**中断大致轮廓**

中断就是将电信号传给芯片的中断控制器，如果中断线没禁用，则中断控制器将中断发往处理器，则处理器会立即停止它正在做的事，保存被中断任务的各个寄存器值，然后将当前PC指针指向内存中预定义的位置开始执行相应中断线所对应的中断处理程序。当中断处理程序执行完毕，退出中断，恢复中断前的进程，如果是返回用户空间，也就是中断了用户空间进程，则会检测nedd\_resched这个重新调度标志，若设置，则调度其他进程。如果中断的是内核线程，则当系统判断此时调度安全是，则调度其他进程，也就是说被中断的进程恢复前内核会先判断是否可以被抢占，如果可以可能会调度别的进程。

当执行中断处理程序时，此时内核处于中断上下文，也叫原子上下文，此处不能睡眠，不能调用导致睡眠的接口，所以尽快执行完然后恢复中断前的工作非常重要，内核采用中断上下部来解决此问题，中断发生时，快速响应中断上半部，在上半部中部署下半部，上半部分执行完，中断的过程就结束了，中断下半部分会在适当的时候得到执行。

中断处理程序打断的代码，可能是其他中断线的中断处理程序，这时候就需要有一个栈保存被中断处理程序的信息，内核为每个处理器提供了一页大小的内核栈，叫中断栈。

芯片对外有很多中断请求线，每个中断请求线对关联一个数值量。

**注册中断处理程序**

内核注册并激活某个中断线的中断处理程序需用到如下接口：

**int request\_irq(unsigned int irq,irq\_handler\_t handler,unsigned long flags,const char \*name,void \*dev)**

irq：表示中断线的数字标号，一般硬件上预先确定

handler:指向中断处理程序

flags:此中断线属性设置

IRQF\_DISABLED:处理此中断程序期间，禁止本CPU所有其他中断，一般不设置

IRQF\_SHARED:中断线可被共享，也就是还可以用此中断线注册中断处理函数，中断触发后，本中断线上注册的中断处理程序都会运行一遍，如果有一个返回IRQ\_HANDLED则表示中断完成处理

name:给此次注册的中断处理程序的设备一个name

dev:这个当中断线属性为IRQF\_SHARED时，必须制定一个值，以便中断处理程序释放时，从中断线的所有中断处理程序中选出正确的那一个，如果中断线不共享，则可以设为NULL;

request\_irq执行成功返回0，否则为非0，request\_irq由于会创建在/proc/irq下创建节点，会调用到kmalloc,所以会导致睡眠发生。

request\_irq会创建一些结点，如**/proc/interrupts**

cat **/proc/interrupts**

cpu0

0： 33333 XT-PIC timer

4： 30 XT-PIC uhci-hcd,eth0

第一列：中断号，没有注册中断处理程序的中断号不显示

第二列：中断发生的计数，系统中每个cpu都会有这一列

第三列：处理这个中断的中断处理器

第四列：中断注册时的name，可以叫与这个中断的设备名字

**释放中断处理程序**

内核使用如下接口注销中断处理程序，并释放中断线

void free\_irq(unsigned int irq,void \*dev);

如果指定的中断线不是共享，则删除中断程序并禁用这条中断线，如果指定中断线是共享，则仅仅删除dev对应的处理程序

**中断处理程序**

中断处理程序声明如下：

static irqreturn\_t intr\_handler(int irq,void \*dev);

irq为中断线号

dev为注册中断处理程序时传入的dev

返回值irqreturn\_t可能是:IRQ\_NONE IRQ\_HANDLED

IRQ\_NONE:表明此中断处理程序处理失败

IRQ\_HANDLED：表明中断处理程序已经正确处理中断请求

中断处理程序是无需考虑重入的，因为当一个给定的中断线触发中断时，系统会将这个中断线在所有处理器上屏蔽，但其他中断线还是可能被触发的。

**中断线共享**

共享中断线需满足如下两点

1：当对一个中断线注册中断处理程序request\_irq参数flags都设置为IRQF\_SHARED标志时，标明中断线是共享，如果某个对此中断线注册处理程序未设置IRQF\_SHARED，则中断线不可共享

2：request\_irq传入的dev参数不能为空

3：中断发生时，硬件需有类似寄存器标明是哪个硬件中断发生，当依次调用注册的中断程序时可以通过判断寄存器值来确定是哪个中断处理程序应该处理这个事件，否则不知道调用哪一个中断处理程序

**激活或禁用中断线接口**

系统提供禁用或激活**当前处理器所有中断线**或者**所有处理器的某条中断线**的接口

禁用或激活当前处理器：

//未保存中断系统状态

local\_irq\_disable();

local\_irq\_enable();

//保存中断系统状态，注意以下两个接口必须在同一个函数中进行

unsigned long flags;

local\_irq\_save(flags);

local\_irq\_restore(flags);

禁用所有处理器上某条中断线：

void disable\_irq(unsigned int irq);//禁止中断线irq，中断线处理程序如正在执行，则待其完毕，才返回

void disable\_irq\_nosync(unsigned int irq);//禁止中断线irq，中断线处理程序如正在执行，也立即返回

void enable\_irq(unsigned int irq);

这三个函数可以嵌套调用，但是必须记住，禁用几次则相应激活几次，这三个函数不会睡眠

**中断下半部的实现**

下半部就是指将工作推后执行的内核机制，就是将一段程序在后面某个时候想执行时让它执行。

早期的linux下半部只有BH机制，是一个静态创建的，由32个bottom halves组成的链表，上半部可以通过一个32位的标识来确定那个bottom halves可以执行。

现在的linux将工作推迟执行的机制有：内核定时器，软中断，tasklet,工作队列

其中内核定时器是指定推迟确定的时间执行，其他的则是推迟到内核空闲时执行

**软中断**

软中断是在编译期间静态分配的，一个软中断由softirq\_action结构表示

struct softirq\_action{

void (\*action)(struct softirq\_action \*);

};

其中的action表示软中断处理程序

void softirq\_handler(struct softirq\_action \*);

系统最多支持32个软中断，放于如下数组中

static struct softirq\_action softirq\_vec[NR\_SOFTIRQS];

软中断有优先级，注册软中断时可以指定优先级，用索引号表示优先级，索引号小的软中断先执行，内核调用如下接口注册软中断处理函数

open\_softirq(NET\_TX\_SOFTIRQ,net\_tx\_action);//NET\_TX\_SOFTIRQ为索引号，net\_tx\_action为软中断处理程序

注册后用raise\_softirq(NET\_TX\_SOFTIRQ)将软中断设置为挂起状态，也就是可执行状态，当内核下次调用do\_softirq()时（一般从一个硬件中断返回时，或在载ksoftirq内核线程中都会调用do\_softirq()），会遍历32个软中断，将其中的挂起软中断依次执行

备注：多处理器系统系统中，每个处理器都会有一个名为ksoftirq/n的内核线程，n表示处理器编号

内核执行完中断处理函数后，会马上调用do\_softirq(),所以一般在中断处理程序中若使用软中断，则在中断处理程序最后调用raise\_softirq

软中断处理程序执行时允许中断抢占，也**只能被**中断处理程序抢占，中断处理运行时，当前处理器的软中断被禁止，但软中断(甚至是相同类型的软中断)可在其他处理器上执行，**同一软中断可在不同处理器上同时运行**，所以对于其中的共享数据要加锁保护，

**tasklet**

**通常使用的下半部机制就是tasklet**,tasklet的实现机制是软中断，但是其接口更简单，锁保护要求更低

tasklet在内核用如下结构体表示：

struct tasklet\_struct{

struct tasklet\_struct \*next;//链表中下一个tasklet

unsigned long state;//tasklet状态

atomic\_t count;//0表示tasklet允许执行，非0表示tasklet被禁止

void (\*func)(unsigned long);//tasklet执行的处理函数

unsigned long data;//给处理函数的参数

};

state可以为0, TASKLET\_STATE\_SCHED(tasklet已经被唤醒就绪，准备投入运行) ,TASKLET\_STATE\_RUN（tasklet正在运行）

count为0表示tasklet允许执行，非0表示tasklet被禁止

只有tasklet设置为**挂起TASKLET\_STATE\_SCHED**和**允许执行**状态，该tasklet才会被执行

tasklet所表示的软中断是索引号为HI\_SOFTIRQ和TASKLET\_SOFTIRQ的两个软中断中的一个，如果tasklet由HI\_SOFTIRQ索引号的软中断实现则其先于由TASKLET\_SOFTIRQ索引号的软中断实现的tasklet执行

索引号为HI\_SOFTIRQ和TASKLET\_SOFTIRQ的两个软中断对应的软中断处理函数分别为tasklet\_hi\_action()和tasklet\_action(),这两个处理函数分别处理两个链表tasklet\_vec和tasklet\_hig\_vec，这两个链表挂的是注册的tasklet，内核循环获取这些tasklet，会检查tasklet中的TASKLET\_STATE\_RUN标志判断这个tasklet是否在别的处理器上运行，如果是则跳到下一个tasklet,若不是则将tasklet状态设置为TASKLET\_STATE\_RUN,在检查tasklet中的count值是否为0，0表示这个tasklet没被禁止，非0表示禁止，当确定这个tasklet在别的处理器上未执行会且不禁止情况下，就调用tasklet中的func处理函数，重复上述步骤依次检查这两个链表上的tasklet

备注：由上可知，同一个tasklet在多处理器系统中也绝不会同时执行，tasklet可被中断抢占

当注册了一个tasklet，怎样让内核调度它了，要想调度它，**首先**得让加入链表tasklet\_vec或tasklet\_hig\_vec，**然后**让对应的软中断处于唤醒状，内核使用tasklet\_schedule()和tasklet\_hi\_schedule()来调度tasklet,其实现细节如下：将tasklet加入到tasklet\_vec或tasklet\_hig\_vec中，将对应的软中断唤醒（32位标志位对应位置1），这样，下次内核执行do\_softirq时就会调度者两个软中断然后执行其软中断处理函数，软中断处理函数则依次处理链表上的tasklet

tasklet使用过程：

1：**创建tasklet**

**动态创建**一个tasklet

tasklet\_init(t,tasklet\_handler,dev);

**静态创建**一个tasklet

**DECLARE\_TASKLET(name,func,data);**//tasklet的count初始为0，允许状态

等价于

struct tasklet\_struct name = {

NULL,

0,

ATOMIC\_INIT(0),

func,

data

};

**DECLARE\_TASKLET\_DISABLED(name,func,data);**//tasklet的count初始为1，禁止状态

等价于

struct tasklet\_struct name = {

NULL,

0,

ATOMIC\_INIT(1),

func,

data

};

tasklet处理程序声明：

void tasklet\_handler(unsigned long data);

2调度tasklet

tasklet\_schedule(&my\_tasklet);

或者

tasklet\_hi\_schedule(&my\_tasklet);

备注：由这两个接口的实现可知，当tasklet执行这个接口但还没有运行前，再度调度这个tasklet，则其只会运行一次，若已经运行且清除了TASKLET\_STATE\_RUN状态，则再度调度会再次运行。**tasklet处理函数总是在调度它的处理器上执行**

**禁止本地处理器的软中断(tasklet也就被禁止了)**

void local\_bh\_disable();

void local\_bh\_enable();

**工作队列**

内核中的**工作队列子系统**是一个用于创建内核线程的系统，它创建的内核线程称为工作者线程，工作队列子系统提供一个缺省的工作者线程event/n(n为处理器编号)，可以将推后的工作交给这个线程处理，**怎么将推后的工作交给这个内核线程，后续会提供一些接口**，当然除了交给缺省的工作者线程，也可以利用工作队列子系统自己创建一个工作者线程，由于工作队列是将工作交给一个内核线程去执行，所以**工作队列可以睡眠**

**工作者线程**:

内核中的工作队列子系统中表示工作者线程的数据结构如下：

struct workqueue\_struct{

struct cpu\_workqueue\_struct cpu\_wq[NR\_CPUS];

struct list\_head list;

const char \*name;

int sinqlethread;

int freezeable;

int rt;

};

其中的struct cpu\_workqueue\_struct cpu\_wq[NR\_CPUS];有几个处理器则数组中就有几项.也就是如果自己创建一个工作者线程,则会在每个处理器中都会创建这个工作者线程

struct cpu\_workqueue\_struct {

spinloc\_t lock;

struct list\_head worklist;//工作列表

wait\_queue\_head\_t more\_work;

struct work\_struct \*current\_struct;

struct workqueue\_struct \*wq;

task\_t \*thread;

};

**挂载工作者线程上的工作**

工作队列子系统中表示工作的数据结构如下:

struct work\_struct{

atomic\_long\_t data;

struct list\_head entry;

work\_func\_t func;

};

工作者线程就是一个内核线程,它们创建完毕后,会执行一个worker\_thread函数,这个函数执行一个死循环然后开始休眠,当有工作work\_struct插入到工作者线程中的处理器struct cpu\_workqueue\_struct cpu\_wq的worklist链表的时候,线程被唤醒,依次执行worklist上的所有工作,工作完毕后将工作work\_struct从链表上移除,然后继续休眠.

工作队列使用:

1:创建工作

静态创建:DECLARE\_WORK(name,void (\*func)(void \*), void \*data);

动态创建:INIT\_WORK(struct work\_struct \*work,void (\*func)(void \*),void \*data);

等价于

struct work\_struct name{

data;

func;

};

其中的处理函数声明如下:

void work\_handler(void \*data);

这个处理程序会被工作者线程执行,运行与进程长下文,允许睡眠,由于是处于内核线程,所以不能访问用户空间.

2:工作的调度(将工作交给工作者线程)

**如果交给缺省的event工作者线程:**

调用**schedule\_work(&work);**一旦其所在的处理器工作者线程唤醒,工作会执行,如果希望经过一段延时再执行,可调用**schedule\_delayed\_work(&work,delay);**

**如果调用自己创将的工作者线程:**

则首先得创建一个自己的工作者线程,用如下接口:

struct workqueue\_struct \*create\_workqueue(const char \*name);//name用于创建的内核线程的命名

如缺省的工作者线程按如下创建:

struct workqueue\_struct \*keventd\_wq;

keventd\_wq = create\_workqueue(“events”);

将工作交于创建的工作者线程用如下接口:

int queue\_work(struct workqueue\_struct \*wq,struct work\_struct \*work);

int queue\_delayed\_work(struct workqueue\_struct \*wq,struct work\_struct \*work,unsigned long delay);

3:为了同步的需求,需要确定工作执行完毕再进一步动作的接口

void flush\_scheduled\_work(void);

flush\_scheduled(void struct workqueue\_struct \*wq);

这些接口会一直睡眠,直到工作者线程event或者wq中所有的工作都被执行才返回

4:取消某个未执行单准备就绪的工作接口

**int cancel\_delayed\_work(struct work\_struct \*work);**

20161107 wangchaoqun