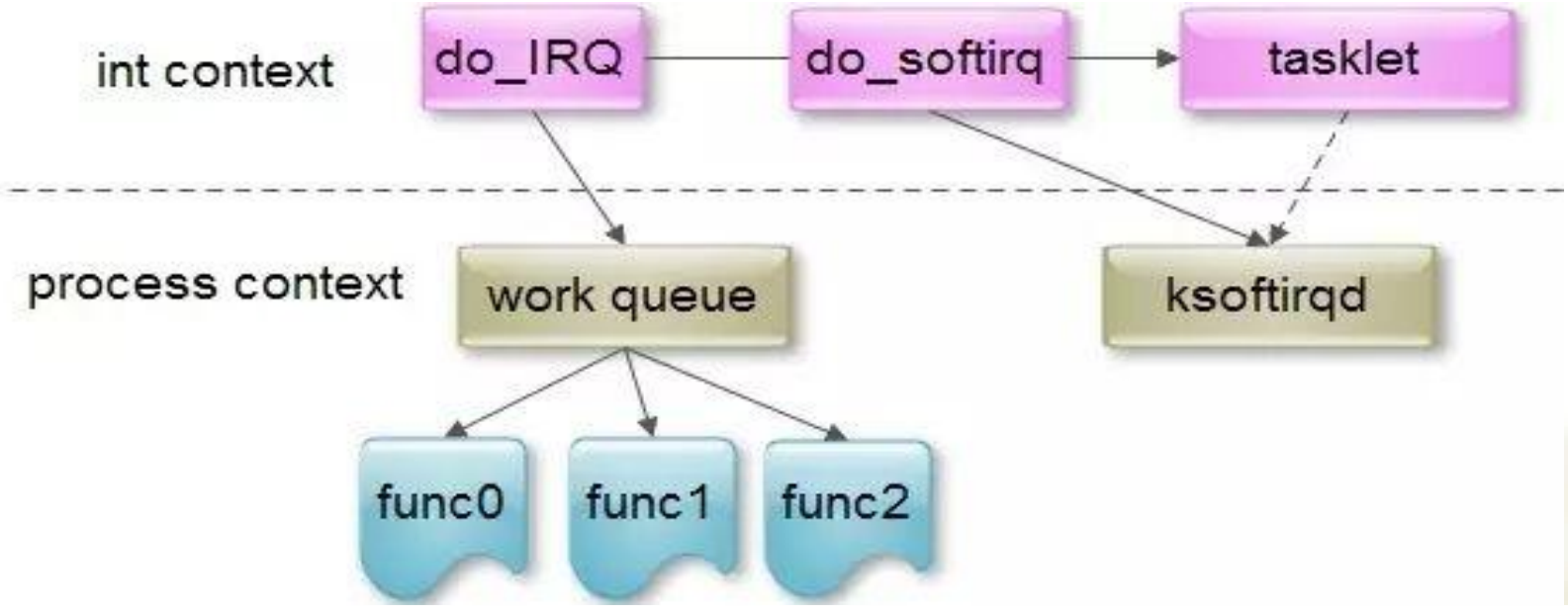
State域的取值为TASKLET_STATE_SCHED或TASKLET_STATE_RUN。TASKLET_STATE_SCHED表示小任务已被调度，正准备投入运行，TASKLET_STATE_RUN表示小任务正在运行。TASKLET_STATE_RUN只有在多处理器系统上才使用，任何时候单处理器系统都清楚一个小任务是不是正在运行。

编写自己的小任务并调度

声明和使用小任务	• **DECLARE_TASKLET(my_tasklet, my_tasklet_handler, dev);**
编写自己的小任务处理程序	• **void tasklet_handler(unsigned long data)** • 小任务不能睡眠，不能在小任务中使用信号量或者其它产生阻塞的函数。但它运行时可以响应中断
调度或杀死小任务	• tasklet_schedule(&my_tasklet); /*调度*/ • tasklet_kill() /*杀死*/

在本章最后一讲中，我们将演示如何编写小任务，并分析相关源代码。

工作队列（workqueue）机制



前面的机制不论如何折腾，有一点是不会变的。它们都在中断上下文中。为什么？说明它们不可挂起。而且由于是串行执行，因此只要有一个处理时间较长，则会导致其他中断响应的延迟。为了完成这些不可能完成的任务，于是出现了工作队列。工作队列说白了就是一组内核线程，作为中断守护线程来使用。多个中断可以放在一个线程中，也可以每个中断分配一个线程。

工作队列对线程作了封装，使用起来更方便。

因为工作队列是线程，所以我们可以使用所有可以在线程中使用的方法。

工作队列数据结构

```
struct  work_struct{
    unsigned long pending; /* 这个工作正在等待处理吗？ */
    struct list_head entry;   /* 工作的链表 */
    void (*func) (void *);   /* 要执行的函数 */
    void *data;                  /* 传递给函数的参数 */
    void *wq_data;              /* 内部使用 */
    struct timer_list timer;    /* 延迟的工作队列所用到的定时器*/
};
```

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，其中，func钩子函数就是推后执行的函数，data就是传递给这个函数的参数。

工作队列类型的数据结构

```
struct workqueue_struct {
    struct cpu_workqueue_struct *cpu_wq;  /*工作者线程数组*/
    struct list_head list; /*连接工作队列类型的链表*/
    const char *name;       /*工作者线程的名称*/
    int singlethread;       /*是否创建新的工作者线程，0表示采用
默认的工作者线程event/n*/
    int freezeable; /* Freeze threads during suspend */
    int rt;
#ifdef CONFIG_LOCKDEP
    struct lockdep_map lockdep_map;
#endif
};
```

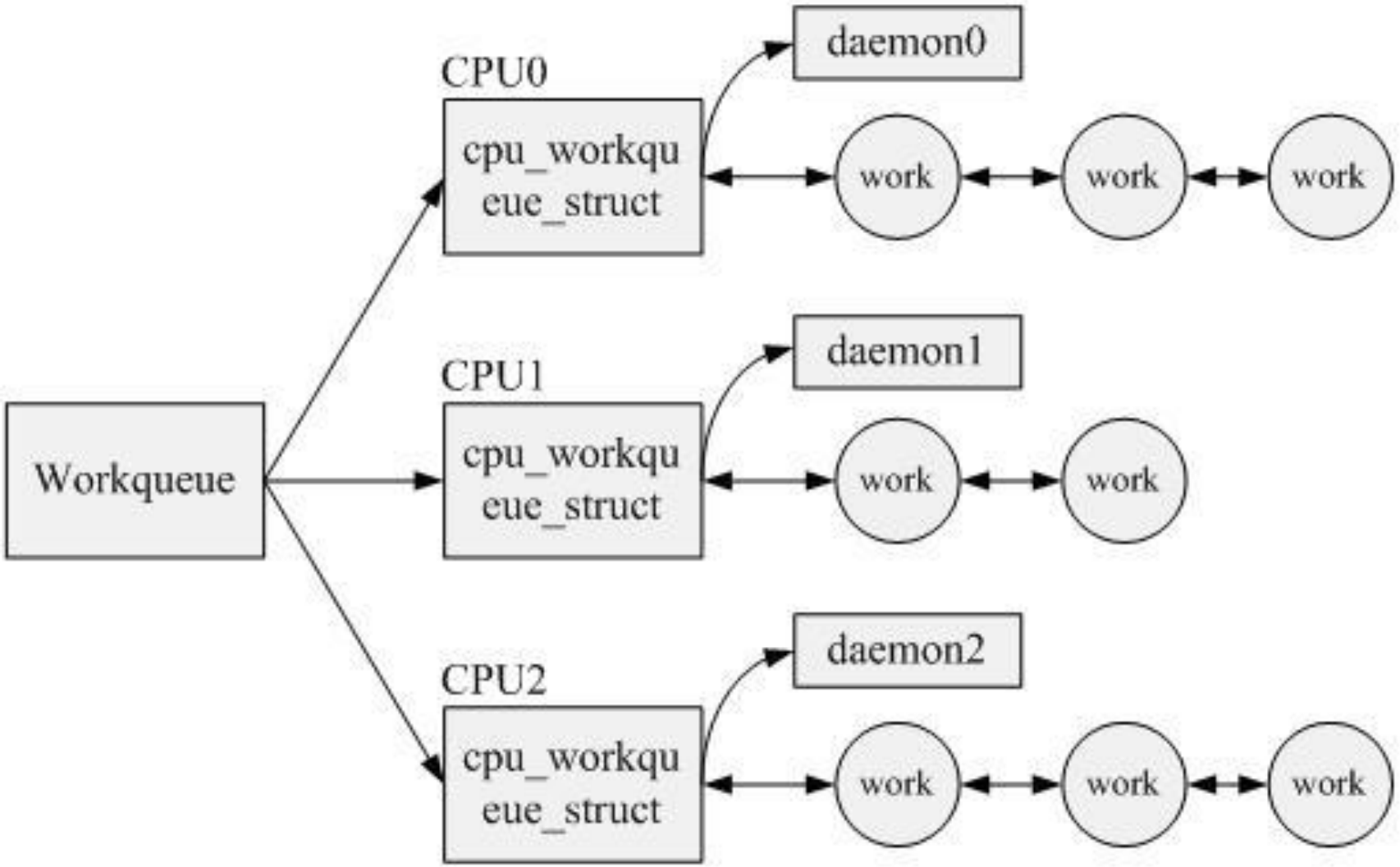
这些工作以队列结构组织成工作队列（workqueue），其数据结构为workqueue_struct，其中第一个字段是工作者线程数组，每个CPU对应一个。第二个字段是工作者队列形成双向链表。

每CPU工作队列结构

```
struct cpu_workqueue_struct {
 spinlock_t lock;/*因为工作者线程需要频繁的处理连接到其上的工作，所以需要枷锁保护*/
 struct list_head worklist;
 wait_queue_head_t more_work;
 struct work_struct *current_work; /*当前的work*/
 struct workqueue_struct *wq;  /*所属的workqueue*/
 struct task_struct *thread; /*任务的上下文*/
} ____cacheline_aligned;
```

如果是多线程，内核根据当前系统CPU的个数，为每CPU创建cpu_workqueue_struct（简称CWS）结构体。
在该结构主要维护了一个任务队列，以及内核线程需要睡眠的等待队列，另外还维护了一个任务上下文，即task_struct。

工作队列运行机制


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

工作队列运行机制

当用户调用工作队列(workqueue)初始化接口函数对工作队列进行初始化时，内核就开始为用户分配一个工作队列对象，并且将其链到一个全局的workqueue队列中。然后内核根据当前CPU的情况，为workqueue对象分配与CPU个数相同的cpu_workqueue_struct对象，每个cpu_workqueue_struct对象都会有一条任务队列。紧接着，内核为每个cpu_workqueue_struct对象分配一个内核线程，即内核daemon去处理每个队列中的任务。至此，用户调用初始化接口将工作队列初始化完毕，返回workqueue的指针。

Workqueue初始化完毕之后，将任务运行的上下文环境构建起来了，但是具体还没有可执行的任务，所以，需要定义具体的work_struct对象。然后将work_struct加入到任务队列中，内核会唤醒内核线程daemon去处理任务。

小结：何时使用哪种中断处理机制

Request_irq挂的中断函数要尽量简单，只做必须在屏蔽中断情况下要做的事情。

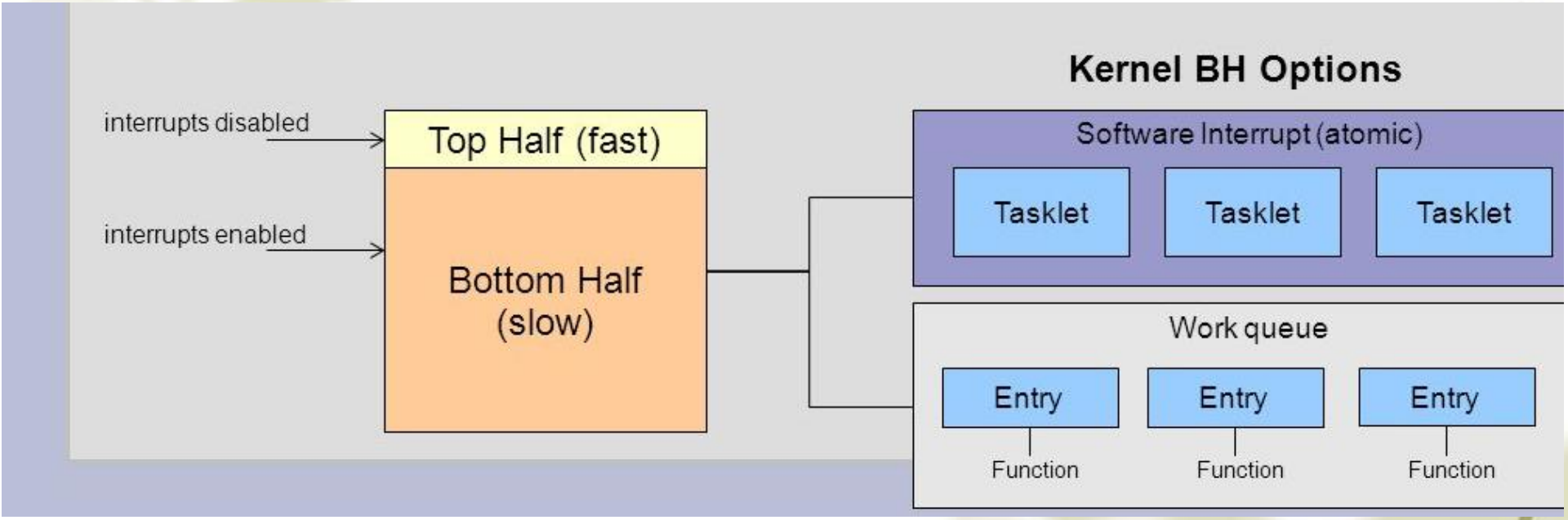
中断的其他部分都在下半部中完成。

软中断的使用原则很简单，最好不用。它甚至都不算是一种真正的中断处理机制，只是tasklet的实现基础。

工作队列也要少用，如果不是必须要用到线程才能用的某些机制，就不要使用工作队列，为什么对工作队列机制详细介绍，是希望大家从这种设计机制中得到启发。其实对于中断来说，只是对中断进行简单的处理，大部分工作是在驱动程序中完成的。除了上述情况，一般都使用小任务tasklet。

即使是下半部，也只是作必须在中断中要做的事情，如保存数据等，其他都交给驱动程序去做。

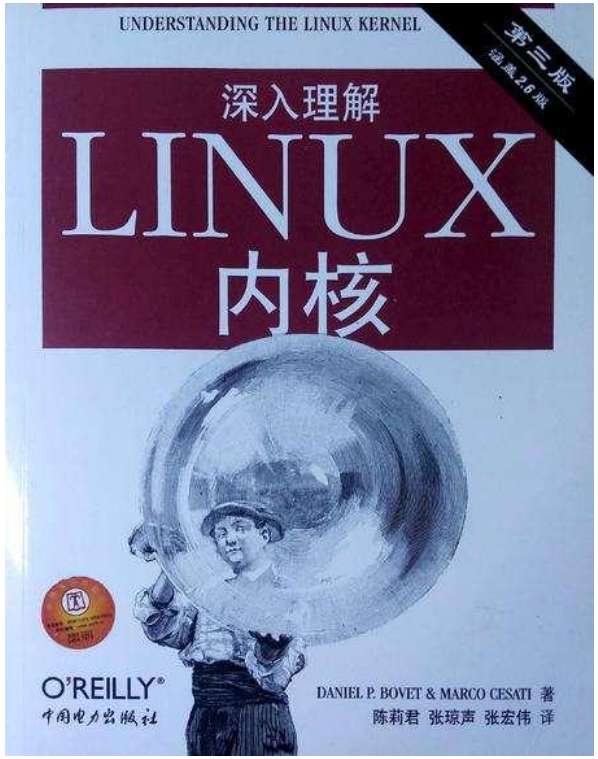
小结：何时使用哪种中断处理机制



动手实践

Linux内核之旅网站
http://www.kerneltravel.net/
新手上路栏目的内核入门系列中“感受小任务机制：中断下半部分之tasklet
感受工作队列：中断下部分之工作队列！”调试其中的程序。

参考资料



深入理解Linux内核 第三版第四章

Linux内核设计与实现 第三版第七章

http://www.wowotech.net/ ，蜗窝科技网站关于中断的系列文章

带着思考离开

1. 为什么要有中断下半部分处理机制？而且有好几种机制？
2. 中断下半部处理机制中，你认为是否还有改进的余地？

谢谢大家！

THANK YOU