Linux设备驱动进阶

www.gec-edu.org

现实驱动中的问题

- 如何使用内核中的系统内存?
- 如何解决使用频度较高的内存?
- 并发和竞争如何处理?
- 如何处理设备的中断请求?
- 如果实现设备的休眠及唤醒?

使用系统内存

- 系统内存作为重要的资源受系统内存管理的子系统的统一管理
- 由于**MMU**的存在,我们不能直接的使用物理内存
- 外设和驱动程序对内存有不同的要求,如连续性和硬件对地址的要求
- 要避免内存碎片的产生

使用系统内存

Linux的内存区划分

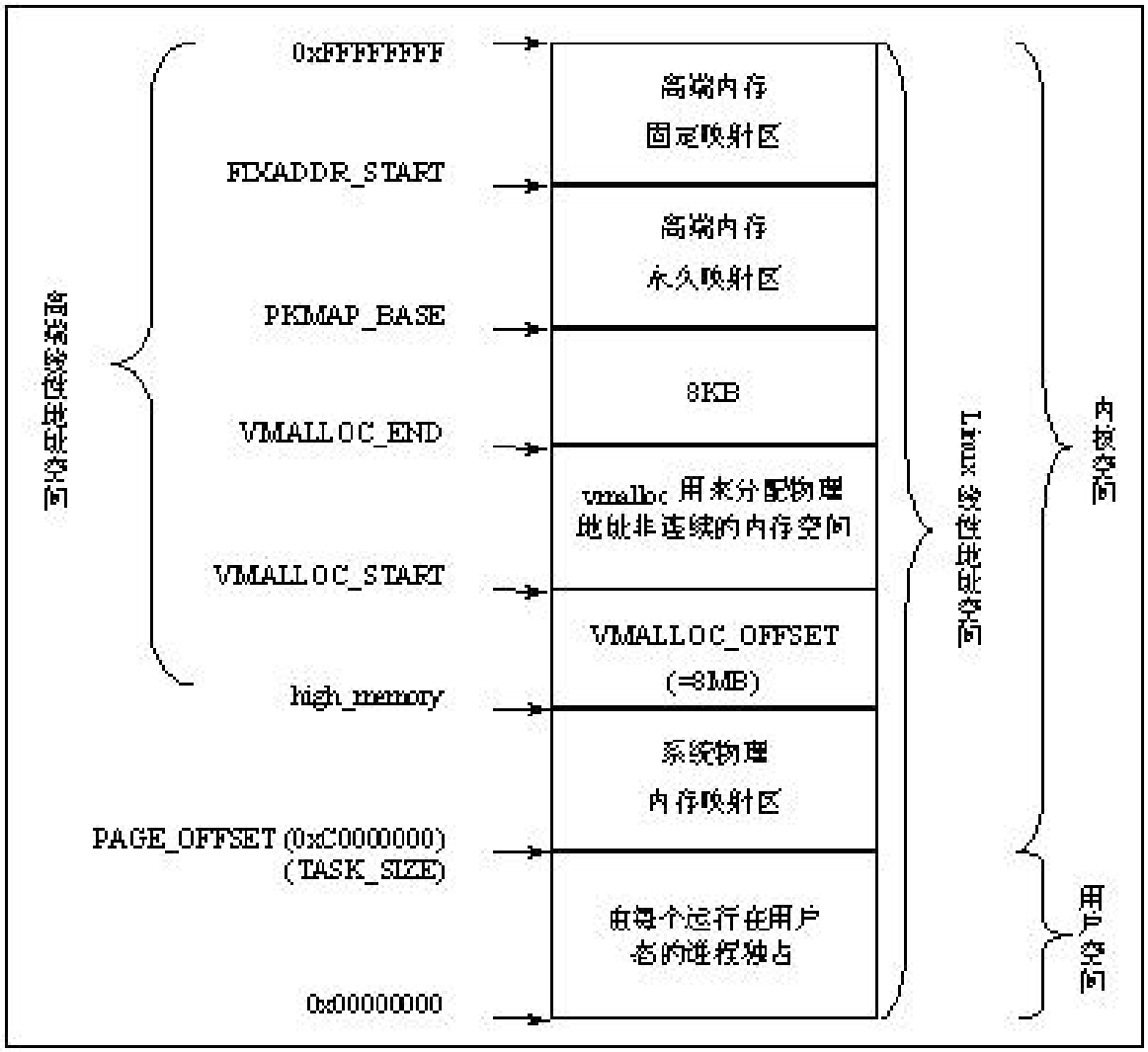
-- linux将物理内存分三个区:DMA,NORMAL,HIGHMEM区,如何分是与平台相关的

-- DMA区,外设可以在这里进行 DMA 存取. 在大部分的健全的平台, 所有的内存都在这个区. 在 x86, DMA 区用在 RAM 的前 16 MB, 这里传统的 ISA 设备可以进行 DMA; PCI 设备没有这个限制

-- NORMAL区,linux内核能访问的地址的物理内存区域, 896M以下

-- HIGHMEM区,linux不能直接映射访问的896M以上的区域

使用系统内存


0xFFFFFFFF
FIXADDR_START
PKMAP_BASE
VMALLOC_END
VMALLOC_START
high_memory
PAGE_OFFSET (0xC0000000)
(TASK_SIZE)
0x00000000
高端内存
固定映射区
高端内存
永久映射区
8KB
vmalloc 用来分配物理
地址非连续的内存空间
VMALLOC_OFFSET
(=8MB)
系统物理
内存映射区
由每个运行在用户
态的进程独占
Linux 系统地址空间
内核空间
用户空间

使用系统内存

- **kmalloc()**

原型

```
#include <linux/slab.h>
void *kmalloc(size_t size, int flags);
```

特性

--与用户程序中malloc有相似之处
--分配的区也是在物理内存中连续
--分配的内存要求不能大于128K
--分配过程中有可能阻塞,这取决于参数
--这个函数快(除非它阻塞)并且不清零它获得的内存
--成功返回一个虚拟地址, 失败返回NULL

- **kfree()**

原型

```
#include <linux/slab.h>
void kfree(void *obj);
```

特性

释放kmalloc分配的内存, obj参数是kmalloc返回的指针

使用系统内存

- **Kmalloc()参数解释**

size --是要分配的块的大小

flags –分配的指示,解释如下

GFP_ATOMIC
- 用来从中断处理和进程上下文之外的其他代码中分配内存. 从不睡眠.

GFP_KERNEL
- 内核内存的正常分配. 可能睡眠.

GFP_USER
- 用来为用户空间页来分配内存; 它可能睡眠.

GFP_HIGHUSER
- 如同 GFP_USER, 但是从高端内存分配

GFP_NOIO
GFP_NOFS

--这个标志功能如同 GFP_KERNEL, 但是它们增加限制到内核能做的来满足请求. 一个 GFP_NOFS 分配不允许进行任何文件系统调用, 而 GFP_NOIO 根本不允许任何 I/O 初始化. 它们主要地用在文件系统和虚拟内存代码。

使用系统内存

kmalloc()参数解释

上面列出的这些分配标志可以是下列标志的相或来作为参数, 这些标志改变这些分配如何进行:

__GFP_DMA

- 这个标志要求分配在能够 DMA 的内存区. 确切的含义是平台依赖的

__GFP_HIGHMEM

- 这个标志指示分配的内存可以位于高端内存.

__GFP_COLD

- 正常地, 内存分配器尽力返回"缓冲热"的页 -- 可能在处理器缓冲中找到的页. 相反, 这个标志请求一个"冷"页, 它在一段时间没被使用.

__GFP_NOWARN

- 这个很少用到的标志,当一个分配无法满足阻止内核来发出警告(使用 printk).

__GFP_HIGH

- 这个标志标识了一个高优先级请求, 它被允许来消耗甚至被内核保留给紧急状况的最后的内存页.

__GFP_REPEAT
__GFP_NOFAIL
__GFP_NORETRY .

- __GFP_REPEAT 意思是" 更尽力些尝试" 通过重复尝试 -- 但是分配可能仍然失败.
- __GFP_NOFAIL 标志告诉分配器不要失败; 它尽最大努力来满足要求. 使用 __GFP_NOFAIL 是强烈不推荐的;
- __GFP_NORETRY 告知分配器立即放弃如果得不到请求的内存.

使用系统内存

vmalloc()和vfree()

原形:

```
#include <linux/vmalloc.h>
void *vmalloc(unsigned long size);
void vfree(void * addr);
```

特性:

-- 分配的内存物理上不连续

-- 可以分配系统允许的大块内存,没有大小限制

-- 成功返回一个虚拟地址, 失败返回NULL

使用系统内存

- 物理地址和虚拟地址的转换

virt_to_phys(), __pa()

-- 将虚拟地址转换成物理地址

-- 不能转化**vmalloc()**的地址

-- 实现与机器相关

phys_to_virt(), __va()

-- 将物理地址转换成虚拟地址

使用系统内存

- 模块需要分配大块的内存，它常常最好是使用一个面向页的技术。

__get_free_page(unsigned int flags); /*返回一个指向新页的指针，未清零该页*/

get_zeroed_page(unsigned int flags); /*类似于__get_free_page，但用零填充该页*/

__get_free_pages(unsigned int flags, unsigned int order);
/*分配若干(物理连续的)页面并返回指向该内存区域的第一个字节的指针，该内存区域未清零*/

当程序不需要页面时，它可用下列函数之一来释放它们。

void free_page(unsigned long addr);
void free_pages(unsigned long addr, unsigned long order);

使用系统内存

- 操作实例：
- **unsigned char * m_free_page;**
- **unsigned char * m_free_pages;**

- **m_free_page = (unsignedchar*)__get_free_page(GFP_KERNEL);**
- **m_free_pages = (unsigned char*)__get_free_pages(GFP_KERNEL,2);**

- **memset(m_free_page,0,PAGE_SIZE);**
- **memset(m_free_pages,0,PAGE_SIZE*2*2);**

- **strcpy(m_free_page,"<<<--- Get Free Page OK! --- >>>");**
- **printk("m_free_page: %s\n", m_free_page);//**打印内容

- **free_page(m_free_page);**
- **free_pages(m_free_pages,2);**

使用系统内存

- 后备高速缓存
- 内核为驱动程序常常需要反复分配许多相同大小内存块的情况，增加了一些特殊的内存池，称为后备高速缓存（**lookaside cache**）。设备驱动程序通常不会涉及后备高速缓存，但是也有例外：在 **Linux 2.6** 中**USB** 和 **SCSI** 驱动。
- **Linux** 内核的高速缓存管理器称为**“slab** 分配器”。**slab** 分配器实现的高速缓存具有 **kmem_cache_t** 类型。对应的函数有：

```
#include <linux/malloc.h>
kmem_cache *kmem_cache_create(
    const char    *name,  size_t size, size_t offset, unsigned long flags,
 void (*ctor)(void *, kmem_cache_t *, unsigned long    flags));
```

- 这个函数创建一个新的可以驻留任意数目全部同样大小的内存区的缓存对象，大小由 size 参数指定. name 参数和这个缓存关联并且作为一个在追踪问题时有用的管理信息; 通常设置为被缓存的结构类型的名子. 这个缓存保留一个指向 name 的指针,这个名子不能包含空格.

使用系统内存

- **offset** 是页内的第一个对象的偏移; 它可被用来确保一个对被分配的对象的特殊对齐, 一般使用 **0** 来请求缺省值.
- **flags** 控制如何进行分配并且是下列标志的一个位掩码:
- **SLAB_NO_REAP**
- 设置这个标志确保缓存在系统查找内存时被削减. 重要的是避免不必要地限制内存分配器的行动自由.
- **SLAB_HWCACHE_ALIGN**
- 这个标志需要每个数据对象被对齐到一个缓存行; 实际对齐依赖主机平台的缓存分布. 这个选项可以是一个好的选择, 如果在 **SMP** 机器上你的缓存包含频繁存取的项. 但是, 用来获得缓存行对齐的填充可以浪费可观的内存量.
- **SLAB_CACHE_DMA**
- 这个标志要求每个数据对象在 **DMA** 内存区分配.

使用系统内存

- 函数的 ctor 参数是可选的构造函数，可以用来初始化新分配的对象
- 一旦一个对象的缓存被创建, 可以通过调用 kmem_cache_alloc 分配对象.
- void *kmem_cache_alloc(kmem_cache_t *cache, int flags);
- 这里, cache 参数是之前已经创建的缓存; flags 与kmalloc 的相同。
- 为释放一个对象, 使用 kmem_cache_free:
- void kmem_cache_free(kmem_cache_t *cache, const void *obj);
- 当驱动代码用完这个缓存, 典型地当模块被卸载, 它应当如下释放它的缓存:
- int kmem_cache_destroy(kmem_cache_t *cache);
- 这个销毁操作只在从这个缓存中分配的所有的对象都已返回给它时才成功. 因此, 一个模块应当检查从 kmem_cache_destroy 的返回值; 一个失败指示某类在模块中的内存泄漏(因为某些对象已被丢失.)

使用系统内存

- 操作实例：

- #inlcude <linux/slab.h>
- struct kmem_cache * my_kmem_cache;
- unsigned char * slabchar;

- my_kmem_cache = kmem_cache_create("slab_cache",4,0,SLAB_HWCACHE_ALIGN,NULL);

- slabchar = kmem_cache_alloc(my_kmem_cache,GFP_KERNEL);
- memset(slabchar,0,4);
- printk("Slab address : %x\n",(unsigned int)slabchar);

- kmem_cache_free(my_kmem_cache,slabchar);
- kmem_cache_destroy(my_kmem_cache);

内核的竞态

- 当在同一时间段出现两个或更多进程并且这些进程彼此交互（例如，共享相同的资源）时，就存在并发现象。

- 在单处理器（uniprocessor，UP）主机上可能发生并发，在这种主机中多个线程共享同一个 CPU 并且抢占（preemption）创建竞态条件。抢占 通过临时中断一个线程以执行另一个线程的方式来实现 CPU 共享。竞态条件发生在两个或更多线程操纵一个共享数据项时，其结果取决于执行的时间。在多处理器（MP）计算机中也存在并发，其中每个处理器中共享相同数据的线程同时执行。注意在 MP 情况下存在真正的并行（parallelism），因为线程是同时执行的。

- Linux 内核在两种模式中都支持并发。内核本身是动态的，而且有许多创建竞态条件的方法。Linux 内核也支持多处理（multiprocessing），称为对称多处理（SMP）。

内核的竞态

- 临界段概念是为解决竞态条件问题而产生的。一个临界段 是一段不允许多路访问的受保护的代码。这段代码可以操纵共享数据或共享服务（例如硬件外围设备）。临界段操作时坚持互斥锁（mutual exclusion）原则（当一个线程处于临界段中时，其他所有线程都不能进入临界段）。

- 临界段中需要解决的一个问题是死锁条件。考虑两个独立的临界段，各自保护不同的资源。每个资源拥有一个锁，在本例中称为 A 和 B。假设有两个线程需要访问这些资源，线程 X 获取了锁 A，线程 Y 获取了锁 B。当这些锁都被持有时，每个线程都试图占有其他线程当前持有的锁（线程 X 想要锁 B，线程 Y 想要锁 A）。这时候线程就被死锁了，因为它们都持有一个锁而且还想要其他锁。一个简单的解决方案就是总是按相同次序获取锁，从而使其中一个线程得以完成。

同步处理

- 同步概述

-- 同步问题是指不同的程序对相同的资源进行访问时,如何使数据一致的问题,比如对相同地址的内存进行访问(全局变量)

-- 同步的问题大多表现为互斥,也就是说对于共享的资源,在一个程序对其进行访问时,另外的程序不能对之访问.对于这样程序,我们称之为临界段

-- 驱动可以被多个进程调用, 经常会有同步问题的产生,驱动设计者需要时刻考虑数据是否有共享的存在

-- 产生同步的原因有

1) 多CPU的存在
2) 多任务和调度的不确定性
3) 中断的产生和不确定性

同步处理

- 一个例子

设计一个驱动,记录每个进程访问的次数,在驱动的open设置一个记数，并初始化为0，每次的读操作中对记数加1，单个进程执行时执行打印的数字正常：1,2,3...

```
static int  num= 0;
 int test_open(struct inode *inode, struct file *filp)
 {
        num=0
        return 0;
 }

 int test_read(struct file *file,char __user *buf,size_t const count,loff_t
  *offset)
 {
        printk(" num is %d \n",num++);
        return 0;
 }
```

但是, 多个进程同时执行时，很有可能会得不到我们希望的结果

同步处理

- 测试样例，十次读写操作，查看内核的信息：

```
static char sz[] = "this is a test string\n";
 static char readback[1024];

 int main(void)
 {
     int fd;
     int i;
     fd = open("/dev/test", O_RDWR);
     if(fd) {
      for(i=0;i<10;i++)
       {
         sleep(1);
            printf("I am writing my device…\n");
            write(fd,  sz, strlen(sz));
            read(fd, readback, strlen(sz) + 1);
           printf("the string I read back is : %s\n", readback);
       }
      }
     return 0;
 }
```

原子操作

- 有时，一个共享资源是一个简单的整数值．假设你的驱动维护一个共享变量 n_op，它告知有多少设备操作目前未完成．正常地，即便一个简单的操作例如：
- n_op++;

- 可能需要加锁．某些处理器可能以原子的方式进行那种递减，但是你不能依赖它．但是一个完整的加锁体制对于一个简单的整数值看来过分了．对于这样的情况，内核提供了一个原子整数类型称为 atomic_t，定义在 <asm/atomic.h>．

- 一个 atomic_t 持有一个 int 值在所有支持的体系上．但是，因为这个类型在某些处理器上的工作方式，整个整数范围可能不是都可用的；因此，你不应当指望一个 atomic_t 持有多于 24 位．原子操作是非常快的，因为它们在任何可能时编译成一条单个机器指令．

原子操作

- 内核提供了两类函数来实现位和整形变量的原子操作。由于 C 不能实现原子操作，因此 Linux 依靠底层架构来提供这项功能，确保每个操作过程都是原子的，不被其他任务所打断，各种底层架构存在很大差异，因此原子函数的实现方法也各不相同。一些方法完全通过汇编语言来实现，而另一些方法依靠 c 语言并且使用 local_irq_save 和 local_irq_restore 禁用中断。

- 下面是一个使用此 API 的示例。

- 1、要声明一个原子变量（atomic variable），首先声明一个 atomic_t 类型的变量。这个结构包含了单个 int 元素。

- 2、确保您的原子变量使用 ATOMIC_INIT 符号常量进行了初始化。
- atomic_t v = ATOMIC_INIT(0);
- 也可以使用 atomic_set() 在运行时对原子变量进行初始化。
- void atomic_set(atomic_t *v, int i);

- 3、获取原子变量的值，int atomic_read(atomic_t *v);

- 4、原子变量的加减：
- void atomic_add(int i, atomic_t *v);
- void atomic_sub(int i, atomic_t *v);

原子操作

- 5、源自变量的自增自减
- void atomic_inc(atomic_t *v);
- void atomic_dec(atomic_t *v);
- 递增或递减一个原子变量.

- 6、操作并测试
- int atomic_inc_and_test(atomic_t *v);
- int atomic_dec_and_test(atomic_t *v);
- int atomic_sub_and_test(int i, atomic_t *v);
- 进行一个特定的操作并且测试结果; 如果, 在操作后, 原子值是 0, 那么返回值是真; 否则, 它是假. 注意没有 atomic_add_and_test.

- 7、操作并返回
- int atomic_add_return(int i, atomic_t *v);
- int atomic_sub_return(int i, atomic_t *v);
- int atomic_inc_return(atomic_t *v);
- int atomic_dec_return(atomic_t *v);
- 就像 atomic_add 和其类似函数, 除了它们返回原子变量的新值给调用者.

原子操作

- 测试样例：

  设计一个驱动,设备只被一个进程打开,定义原子量并初始化为**1**，在驱动的**open**中对原子量进行自减检测，如果为**0**则继续操作设备，否则返回异常

```
#include <asm/atomic.h>

static  atomic_t  my_value = ATOMIC_INIT(1);

 int scull_open(struct inode *inode, struct file *filp)
 {
        if(!atomic_dec_and_test(&my_value))
        {
         atomic_inc(&my_value);
         return -EBUSY;
         }
        return 0;
 }

static int test_chardev_release(struct inode *inode,struct file *file)
{
      atomic_inc(&my_value);
      printk("close major=%d, minor=%d\n", imajor(inode), iminor(inode));
      return 0;
}
```

原子操作

- 当你需要以原子方式操作单个位时. 为此, 内核提供了一套函数来原子地修改或测试单个位. 因为整个操作在单步内发生, 没有中断(或者其他处理器)能干扰.
- 原子位操作非常快, 因为它们使用单个机器指令来进行操作, 而在任何时候低层平台做的时候不用禁止中断. 函数是体系依赖的并且在 <asm/bitops.h> 中声明. 它们保证是原子的, 即便在 SMP 计算机上, 并且对于跨处理器保持一致是有用的.
- 不幸的是, 键入这些函数中的数据也是体系依赖的. nr 参数(描述要操作哪个位)常常定义为 int, 但是在几个体系中是 unsigned long. 要修改的地址常常是一个 unsigned long 指针, 但是几个体系使用 void * 代替.
- 各种位操作是:
- void set_bit(nr, void *addr);
- 设置第 nr 位在 addr 指向的数据项中.
- void clear_bit(nr, void *addr);
- 清除指定位在 addr 处的无符号长型数据. 它的语义与 set_bit 的相反.

原子操作

- void change_bit(nr, void *addr);
- 翻转这个位.

- test_bit(nr, void *addr);
- 这个函数是唯一一个不需要是原子的位操作; 它简单地返回这个位的当前值.

- int test_and_set_bit(nr, void *addr);
- int test_and_clear_bit(nr, void *addr);
- int test_and_change_bit(nr, void *addr);
- 原子地动作如同前面列出的, 除了它们还返回这个位以前的值.

- 当这些函数用来存取和修改一个共享的标志, 除了调用它们不用做任何事; 它们以原子发生进行它们的操作. 使用位操作来管理一个控制存取一个共享变量的锁变量, 另一方面, 是有点复杂并且应该有个例子. 大部分现代的代码不以这种方法来使用位操作

信号量

Linux中的信号量是一种睡眠锁

- 如果有一个任务试图获得一个已被持有的信号量时，信号量会将其推入等待队列，然后让其睡眠。
- 当持有信号量的进程将信号量释放后，在等待队列中的一个任务将被唤醒，从而便可以获得这个信号量。
- 信号量的睡眠特性，使得信号量适用于锁会被长时间持有的情况

信号量的操作

信号量为一个整数，信号量的值与相应资源的使用情况有关。当它的值大于0时，表示当前可用资源的数量；当它的值小于0时，其绝对值表示等待使用该资源的进程个数。注意，信号量的值仅能由PV操作来改变。

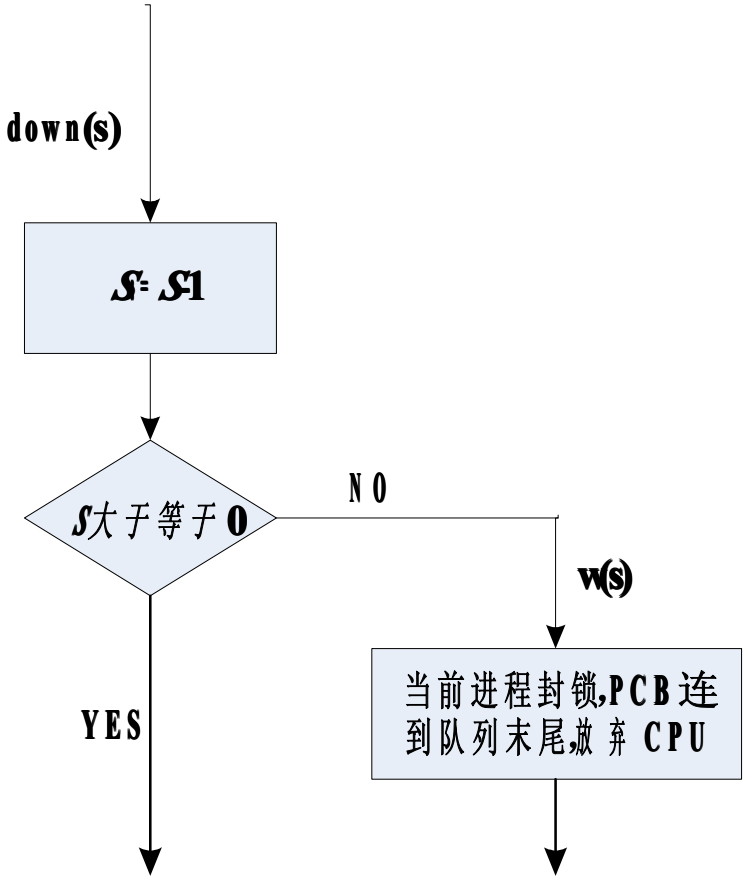
- 信号量支持两个原子操作P()和V()，Linux中分别叫做down()和up()。
- 不能在中断处理中进行信号的P()操作

信号量

- 一般来说，信号量S>=0时，S表示可用资源的数量。执行一次P操作意味着请求分配一个单位资源，因此S的值减1；
- 当S<0时，表示已经没有可用资源，请求者必须等待别的进程释放该类资源，它才能运行下去。而执行一个V操作意味着释放一个单位资源，因此S的值加1；
- 若S<=0，表示有某些进程正在等待该资源，因此要唤醒一个等待状态的进程，使之运行下去。
- 利用信号量和PV操作实现进程互斥的一般模型是：
- 进程P1 进程P2 …… 进程Pn
- …… …… ……
- P（S）； P（S）； P（S）；
- 临界区； 临界区； 临界区；
- V（S）； V（S）； V（S）；

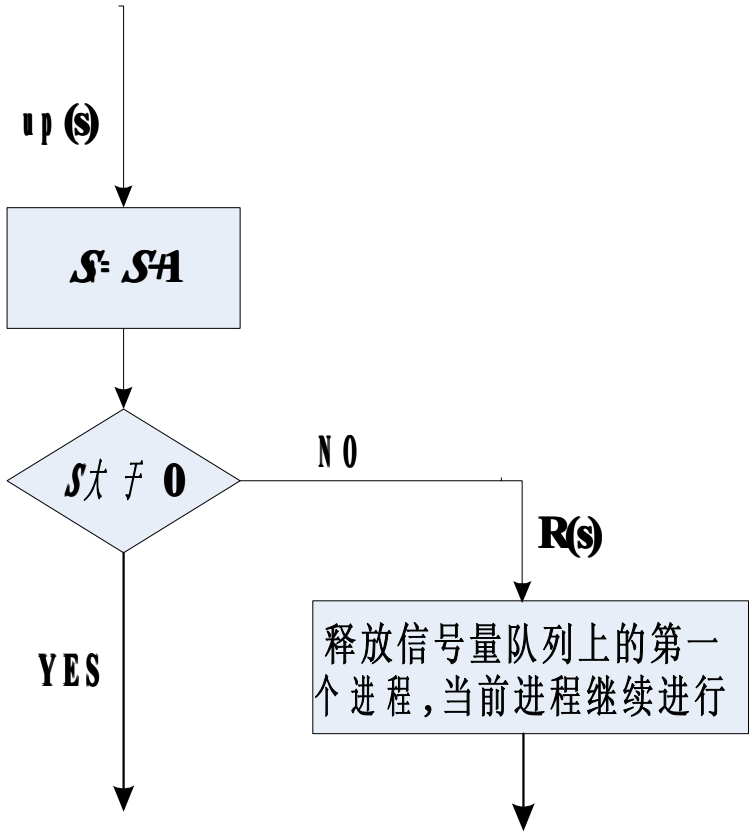
信号量

❖ 信号量(P操作)



信号量

❖ 信号量(V操作)



信号量

- **Linux信号量初始化**

#include <asm/semaphore.h>

- 直接创建一个信号量
- struct semaphore sem;
- 接着使用 sema_init 来初始化这个信号量
  void sema_init(struct semaphore *sem, int val)； 初始化为1的时候,就为互斥量

互斥模式的信号量声明，内核提供宏定义

- DECLARE_MUTEX(name);
  定义并初始化信号量为 1

- DECLARE_MUTEX_LOCKED(name);
  定义并初始化信号量为0

信号量

- 获取信号量，P操作

void down(struct semaphore *sem);

down减小信号量的值，获得信号量，并根据信号量的值决定是否进入睡眠，不可被信号打断，不能在中断上下文使用。

int down_interruptible(struct semaphore *sem);

该操作是可在睡眠中被信号打断，并返回0。

int down_trylock(struct semaphore *sem);

尝试获得信号量，获得并返回0，否则非0，不会睡眠。

- 释放信号量,V操作
  - **void up(struct semaphore *sem);**

通过down操作进入临界区的进程，再退出的时候都需要调用一个up操作，释放信号量。

信号量

❖ 测试样例：

设计一个驱动,设备只被一个进程打开,定义信号量并初始化为1，在驱动的open中对信号量进行获取，
如果成功则继续操作设备，否则返回异常

```
#include <asm/semaphore.h>

 DECLARE_MUTEX(my_sem);

int scull_open(struct inode *inode, struct file *filp)
   {
        if( down_trylock(&my_sem))
         {
          return -EBUSY;
          }
         return 0;
   }

static int test_chardev_release(struct inode *inode,struct file *file)
{
       up(&my_sem);
       printk("close major=%d, minor=%d\n", imajor(inode), iminor(inode));
       return 0;
}
```

自旋锁

❖ 自旋锁(spinlock)

-- 自旋锁是专为防止多处理器并发和强占式内核而引 入的一种同步机制，它可以被应用于内核的中断处理部分

-- 一个自旋锁是一个互斥设备, 只能有 2 个值:
"上锁"和"解锁".如果这个锁已经被别人获得, 代码进入一个忙的循环中反复检查这个锁,直到它变为可用. 这个循环就是自旋锁的"自旋"部分.

--自旋锁的临界区代码要尽量的快,以免影响性能

--对于单CPU的非强占式内核,内核不必实现自旋锁

--在单CPU和内核可抢占的系统中，自旋锁持有期间内核的抢占将被禁止。

自旋锁

- 1. 自旋锁在同一时刻至多被一个执行线程持有，所以一个时刻只能有一个线程位于临界区内，这就为多处理器提供了防止并发访问所需要的包含机制。在单处理器机器上，编译的时候并不会加入自旋锁。

- 2. 自旋锁不能递归！

- 3. 自旋锁可以使用在中断处理程序中(此处不能使用信号量，因为它会导致睡眠)。在中断处理程序中使用自旋锁时，一定要在获取锁之前，首先禁本地中断（在当前处理器上的中断请求），否则，中断处理程序就会打断正持有锁的内核代码，有可能会试图去争用这个已经被持有的自旋锁。注意，需要关闭的只是当前处理器上的中断，因为如果中断发生在不同的处理器上，即使中断处理程序在同一锁上的自旋，也不会妨碍锁的持有者最终释放锁。

自旋锁

❖ 自旋锁(spinlock)及中断屏蔽

--中断屏蔽是同步内核程序和中断处理程序产生的竞争关系而设的同步机制

--中断屏蔽是在本CPU上暂时屏蔽中断的产生，处理完临界区后再打开中断

--中断屏蔽对多CPU的系统无效,所以经常需用和自旋锁一起使用

--中断屏蔽的临界区要尽量快,以免引起中断丢失和系统效率

--自旋锁锁定期间不能调用可能引起进程调度的函数，如进程获得自旋锁后阻塞，则有可能引起系统崩溃。

自旋锁

- 自旋锁使用

#include <linux/spinlock.h>

一个自旋锁必须初始化. 这个初始化可以在编译时完成, 如下:

spinlock_t my_lock = SPIN_LOCK_UNLOCKED;

或者在运行时使用:

void spin_lock_init(spinlock_t *lock);

在进入一个临界区前, 你的代码必须获得需要的 lock , 用:

void spin_lock(spinlock_t *lock);

注意所有的自旋锁等待是由于它们的特性, 不可被信号中断的.
一旦你调用 spin_lock, 将自旋直到获得锁.

void spin_trylock(spinlock_t *lock);

非阻塞版本，如能获得锁立即返回真，否则返回假；

为释放一个你已获得的锁, 传递它给:

void spin_unlock(spinlock_t *lock);

自旋锁

- 自旋锁(spinlock)及中断屏蔽

- 尽管使用自旋锁可以避免其他进程的打扰，但是依然会受到中断的影响，为此，产生了自旋锁的衍生版本

```
void spin_lock_irqsave(spinlock_t *lock, unsigned long flags);
void spin_lock_irq(spinlock_t *lock);

void spin_unlock_irqrestore(spinlock_t *lock, unsigned long flags);
void spin_unlock_irq(spinlock_t *lock);
```

spin_lock_irqsave()保存中断的当然状态，并禁止本地中断，然后再去获取指定的锁。
spin_unlock_irqrestore ()对指定的锁解锁，然后让中断恢复到加锁前的状态。

不需要在上锁与解锁时对中断期间的状态进行记录。可以使用spin_lock_irq()和spin_unlock_irq()。

自旋锁在等待期间不做任何有有用的工作，仅为忙等待，因此在占用锁时间极短的情况下才使用自旋锁，递归使用自旋锁将导致系统崩溃，调用自旋锁期间不可使用任何调度导阻塞的函数。

自旋锁

测试样例：

设计一个驱动,设备只被一个进程打开,定义自旋锁并初始化为未上锁状态，在驱动的**open**中对自旋锁进行获取，如果获取成功则继续操作设备，对状态字进行判断及操作，否则返回异常

```
#include <linux/spinlock.h>
static int num=0;
spinlock_t my_lock = SPIN_LOCK_UNLOCKED;

   int scull_open(struct inode *inode, struct file *filp)
   {
         spin_lock(&my_lock);
         if( num)
          {
           spin_unlock(&my_lock);
           return -EBUSY;
           }
          num++;
          spin_unlock(&my_lock);
          return 0;
   }

static int test_chardev_release(struct inode *inode,struct file *file)
{
      spin_lock(&my_lock);
      num--;
      spin_unlock(&my_lock);
       printk("close major=%d, minor=%d\n", imajor(inode), iminor(inode));
       return 0;
}
```

读写锁

- 读写锁实际是一种特殊的自旋锁，它把对共享资源的访问者划分成读者和写者，读者只对共享资源进行读访问，写者则需要对共享资源进行写操作。这种锁相对于自旋锁而言，能提高并发性，因为在多处理器系统中，它允许同时有多个读者来访问共享资源，最大可能的读者数为实际的逻辑CPU数。写者是排他性的，一个读写锁同时只能有一个写者或多个读者（与CPU数相关），但不能同时既有读者又有写者。

- 在读写锁保持期间也是抢占失效的。

- 如果读写锁当前没有读者，也没有写者，那么写者可以立刻获得读写锁，否则它必须自旋在那里，直到没有任何写者或读者。如果读写锁没有写者，那么读者可以立即获得该读写锁，否则读者必须自旋在那里，直到写者释放该读写锁。

- 读写锁的API看上去与自旋锁很象，只是读者和写者需要不同的获得和释放锁的API。

读写锁

- 读者写者锁有一个类型 rwlock_t, 在 <linux/spinlokc.h> 中定义
- 1、定义和初始化读写自旋锁
- rwlock_t my_rwlock = RW_LOCK_UNLOCKED; /* Static way */
- 等价于
- rwlock_t my_rwlock;
- rwlock_init(&my_rwlock); /* Dynamic way */

- 2、读锁定
- void read_lock(rwlock_t *lock);
- void read_lock_irqsave(rwlock_t *lock, unsigned long flags);
- void read_lock_irq(rwlock_t *lock);
- void read_lock_bh(rwlock_t *lock);

- 3、读解锁
- void read_unlock(rwlock_t *lock);
- void read_unlock_irqrestore(rwlock_t *lock, unsigned long flags);
- void read_unlock_irq(rwlock_t *lock);
- void read_unlock_bh(rwlock_t *lock);
- 对共享资源进行读之前，应当先调用读锁定函数，完成后调用读解锁函数

读写锁

- 4、写锁定
- void write_lock(rwlock_t *lock);
- void write_lock_irqsave(rwlock_t *lock, unsigned long flags);
- void write_lock_irq(rwlock_t *lock);
- void write_lock_bh(rwlock_t *lock);
- int write_trylock(rwlock_t *lock);

- 5、写解锁
- void write_unlock(rwlock_t *lock);
- void write_unlock_irqrestore(rwlock_t *lock, unsigned long flags);
- void write_unlock_irq(rwlock_t *lock);
- void write_unlock_bh(rwlock_t *lock);
- 对共享资源进行写之前，应当先调用写锁定函数，完成后调用写解锁函数

读写锁

- 测试样例：

设计一个驱动,设备只被一个进程打开,定义读写锁并初始化为未上锁状态，在驱动的**read**中对读写锁的读锁进行获取，另外在驱动的**write**中对读写锁的写锁进行获取，读锁如果成功则读取设备状态，否则自旋等待，写锁如果成功则修改设备状态，否则自旋等待，

```
#include <linux/spinlock.h>
static int num=0;
stati unsigned long flags
rwlock_t my_rwlock = RW_LOCK_UNLOCKED;

static ssize_t test_chardev_read(struct file *file,char __user *buf,
                        size_t const count,loff_t *offset)
{
          ...
            read_lock(&my_rwlock);
            tmpbuf=*gpbd;  //GPIO读取数据
            read_unlock(&my_rwlock);

             if(copy_to_user(buf,tmpbuf,count))
             {
               printk("copy to user fail \n");
               return -EFAULT;
             }
         ...
}
```

读写锁

```
static ssize_t test_chardev_write(struct file *file, const char __user *buf,size_t const count,loff_t *offset)
{
   ....
     if(copy_from_user(tmpbuf,buf,count))
       {
         printk("copy from user fail \n");
         return -EFAULT;
       }

     write_lock(&my_rwlock);
     *gpbc=*gpbc & ~(0xFF<<10) | (tmpbuf[0]<<10);
     *gpbd=*gpbd & ~(0xF<<5) | (0xF<<5);
     *gpbu=*gpbu & ~(0xF<<5) | (0xF<<5);
     write_unlock(&my_rwlock);
   ....
}
```

顺序锁

- 顺序锁也是对读写锁的一种优化，对于顺序锁，读者绝不会被写者阻塞，也就说，读者可以在写者对被顺序锁保护的共享资源进行写操作时仍然可以继续读，而不必等待写者完成写操作，写者也不需要等待所有读者完成读操作才去进行写操作。但是，写者与写者之间仍然是互斥的，即如果有写者在进行写操作，其他写者必须自旋在那里，直到写者释放了顺序锁。

- 这种锁有一个限制，它必须要求被保护的共享资源不含有指针，因为写者可能使得指针失效，但读者如果正要访问该指针，将导致OOPs。

- 如果读者在读操作期间，写者已经发生了写操作，那么，读者必须重新读取数据，以便确保得到的数据是完整的。

- 这种锁对于读写同时进行的概率比较小的情况，性能是非常好的，而且它允许读写同时进行，因而更大地提高了并发性。

顺序锁

- seqlock 定义在 <linux/seqlock.h>.
- 有 2 个通常的方法来初始化一个 seqlock(有 seqlock_t 类型):

- seqlock_t lock1 =SEQLOCK_UNLOCKED;
- 等价于
- seqlock_t lock2;
- seqlock_init(&lock2);

- void write_seqlock(seqlock_t *sl);

- 写者在访问被顺序锁s1保护的共享资源前需要调用该函数来获得顺序锁s1。它实际功能上等同于spin_lock，只是增加了一个对顺序锁顺序号的加1操作，以便读者能够检查出是否在读期间有写者访问过。

- void write_sequnlock(seqlock_t *sl);

- 写者在访问完被顺序锁s1保护的共享资源后需要调用该函数来释放顺序锁s1。它实际功能上等同于spin_unlock，只是增加了一个对顺序锁顺序号的加1操作，以便读者能够检查出是否在读期间有写者访问过。

顺序锁

- int write_tryseqlock(seqlock_t *sl);
- 写者在访问被顺序锁s1保护的共享资源前也可以调用该函数来获得顺序锁s1。它实际功能上等同于spin_trylock，只是如果成功获得锁后，该函数增加了一个对顺序锁顺序号的加1操作，以便读者能够检查出是否在读期间有写者访问过。

- unsigned read_seqbegin(const seqlock_t *sl);
- 读者在对被顺序锁s1保护的共享资源进行访问前需要调用该函数。读者实际没有任何得到锁和释放锁的开销，该函数只是返回顺序锁s1的当前顺序号。

- int read_seqretry(const seqlock_t *sl, unsigned iv);
- 读者在访问完被顺序锁s1保护的共享资源后需要调用该函数来检查，在读访问期间是否有写者访问了该共享资源，如果是，读者就需要重新进行读操作，否则，读者成功完成了读操作。

顺序锁

- 读存取通过在进入临界区入口获取一个(无符号的)整数序列来工作. 在退出时, 那个序列值与当前值比较; 如果不匹配, 读存取必须重试. 读者代码可采用下面的形式:

- unsigned int seq;

- do {
- seq = read_seqbegin(&the_lock);
- /* Do what you need to do */
- .....
- } while read_seqretry(&the_lock, seq);

- 这个类型的锁常常用在保护某种简单计算, 需要多个一致的值. 如果这个计算最后的测试表明发生了一个并发的写, 结果被简单地丢弃并且重新计算.

顺序锁

- write_seqlock_irqsave(lock, flags)
- 写者也可以用该宏来获得顺序锁lock，与write_seqlock不同的是，该宏同时还把标志寄存器的值保存到变量flags中，并且失效了本地中断。

- write_seqlock_irq(lock)
- 写者也可以用该宏来获得顺序锁lock，与write_seqlock不同的是，该宏同时还失效了本地中断。与write_seqlock_irqsave不同的是，该宏不保存标志寄存器。

- write_seqlock_bh(lock)
- 写者也可以用该宏来获得顺序锁lock，与write_seqlock不同的是，该宏同时还失效了本地软中断。

顺序锁

- write_sequnlock_irqrestore(lock, flags)
- 写者也可以用该宏来释放顺序锁lock，与write_sequnlock不同的是，该宏同时还把标志寄存器的值恢复为变量flags的值。它必须与write_seqlock_irqsave配对使用。

- write_sequnlock_irq(lock)
- 写者也可以用该宏来释放顺序锁lock，与write_sequnlock不同的是，该宏同时还使能本地中断。它必须与write_seqlock_irq配对使用。

- write_sequnlock_bh(lock)
- 写者也可以用该宏来释放顺序锁lock，与write_sequnlock不同的是，该宏同时还使能本地软中断。它必须与write_seqlock_bh配对使用。

- read_seqbegin_irqsave(lock, flags)
- 读者在对被顺序锁lock保护的共享资源进行访问前也可以使用该宏来获得顺序锁lock的当前顺序号，与read_seqbegin不同的是，它同时还把标志寄存器的值保存到变量flags中，并且失效了本地中断。注意，它必须与read_seqretry_irqrestore配对使用。

顺序锁

- read_seqretry_irqrestore(lock, iv, flags)

- 读者在访问完被顺序锁lock保护的共享资源进行访问后也可以使用该宏来检查，在读访问期间是否有写者访问了该共享资源，如果是，读者就需要重新进行读操作，否则，读者成功完成了读操作。它与read_seqretry不同的是，该宏同时还把标志寄存器的值恢复为变量flags的值。注意，它必须与 read_seqbegin_irqsave配对使用。

- 因此，读者使用顺序锁的模式也可以为：

- do {
- seqnum = read_seqbegin_irqsave(&seqlock_a, flags);
- //读操作代码
- ...
- } while (read_seqretry_irqrestore(&seqlock_a, seqnum, flags));
-
- 读者和写者所使用的API的几个版本应该如何使用与自旋锁的类似。

顺序锁

- 如果写者在操作被顺序锁保护的共享资源时已经保持了互斥锁保护对共享数据的写操作，即写者与写者之间已经是互斥的，但读者仍然可以与写者同时访问，那么这种情况仅需要使用顺序计数（seqcount），而不必要spinlock。

- 顺序计数的API如下：

- unsigned read_seqcount_begin(const seqcount_t *s);
- 读者在对被顺序计数保护的共享资源进行读访问前需要使用该函数来获得当前的顺序号。

- int read_seqcount_retry(const seqcount_t *s, unsigned iv);
- 读者在访问完被顺序计数s保护的共享资源后需要调用该函数来检查，在读访问期间是否有写者访问了该共享资源，如果是，读者就需要重新进行读操作，否则，读者成功完成了读操作。

顺序锁

- 因此，读者使用顺序计数的模式如下：

- do {
- seqnum = read_seqbegin_count(&seqcount_a);
- //读操作代码
- } while (read_seqretry(&seqcount_a, seqnum));

- void write_seqcount_begin(seqcount_t *s);
- 写者在访问被顺序计数保护的共享资源前需要调用该函数来对顺序计数的顺序号加1，以便读者能够检查出是否在读期间有写者访问过。

- void write_seqcount_end(seqcount_t *s);
- 写者在访问完被顺序计数保护的共享资源后需要调用该函数来对顺序计数的顺序号加1，以便读者能够检查出是否在读期间有写者访问过。

- 写者使用顺序计数的模式为：
- write_seqcount_begin(&seqcount_a);
- /写操作代码...
- write_seqcount_end(&seqcount_a);
- 需要特别提醒，顺序计数的使用必须非常谨慎，只有确定在访问共享数据时已经保持了互斥锁才可以使用。

中断处理

- 什么是中断？

  -- 先打个比方。当一个经理正处理文件时，电话铃响了（中断请求），不得不在文件上做一个记号（返 回地址），暂停工作，去接电话（中断），并指示"按第二方案办"（调中断服务程序），然后，再静下心来（恢复中断前状态），接着处理文件.....。

  -- 中断是CPU处理外部突发事件的一个重要技术。它能使CPU在运行过程中对外部事件发出的中断请求及时地进行处理，处理完成后又立即返回断点，继续进行CPU原来的工作。

- 中断的作用

  -- 提高系统的效率,由于CPU速度远比
  外设高,轮询的效率非常底下

  -- 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

中断处理

- 中断的概念

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

中断类型 -- 外部中断,内部中断. 外部中断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。内部中断是指因硬件出错（如突然掉电、奇偶校验错等）或运算出错（除数为零、运算溢出、单步中断等）所引起 的中断,也叫异常。内部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。

中断处理

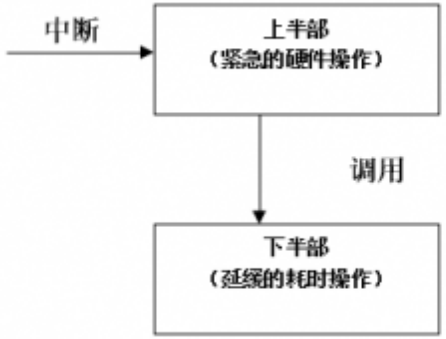
Linux的中断

-- Linux把中断处理分为两半,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,这部分要非常块完成,在这部分处理中,所有中断被屏蔽

-- 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

所有的中断的处理程序在init_IRQ函数中都被初始化为interrupt[i]。interrupt数组中每一项均指向一个代码片段：

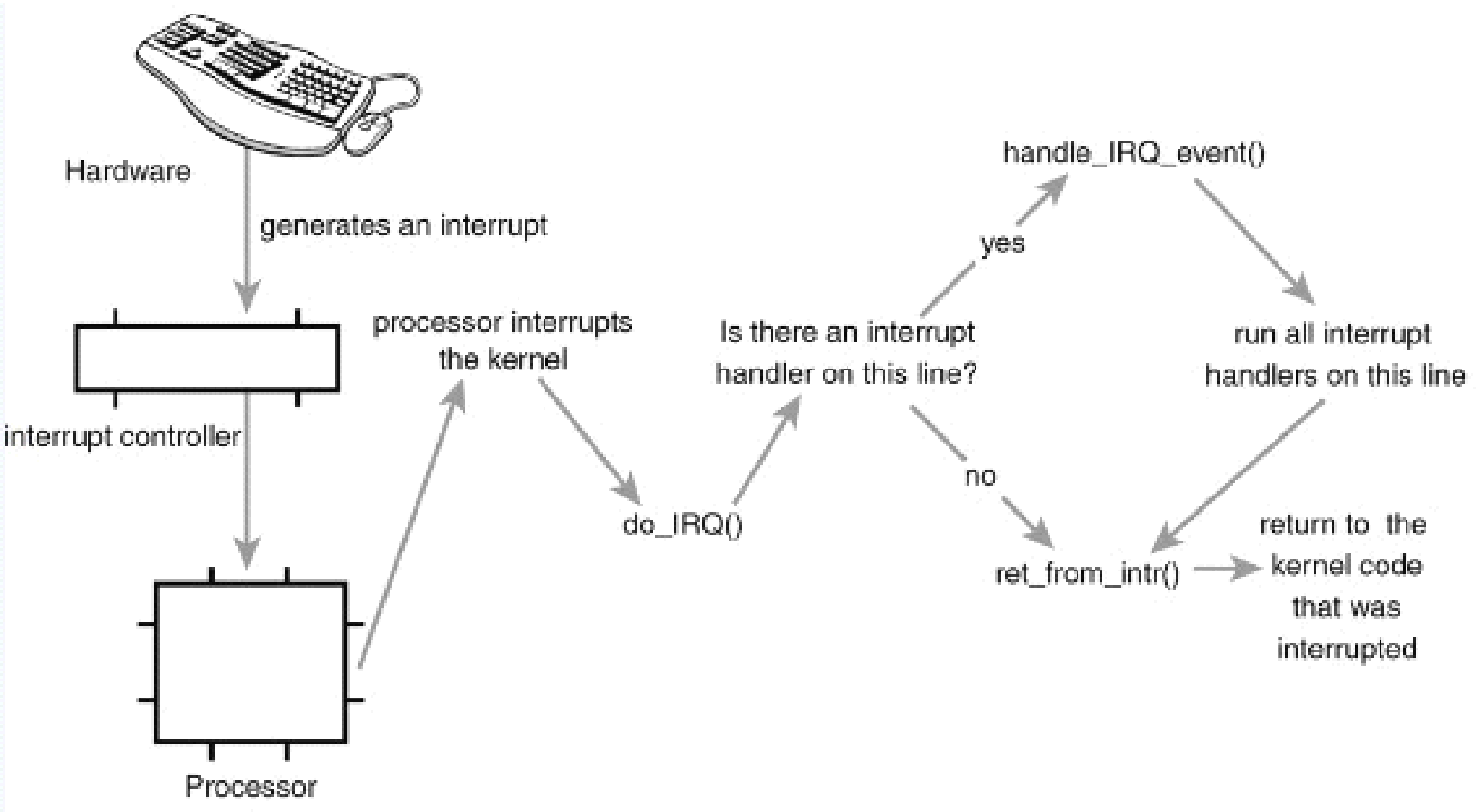
```
pushl $n-256
/*省略部分代码*/
jmp common_interrupt
```

该代码片段除了将中断向量号压入堆栈，还会跳到一个公共处理程序common_interrup：

```
//在linux/arch/x86/kernel/entry_32.S
 863 common_interrupt:
 864         addl $-0x80,(%esp)      /* Adjust vector into the [-256,-1]
range *     /
 865         SAVE_ALL
 866         TRACE_IRQS_OFF
 867         movl %esp,%eax
 868         call do_IRQ
 869         jmp ret_from_intr
```

这段公共处理程序会将中断发生前的所有寄存器的值压入堆栈，也就是保存被中断任务的现场。然后调用do_IRQ函数，在do_IRQ函数中会调用（并非直接调用那么简单）到handle_IRQ_event函数，在此函数中会执行实际的中断服务例程。当中断服务例程执行完毕后，会返回到上面的那段汇编程序中，转入ret_from_intr代码段从中断返回。

中断处理


Hardware
generates an interrupt
interrupt controller
processor interrupts
the kernel
Processor
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

```
31struct irq_desc;
 32typedef void (*irq_flow_handler_t)(unsigned int irq,
 33                                   struct irq_desc *desc);
175struct irq_desc {
176         unsigned int              irq;
177         struct timer_rand_state   *timer_rand_state;
178         unsigned int              *kstat_irqs;
179#ifdef CONFIG_INTR_REMAP
180         struct irq_2_iommu        *irq_2_iommu;
181#endif
182         irq_flow_handler_t        handle_irq;
183         struct irq_chip           *chip;
184         struct msi_desc           *msi_desc;
185         void                      *handler_data;
186         void                      *chip_data;
187         struct irqaction          *action;
188         unsigned int              status;
189
190         unsigned int              depth;
191         unsigned int              wake_depth;
192         unsigned int              irq_count;
193         unsigned long             last_unhandled;
194         unsigned int              irqs_unhandled;
195         raw_spinlock_t            lock;
196#ifdef CONFIG_SMP
197         cpumask_var_t             affinity;
198         const struct cpumask      *affinity_hint;
199         unsigned int              node;
200#ifdef CONFIG_GENERIC_PENDING_IRQ
201         cpumask_var_t             pending_mask;
202#endif
203#endif
204         atomic_t                  threads_active;
205         wait_queue_head_t         wait_for_threads;
206#ifdef CONFIG_PROC_FS
207         struct proc_dir_entry     *dir;
208#endif
209         const char                *name;
210}
216#ifndef CONFIG_SPARSE_IRQ
217extern struct irq_desc irq_desc[NR_IRQS];
218#endif
```

struct irq_desc

- 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：通过数据类型可知这便是这个描述符所对应的中断号；
- handle_irq：指向该IRQ线的公共服务程序；
- chip：它是一个struct irq_chip类型的指针，是中断控制器的描述符，与平台有关。
- handler_data：用于handler_irq的参数；
- chip_data：用于chip的参数；
- action：一个struct irqaction类型的指针（下文有该结构的详细描述）；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：描述中断线当前的状态；
- 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；
- name： /proc/interrupts 中显示的中断名称；

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：

```
98typedef irqreturn_t (*irq_handler_t)(int, void *);
113struct irqaction {
114         irq_handler_t handler;
115         unsigned long flags;
116         const char *name;
117         void *dev_id;
118         struct irqaction *next;
119         int irq;
120         struct proc_dir_entry *dir;
121         irq_handler_t thread_fn;
122         struct task_struct *thread;
123         unsigned long thread_flags;
124};
```

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。

- 比如经典的中断控制器是2片级联的8259A，那么得15个irq_desc描述符，每一个描述符的irq_chip都指向描述8259A的i8259A_irq_type变量(arch/alpha/kernel/irq_i8259.c)：

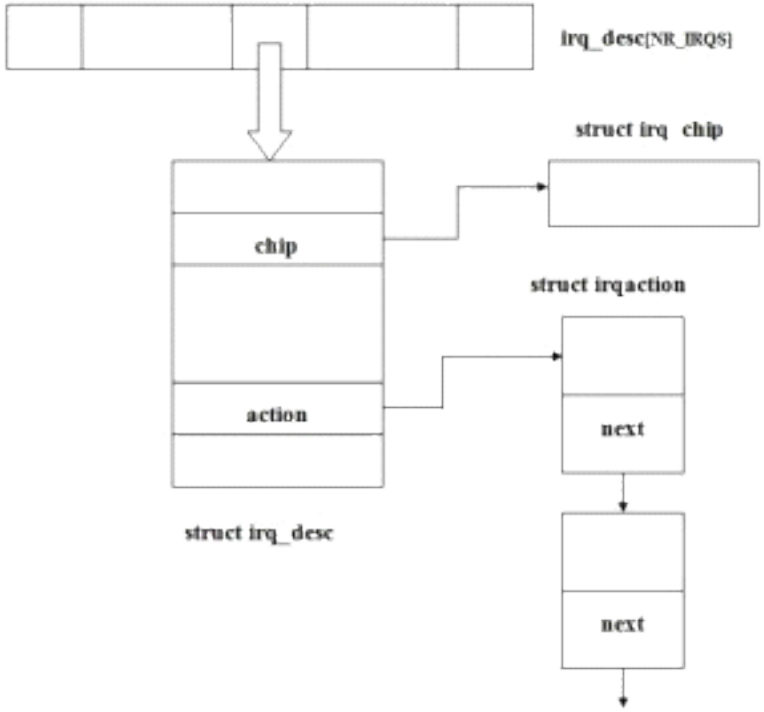
```
1  86struct irq_chip i8259a_irq_type = {
2   87          .name           = "XT-PIC",
3   88          .startup        = i8259a_startup_irq,
4   89          .shutdown       = i8259a_disable_irq,
5   90          .enable         = i8259a_enable_irq,
6   91          .disable        = i8259a_disable_irq,
7   92          .ack            = i8259a_mask_and_ack_irq,
8   93          .end            = i8259a_end_irq,
9   94};
```

这一点类似于VFS中所采用的原理：通过struct file_operations提供统一的接口，每种文件系统都必须具体实现这个结构体中所提供的接口。

struct irq_chip

```
01 111struct irq_chip {
02 112          const char      *name;
03 113          unsigned int    (*startup)(unsigned int irq);
04 114          void            (*shutdown)(unsigned int irq);
05 115          void            (*enable)(unsigned int irq);
06 116          void            (*disable)(unsigned int irq);
07 117
08 118          void            (*ack)(unsigned int irq);
09 119          void            (*mask)(unsigned int irq);
10 120          void            (*mask_ack)(unsigned int irq);
11 121          void            (*unmask)(unsigned int irq);
12 122          void            (*eoi)(unsigned int irq);
13 123
14 124          void            (*end)(unsigned int irq);
15 125          int             (*set_affinity)(unsigned int irq,
16 126                                          const struct cpumask *dest);
17 127          int             (*retrigger)(unsigned int irq);
18 128          int             (*set_type)(unsigned int irq, unsigned int
   flow_type);
19 129          int             (*set_wake)(unsigned int irq, unsigned int on);
20 130
21 131          void            (*bus_lock)(unsigned int irq);
22 132          void            (*bus_sync_unlock)(unsigned int irq);
23 133
24 134          /* Currently used only by UML, might disappear one day.*/
25 135#ifdef CONFIG_IRQ_RELEASE_METHOD
26 136          void            (*release)(unsigned int irq, void *dev_id);
27 137#endif
28 138          /*
29 139           * For compatibility, ->typename is copied into ->name.
30 140           * Will disappear.
31 141           */
32 142          const char      *typename;
33 143};
```

三个数据结构的关系



中断处理程序的调用过程：在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。

中断处理

- 安装一个中断处理

```
#inclue <linux/interrupt.h>

int request_irq(
        unsigned int irq,
        irqreturn_t (*handler)(int, void *),
        unsigned long flags, const char *dev_name, void *dev_id);
```

从 request_irq 返回给请求函数的返回值或者是 0 指示成功, 或者是一个负的错误码, 如同平常. 函数返回 -EBUSY 来指示另一个驱动已经使用请求的中断线是不寻常的.

- 释放一个中断处理

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

中断处理

- **request_irq函数的参数说明**
- irq是要申请的硬件中断号。
- handler是向系统注册的中断处理函数，是一个回调函数，中断发生时，系统调用这个函数，dev_id参数将被传递给它。
- irqflags是中断处理的属性，若设置了IRQF_DISABLED （老版本中的SA_INTERRUPT），则表示中断处理程序是快速处理程序，快速处理程序被调用时屏蔽所有中断，慢速处理程序不屏蔽；若设置了IRQF_SHARED （老版本中的SA_SHIRQ），则表示多个设备共享中断。
- devname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()返回0表示成功，返回-INVAL表示中断号无效或处理函数指针为NULL，返回-EBUSY表示中断已经被占用且不能共享。

中断处理

- 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。

- 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。

中断处理

- 启动和禁用中断
  - 驱动禁止和使能特定中断线的中断：
    - #include <asm/irq.h>.
    - void disable_irq(int irq);
    - void enable_irq(int irq);
  - 禁止所有中断
    - void local_irq_save(unsigned long flags);
    - local_irq_save 在当前处理器上禁止中断递交, 在保存当前中断状态到 flags。
    - void local_irq_disable(void);
    - local_irq_disable 关闭本地中断递交而不保存状态
- 打开中断
  - 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);

中断处理

-- Linux把中断处理例程分两部分：

- 上半分：实际响应中断的例程。
- 下半分：被顶部分调用，通过开中断的方式进行。

- 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

- Tasklet(小任务机制)

-- 内核在BH机制的基础上进行了扩展， 实现"软中断请求"（softirq）机制。利用软中断代替 bottom half handler 的处理。

-- tasklet 机制正是利用软中断来完成对驱动 bottom half 的处理。

-- tasklet会让内核选择某个合适的时间来执行给定的小任务。

Tasklet

- 小任务**tasklet**的实现

- 其数据结构为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

```
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*/
};
```

- state的取值参照下边的枚举型：

```
enum
{
    TASKLET_STATE_SCHED,    /* 小任务已被调用执行*/
    TASKLET_STATE_RUN    /*仅在多处理器上使用*/
};
```

- 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。

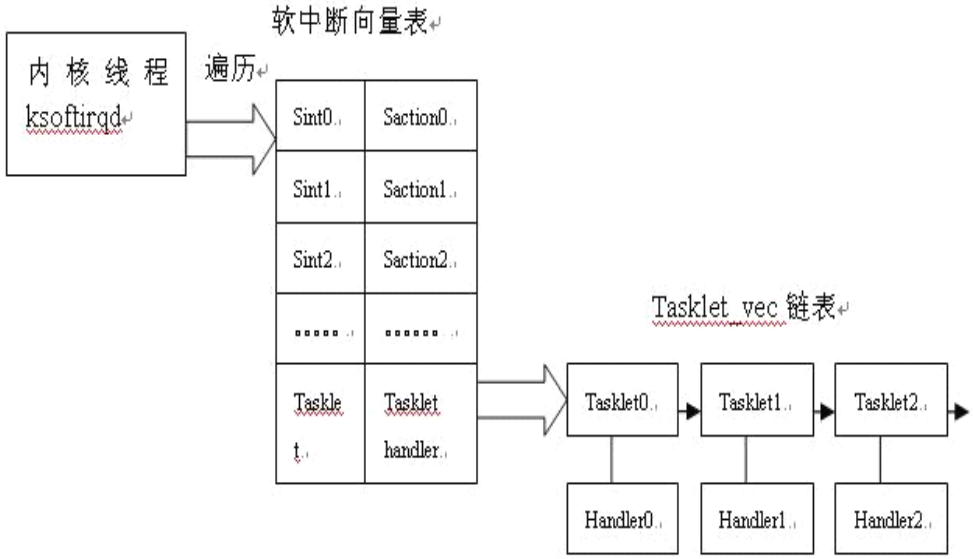
Tasklet

- Tasklet

-- 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（ pend ），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

- **I、声明和使用小任务tasklet**

- 静态的创建一个小任务的宏有一下两个：

- **#define DECLARE_TASKLET(name, func, data) **
- **struct tasklet_struct name = { NULL, 0, ATOMIC_INIT(0), func, data }**

- name 是 tasklet 的名字，
- Func 是执行 tasklet 的函数；
- data 是 unsigned long 类型的 function 参数。

- **#define DECLARE_TASKLET_DISABLED(name, func, data) **
- **struct tasklet_struct name = { NULL, 0, ATOMIC_INIT(1), func, data }**

- 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。

- 其中ATOMIC_INIT宏为：
- **#define ATOMIC_INIT(i) { (i) }**

- 此宏在include/asm-generic/atomic.h中定义。这样就创建了一个名为name的小任务，其处理函数为func。当该函数被调用的时候，data参数就被传递给它。

Tasklet

- **II、小任务处理函数程序**

- 处理函数的的形式为：void my_tasklet_func(unsigned long data)。这样DECLARE_TASKLET(my_tasklet, my_tasklet_func, data)实现了小任务名和处理函数的绑定，而data就是函数参数。

- **III、调度编写的tasklet**

- 调度小任务时引用tasklet_schedule(&my_tasklet)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) 。

- 这个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 handler）后，转去执行下半部的小任务。

Tasklet

tasklet 的接口

void tasklet_disable(struct tasklet_struct *t);

- 这个函数禁止给定的 tasklet. tasklet ， 但仍然可以被 tasklet_schedule 调度, 但是它的执行被延后直到这个 tasklet 被再次激活。

void tasklet_enable(struct tasklet_struct *t);

- 激活一个之前被禁止的 tasklet. 如果这个 tasklet 已经被调度, 它会很快运行.
- 一个对tasklet_enable 的调用必须匹配每个对 tasklet_disable 的调用, 因为内核跟踪每个 tasklet 的"禁止次数".

void tasklet_hi_schedule(struct tasklet_struct *t);

- 调度 tasklet 在更高优先级执行. 当软中断处理运行时, 它在其他软中断之前处理高优先级 tasklet。

void tasklet_kill(struct tasklet_struct *t);

- 这个函数确保了这个 tasklet 没被再次调度来运行; 它常常被调用当一个设备正被关闭或者模块卸载时. 如果这个 tasklet 被调度来运行, 这个函数等待直到它已执行.

Tasklet

- 使用tasklet作为下半部的处理中断的设备驱动程序模板如下：

```
/*定义tasklet和下半部函数并关联*/
void my_do_tasklet(unsigned long);
DECLARE_TASKLET(my_tasklet, my_tasklet_func, 0);

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

/*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

```
/*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);
……
}
```

workqueue

工作队列workqueue

- 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。这种差异的本质原因是，在工作队列机制中，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。也就是说由工作队列所执行的中断代码会表现出进程的一些特性，最典型的就是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就不能睡眠。

- 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。

workqueue

- 工作队列的实现

- 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
};
```

- 这些结构被连接成链表。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。

workqueue

- 工作的创建（静态方法）

```
#define DECLARE_WORK(n, f)                                  \
        struct work_struct n = __WORK_INITIALIZER(n, f)
```

- 而后边这个宏为一下内容：

```
#define __WORK_INITIALIZER(n, f) {                          \
    .data = WORK_DATA_INIT(),                               \
    .entry      = { &(n).entry, &(n).entry },               \
    .func = (f),                                            \
    __WORK_INIT_LOCKDEP_MAP(#n, &(n))                       \
    }
```

- 其为参数data赋值的宏定义为：

```
#define WORK_DATA_INIT()      ATOMIC_LONG_INIT(0)
```

- 这样就会静态地创建一个名为n，待执行函数为f，参数为data的work_struct结构。

workqueue

- 工作的创建（动态方法）
- 在运行时通过指针创建一个工作：
- INIT_WORK(struct work_struct *work, void(*func) (void *));
- 这会动态地初始化一个由work指向的工作队列，并将其与处理函数绑定。宏原型为：

```
#define INIT_WORK(_work, _func)                                 \
    do {                                                \
        static struct lock_class_key __key;             \
                                                        \
        (_work)->data = (atomic_long_t) WORK_DATA_INIT();  \
        lockdep_init_map(&(_work)->lockdep_map, #_work, &__key, 0);\
        INIT_LIST_HEAD(&(_work)->entry);                \
        PREPARE_WORK((_work), (_func));                 \
    } while (0)
```

workqueue

- 在需要调度的时候引用类似tasklet_schedule()函数的相应调度工作队列执行的函数schedule_work()，如：
- schedule_work(&work)；/*调度工作队列执行*/
- 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：
- schedule_delayed_work(&work,delay);函数原型为：
- 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。

workqueue

❖工作队列workqueue

建立 work_struct 结构并初始化， 使用下面宏:

INIT_WORK(struct work_struct *work, void (*function)(void *), void *data);

一个工作队列必须明确的在使用前创建，宏为:

struct workqueue_struct *create_workqueue(const char *name);

当用完一个工作队列，可以去掉它，使用:

void destroy_workqueue(struct workqueue_struct *queue);

把任务(work_struct)加入到工作队列中

int **queue_work**(struct workqueue_struct *queue, struct work_struct *work);

int **queue_delayed_work**(struct workqueue_struct *queue, struct work_struct *work, unsigned long delay);

//delay是为了保证至少在经过一段给定的最小延迟时间以后，工作队列中的任务才可以真正执行

workqueue

1.任何一个在工作队列中等待了无限长的时间也没有运行的任务可以用下面的方法取消：

```
int cancel_delayed_work(struct work_struct *work);
```

2.清空工作队列中的所有任务使用：

```
void flush_workqueue(struct workqueue_struct *queue);
```

3.销毁工作队列使用：

```
void destroy_workqueue(struct workqueue_struct *queue);
```

向内核缺省工作队列中加入任务

```
int schedule_work(struct work_struct *work);

int schedule_delayed_work(struct work_struct *work, unsigned long delay);
```

workqueue

- 使用工作队列处理中断下半部的设备驱动程序模板如下：

```
/*定义工作队列和下半部函数并关联*/
struct work_struct my_wq;

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

/*中断处理上半部*/
irpreturn_t my_interrupt(unsigned int irq,void *dev_id)
{
 ……
 schedule_work(&my_wq)/*调度my_wq函数，根据工作队列初始化函数将去执行my_do_work函数*/
 ……
}
```

workqueue

```
/*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);
 ……
 /*初始化工作队列函数,并与自定义处理函数关联*/
 INIT_WORK(&my_irq,(void (*)(void *))my_do_work);
 ……
}

/*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);
……
}
```

内核定时器

- 内核定时器 （也称为动态定时器）是内核在以后某一个时刻运行一段程序或进程的基础，软件定时器可以在一个确切的时间点上（更严格地说是一个时间点以后）激活相应的程序段或进程。软件定时器在设备驱动程序中被大量应用以检测设备的状态。

- 使用一个软件定时器很简单，只需做一些初始化工作，设置一个相对于当前时刻的超时时间和超时处理函数，将其插入到内核定时器队列中即可，设置的超时处理函数会在定时器超时时自动运行。

- 时钟和定时器对Linux内核来说十分重要。首先内核要管理系统的运行时间(uptime)和当前墙上时间(wall time)， 即当前实际时间。其次，内核中大量的活动由时间驱动(time driven)。其中一些活动是周期性的，比如调度调度器(scheduler)中的运行队列(runqueue)或者刷新屏幕这样的活动，它们以固有的频率定时发生；同时，内核要非周期性地调度某些函数在未来某个时间发生，比如推迟执行的磁盘I/O操作等。

内核定时器

- 周期性发生的事件都是由系统定时器(system timer)驱动。在X86体系结构上，系统定时器通常是一种可编程硬件芯片(如8254 CMOS芯片)，又称可编程间隔定时器(PIT， Programmable Interval Timer)，其产生的中断就是时钟中断(timer interrupt)。时钟中断对应的处理程序负责更新系统时间和执行周期性运行的任务。系统定时器的频率称为节拍率(tick rate)，在内核中表示为HZ。

- 以X86为例，在2.4之前的内核中其大小为100； 从内核2.6开始，HZ = 1000， 也就是说每秒时钟中断发生1000次。这一变化使得系统定时器的精度(resolution)由10ms提高到1ms，这大大提高了系统对于时间驱动事件调度的精确性。过于频繁的时钟中断不可避免地增加了系统开销(overhead)，但是总的来说，在现在计算机系统上，HZ = 1000不会导致难以接受的系统开销。

内核定时器

- 与系统定时器相对的是动态定时器(dynamic timer)，它是调度事件(执行调度程序)在未来某个时刻发生的时机。内核可以动态地创建或销毁动态定时器。
- 系统定时器及其中断处理程序是内核管理机制的中枢，下面是一些利用系统定时器周期执行的工作(中断处理程序所做的工作):
- (1) 更新系统运行时间(uptime)
- (2) 更新当前墙上时间(wall time)
- (3) 在对称多处理器系统(SMP)上，均衡调度各处理器上的运行队列
- (4) 检查当前进程是否用完了时间片(time slice)，如果用尽，则进行重新调度
- (5) 运行超时的动态定时器
- (6) 更新资源耗尽和处理器时间的统计值

- 内核动态定时器依赖于系统时钟中断，因为只有在系统时钟中断发生后内核才会去检查当前是否有超时的动态定时器。

内核定时器

- HZ和Jiffies
- hz就是一秒钟产生的时间片的数量，也就是时钟发生器产生时钟中断的次数 。系统定时器能以可编程的频率中断处理器。此频率即为每秒的定时器节拍数，对应着内核变量 HZ。选择合适的HZ值需要权衡。HZ值大，定时器间隔时间就小，因此进程调度的准确性会更高。但是，HZ值越大也会导致开销和电源消耗更多，因为更多的处理器周期将被耗费在定时器中断上下文中。HZ的值取决于体系架构。在目前的内核中，可以在编译内核时通过配置菜单选择一个HZ值。该选项的默认值取决于体系架构的版本。
- jiffies变量记录了系统启动以来，系统定时器已经触发的次数。内核每秒钟将jiffies变量增加HZ次。因此，对于HZ值为100的系统，1个jiffy等于10ms，而对于HZ为1000的系统，1个jiffy仅为1ms。
-

内核定时器

- 内核定时器由数据结构timer_list表示，该结构表示了一个待处理的延迟任务，我们称该数据结构为内核定时器节点。

```
struct timer_list {
    struct list_head entry;
    unsigned long expires;
    void (*function)(unsigned long);
    unsigned long data;
    struct timer_base_s *base;
};
```

- 成员变量entry：该内核链表表头类型成员变量用于将该内核定时器节点连接到系统中的定时器链表中。
- 成员变量expires：该无符号长整型变量保存了该定时器的超时时间，用于和系统核心变量jiffies进行比较。
- 成员变量function：该函数指针变量保存了内核定时器超时后要执行的函数，即定时器超时处理函数。
- 成员变量data：该无符号长整型变量用作定时器超时处理函数的参数。
- 成员变量base：该指针变量表明了该内核定时器节点归属于系统中哪一个处理器，在使用函数init_timer()初始化内核定时器节点的过程中，将该指针指向了一个每处理器变量tvec_bases的成员变量t_base。

内核定时器

- ① 首先，使用下面语句声明一个内核定时器数据结构。
- struct timer_list my_timer;
- ② 使用函数init_timer()对上一步声明的内核定时器结构进行初始化。函数init_timer()主要设置该内核定时器归属系统中哪一个处理，并初始化内核定时器链表指针的next域为NULL。
- init_timer(&my_timer);
- ③ 使用下面的语句来设置内核定时器的超时时间expires、超时处理函数function、超时处理函数所使用的参数data。
- my_timer.expires = jiffies + delay;
- my_timer.data = 0;
- my_timer.function = my_function;

内核定时器

- ④ 也是最后一步，通过函数add_timer()来激活内核定时器，使用的语句如下：

- add_timer(&my_timer);

- 通过上面4步，我们就创建了一个内核定时器节点my_timer。该内核定时器在当前时刻以后delay个时钟中断后超时，执行超时处理函数my_function，传给超时处理函数的参数为0。

内核定时器

- 除了上述过程中介绍的内核定时器接口函数之外，内核同时提供了以下接口函数来辅助对内核定时器的操作。

- int mod_timer(struct timer_list *timer，unsigned long expires)：该函数负责修改内核定时器timer的超时字段expires。该函数可以修改激活和没有激活的内核定时器的超时时间，并把它们都设置为激活状态；返回值为0表示修改的内核定时器在修改之前处于未激活状态，返回值为1表示修改的内核定时器在修改之前处于已激活状态。

- int del_timer(structtimer_list* timer)、int del_timer_sync(structtimer_list* timer)：这两个函数负责从链表中删除内核定时器timer。它们的区别在于，后者在多处理器系统中会确保其他处理器上没有处理或者处理完毕当前内核定时器timer时才退出。

内核定时器

- 使用定时器的一般流程为：
- （1）timer、编写function；
- （2）为timer的expires、data、function赋值；
- （3）调用add_timer将timer加入列表；
- （4）在定时器到期时，function被执行；
- （5）在程序中涉及timer控制的地方适当地调用del_timer、mod_timer删除timer或修改timer的expires。

睡眠与唤醒

- 阻塞 I/O

  -- 当我们的驱动在操作中某种条件没有满足,为了提高系统效率,我们需要先放弃cpu时间(睡眠),直到条件得到满足,在应用程序中就表现为阻塞

  -- 对于一个进程"睡眠"意味着它被标识为处于一个特殊的状态并且从调度器的运行队列中去除, 这个进程将不被在任何 CPU 上调度,直到发生某些事情改变了那个状态

- 睡眠规则

  1) 当你运行在原子上下文或中断处理时不能睡眠

  2) 当你醒来,结果是你不能关于你醒后的系统状态做任何的假设, 并且你必须检查来确保你在等待的条件是否得当满足

  3) 是你的进程不能睡眠,除非确信其他人, 在某处的, 将唤醒它

睡眠与唤醒

- 进入睡眠

-- 驱动需要检查filp->f_flags 中的
O_NONBLOCK 标志
-- 进程进入睡眠需要一个等待队列

```
#include <linux/wait.h>
DECLARE_WAIT_QUEUE_HEAD(name);
```

或者动态地初始化

```
wait_queue_head_t my_queue;
init_waitqueue_head(&my_queue);
```

```
wait_event(queue, condition) ;
wait_event_interruptible(queue, condition) ;
wait_event_timeout(queue, condition, timeout) ;
wait_event_interruptible_timeout(queue, condition, timeout);
```

- 唤醒进程

```
void wake_up(wait_queue_head_t *queue);
void wake_up_interruptible(wait_queue_head_t *queue);
```

-- 唤醒可以在任何的程序中进行,包括中断处理


谢谢！
THANKS