# Linux驱动之并发与竞态

|  |
| --- |
| **1. 中断屏蔽:**  **local**\_**irq**\_**disable**() /\*屏蔽中断\*/...  **local**\_**irq**\_**enable**(); /\*打开中断\*/  需要注意的是上述两个函数都只能禁止和使能本CPU内的中断。  这也就是说，这种方法不能解决多CPU(SMP)引发的竞态  **local**\_**irq**\_**save**(flags) //屏蔽中断并且保存目前CPU的中断位信息  **local**\_**irq**\_**restore**(flags) //打开中断，恢复原先CPU中断信息    **local**\_**bh**\_**disable**()  **local**\_**bh**\_**enable**()  /\* 只针对中断底半部的禁止和打开 \*/  **2. 原子操作:**  **整型原子操作:**  **1.设置原子变量的值**  void **atomic**\_**set**(atomic\_t \*v,int i); /\*设置原子变量v的值为i\*/  atomic\_t v = **ATOMIC**\_**INIT**(0); /\*定义原子变量v并初始化为0\*/  **atomic**\_**read**(atomic\_t \*v); /\*返回源自变量的值\*/  void **atomic**\_**add**(int i,atomic\_t \*v); /\*原子变量增加i\*/  void **atomic**\_**sub**(int i,atomic\_t \*v); /\*原子变量减少i\*/  void **atomic**\_**inc**(atomic\_t\*v); /\*原子变量增加一\*/  void **atomic**\_**dec**(atomic\_t\*v); /\*原子变量减少一\*/  int **atomic**\_**inc**\_**and**\_**test**(atomic\_t \*v);  int **atomic**\_**dec**\_**and**\_**test**(atomic\_t\*v);  int **atomic**\_**sub**\_**and**\_**test**(atomic\_t\*v);  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);    **位原子操作**  void **set**\_**bit**(nr,void \*addr); /\*设置addr地址的第nr位为1\*/  void **clear**\_**bit**(nr,void \*addr); /\*设置addr地址的第nr位为0\*/  void **change**\_**bit**(nr,void \*addr); /\*对addr地址的第nr位进行反置\*/  **test**\_**bit**(nr,void \*addr); /\*返回addr地址的第nr位\*/  int **test**\_**and**\_**set**\_**bit**(nr,void \*addr);  int **test**\_**and**\_**clear**\_**bit**(nr,void \*addr);  int **test**\_**and**\_**change**\_**bit**(nr,void \*addr);  **3. 自旋锁：**  **spinlock**\_t lock; //自定义自旋锁  **spin**\_**lock**\_**init**(lock); //初始化自旋锁: 动态初始化自旋锁lock。  **spin**\_**lock**(lock) //获得自旋锁：如果能够获得锁，它就马上返回，否则他将自选在那里，直到该自旋锁的保持者释放。  **spin**\_**trylock**(lock) //尝试获取自旋锁：如果能获得自旋锁Lock就立即获得锁，并返回true，否则返回false，而并不原地等待  **spin**\_**unlock**(lock) //释放自旋锁  **中断自旋锁：**  spin\_**lock**\_**irq**() = spin\_lock()+local\_irq\_disable()  spin\_**unlock**\_irq() = spin\_unlock()+local\_irq\_enable()  spin\_**lock**\_**irqsave**() = spin\_lock()+local\_irq\_save()  spin\_**unlock**\_irqrestore() = spin\_unlock()+local\_irq\_restore()  **读写自旋锁：**  rwlock\_t my\_rwlock = **PW**\_**LOCK**\_**UNLOCKED**; /\*静态初始化\*/  **rwlock**\_**init**(&my\_rwlock); //动态初始化  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);  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);  在对共享资源进行读取之前，应该先调用读读锁定函数，完成之后调用解锁函数  void **write**\_**lock**(rwlock\_t \*lock);  void **write**\_**lock**\_**irwqsave**(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); //尝试获取自旋锁，不管成功与否都立即返回  void **write**\_**unlock**(rwlock\_t \*lock);  void **write**\_**unlock**\_**irwqsave**(rwlock\_t \*lock,unsigned long flags);  void **write**\_**unlock**\_**irq**(rwlock\_t \*lock);  **顺序锁操作:**  void **write**\_**seqlock**(seqlock\_t sl);  int **write**\_**tryseqlock**(seqlock\_t \*sl);  **write**\_**seqlock**\_**irqsave**(lock,flags); //= write\_seqlock()+local\_irq\_save()  **write**\_**seqlock**\_**irq**(lock); //spin\_unlock()+local\_irq\_enable()  **write**\_**seqlock**\_**bh**(lock); //spin\_unlock()+local\_bh\_enable()  /\*读执行单玉环在对被顺序锁sl保护的共享资源进行访问前，需要调用该函数  ，该函数返回顺序锁sl的当前序号\*/  unsigned **read**\_**seqbegin**(const seqlock\_t \*sl);  **read**\_**seqbegin**\_**irqsave**(lock,flag);  /\*读执行单元在访问玩被顺序锁sl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操作，如果有写操作，读执行单元就要重新进行读取\*/  int **read**\_**seqretry**(const seqlock\_t \*sl,unsigned iv);  **read**\_**seqretry**\_**irqrestore**(lock,iv,flags);  **4.信号量**  struct **semaphore** sem;  void **sema**\_**init**(struct semaphore \*sem,int val); //设置信号量的值为val  #define init\_**MUTEX**(sem) sem\_init(sem,1); //定义并初始化信号量为1  #define init\_**MUTEX**\_**LOCKED**(sem) sema\_init(sem,0); //定义并初始化信号量为0  **DECLARE**\_**MUTEX**(name); //定义一个名为name的信号量并初始化为1,  **DECLARE**\_**MUTEX**\_**LOCKED**(name); //定义一个名为name的信号量并初始化为0    void **down**(struct semaphore \* sem);  该函数用于获得信号量，如果获取不到会导致睡眠，因此不能在中断上下文中使用  int **down**\_**interruptible**(struct semaphore \* sem);  与down类似，不同之处在于，由此函数引起的睡眠可以被信号打断，信号也会导致该函数返回非零值  int **down**\_**trylock**(struct semaphore \* sem);  尝试获取信号量，如果能够立即获得，它就获得信号量并返回0，否则返回非零值，不会导致睡眠  void **up**(struct semaphore \* sem); //释放信号量唤醒等待者  **5. 完成量**  struct **completion** my\_completion;  **init**\_**completion**(&my\_completion);  **DECLARE**\_**COMPLETION**(my\_completion);  void **wait**\_**for**\_**completion**(struct completion \*c);  void **complete**(struct completion \*c);  void **complete**\_**all**(struct completion \*c);  **6. 互斥体**  struct **mutex** my\_mutex;  **mutex**\_**init**(&my\_mutex);  void inline \_\_sched **mutex**\_**lock**(struct mutex \*lock);  int \_\_sched **mutex**\_**lock**\_**interruptible**(struct mutex \*lock);  int \_\_sched **mutex**\_**trylock**(struct mutex \*lock);  void \_\_sched **mutex**\_**unlock**(struct mutex \*lock); |

## 各种同步机制的比较

|  |  |  |
| --- | --- | --- |
| 类型 | 机制 | 应用场合 |
| spinlock | 使用忙等方法，进程不挂起 | (1)用于多处理器间共享数据 (2)在可抢占的内核线程里共享数据 (3)自旋锁适合于保持时间非常短的情况，它可以在任何上下文使用，比如中断上下文 |
| 信号量 | 阻塞式等待，进程挂起 | (1)适合于共享区保持时间教长的情况 (2)只能用于进程上下文 |
| 原子操作 | 数据的原子访问 | (1)共享简单的数据类型：整型，比特性 (2)适合高效率的场合 |
| rwlock | 特殊的自旋锁 | (1)允许同时读共享资源，但只能有一个写 (2)读优先于写，读写不能同时 |
| 顺序锁 | 一种免锁机制，基于访问计数 | (1)允许同时读共享资源，但只能有一个写 (2)写优先于读，读写不能同时 |
| RCU | 通过副本的免锁访问 | (1)对读占主要的场合提供高性能 (2)读访问不必获取锁，不必执行原子操作或禁止中断 |
| 关闭中断 | 通过禁止中断的手段，排除单处理器上的并发，会导致中断延迟 | (1)中断与正常进程共享数据 (2)多个中断共享数据 (3)临界区一般很短 |

## 并发控制与竞态:

**并发：**

并发是指多个单元**同时、并行被执行**，而并发执行单元对**共享资源的访问**很容易导致竞态。

**竞态：**

假设有一个设备，执行单元A对其写入3000个字符'a'而另一个执行单元B对其写入4000个'b'，第三个执行单元C读取globalmem的所有字符，**如果执行单元A、B对于设备的写入操作同时发生，此时就会造成竞态**

## 竞争状态的分类:

**1.对称多处理器(SMP)的多个CPU（多核CPU同时对共享资源的访问）**

SMP是一种紧耦合、共享存储的系统类型，因为多个CPU同时共享系统总线，因此可以访问共同的外设和存储器。

**2.单CPU内进程与抢占它的进程（单核CPU的进程调度对共享资源的访问）**

Linux2.6支持内核抢占调度,一个进程在内核执行的时候可能被另一个高优先级的进程打断，进程与抢占它的进程访问共享资源时会发生竞争。

**3.中断(硬中断、软中断、Tasklet、底半部)与进程之间**

中断可以打断重在执行的进程、如果中断处理程序访问进程正在访问的资源、此时静态会发生。此外中断被更高级的中断打断时也会发生

## 竞态的解决方法:

保证对共享资源的互斥访问(即一个执行单元在访问共享资源时，其他的执行单元被禁止访问)

访问**共享资源的代码区域被称为临界区**(critical sections),临界区需要被以某种互斥机制加以保护。

Linux常见互斥机制:**中断屏蔽**、**原子操作**、**自旋锁**和**信号量**等是Linux设备驱动中科采用的互斥路径

## 中断屏蔽:

我们知道，中断会引起竞态。所以在进入临界区之前屏蔽系统中断，这样就不必在担心在进入临界区后被中断打断。

实质:中断屏蔽将使进程之间的并发不再发生，而且，由于Linux内核的进程调度等操作都依赖于中断，所以屏蔽中断后其实就相当于避免了进程之间的并发

|  |
| --- |
| **法一：**  **local**\_**irq**\_**disable**() /\*屏蔽中断\*/  ...  critical section /\*临界区\*/  ...  **local**\_**irq**\_**enable**(); /\*打开中断\*/  需要注意的是上述两个函数都只能禁止和使能本CPU内的中断。  这也就是说，这种方法不能解决多CPU(SMP)引发的竞态  **法二：**  **local**\_**irq**\_**save**(flags) //屏蔽中断并且保存目前CPU的中断位信息  ...  critical section /\*临界区\*/  ...  **local**\_**irq**\_**restore**(flags) //打开中断，恢复原先CPU中断信息    **法三：**  /\* 只针对中断底半部的禁止和打开 \*/  **local**\_**bh**\_**disable**()    **local**\_**bh**\_**enable**()  **Linux中断的底半部机制:**  Linux系统把中断处理分为定半部和底半部。Linux上半部只处理中断框架流程和紧急事件，例如对硬件设备的访问、或者修改由设备和处理器同时访问的数据，以及修改哪些只有处理器才可以访问的数据结构. |

## 原子操作:

性质: 原子操作是指在执行过程中不会被代码路径所中断的操作。

作用: 从它的性质上我们就可以推算出它的功能:主要用作内核计数

分类: 位变量原子操作和整型变量原子操作

|  |
| --- |
| **整型原子操作:**  **1.设置原子变量的值**  void **atomic**\_**set**(atomic\_t \*v,int i); /\*设置原子变量v的值为i\*/  atomic\_t\_v = **ATOMIC**\_**INIT**(0); /\*定义原子变量v并初始化为0\*/  **2.获取原子变量的值**  **atomic**\_**read**(atomic\_t \*v); /\*返回源自变量的值\*/  **3.原子变量的加减**  void **atomic**\_**add**(int i,atomic\_t \*v); /\*原子变量增加i\*/  void **atomic**\_**sub**(int i,atomic\_t \*v); /\*原子变量减少i\*/  **4.原子变量自增/自减**  void **atomic**\_**inc**(atomic\_t\*v); /\*原子变量增加一\*/  void **atomic**\_**dec**(atomic\_t\*v); /\*原子变量减少一\*/  **5.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**(atomic\_t\*v);  **6.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);    **位原子操作**  **1.设置位**  void **set**\_**bit**(nr,void \*addr); /\*设置addr地址的第nr位为1\*/  **2.清除位**  void **clear**\_**bit**(nr,void \*addr); /\*设置addr地址的第nr位为0\*/  **3.改变位**  void **change**\_**bit**(nr,void \*addr); /\*对addr地址的第nr位进行反置\*/  **4.测试位**  **test**\_**bit**(nr,void \*addr); /\*返回addr地址的第nr位\*/  **5.测试并操作位**  int **test**\_**and**\_**set**\_**bit**(nr,void \*addr);  int **test**\_**and**\_**clear**\_**bit**(nr,void \*addr);  int **test**\_**and**\_**change**\_**bit**(nr,void \*addr); |

下面就演示一个实例来利用原子变量实现设备在同一时刻只能由一个进程的打开

|  |
| --- |
| static atomic\_t xxx\_available = **ATOMIC\_INIT**(1); /\*定义并初始化原子变量为1\*/    static int xxx\_**open**(struct inode\*inode,struct file \*filp)  {  ...  if(!**atomic**\_**dec**\_**and**\_**test**(&xxx\_available)){//如果该原子减1后不为0，说明该设备已经被以进程打开  **atomic**\_**inc**(&xxx\_available);//将刚才减去的一加上  return -EBUSY;/\*已经打开\*/  }  ...  return 0;/\*成功\*/  }    static int xxx\_**release**(struct inode\*inode,struct file \*filp)  {  **atomic**\_**inc**(&xxx\_available);/\*释放设备\*/  return 0;  } |

## 自旋锁:

前面介绍过，要防止静态现象的发生，就要使线程对临界区对象进行互斥访问；

**自旋锁定义**: 自旋锁是一种典型的对临界区对象进行互斥访问的手段。

**机制简介**: 我们在**进入临界区之前首先要进行一个原子操作**(测试原子变量)，**如果测试结果表明资源已经没有被占用(即锁锁空闲)，则程序获取这个自旋锁(相当于该程序在进入临界区后将进入该临界区的大门关闭) ，否则程序将不停的在原地重复测试，**这也就是为什么叫它“自旋锁”。

### 普通自旋锁

|  |
| --- |
| **spinlock**\_t lock; //自定义自旋锁  **spin**\_**lock**\_**init**(lock); //初始化自旋锁: 动态初始化自旋锁lock。  **spin**\_**lock**(lock) //获得自旋锁：如果能够获得锁，它就马上返回，否则他将自选在那里，直到该自旋锁的保持者释放。  **spin**\_**trylock**(lock) //尝试获取自旋锁：如果能获得自旋锁Lock就立即获得锁，并返回true，否则返回false，而并不原地等待  **spin**\_**unlock**(lock) //释放自旋锁 |

**Sample：自旋锁的使用一：**

|  |
| --- |
| **spinlock**\_t lock; /\* 定义自旋锁 \*/  spin\_lock\_**init**(&lock); /\* 初始化自旋锁 \*/  spin\_**lock**(&lock); /\*等待直到获取自旋锁，保护临界区\*/  ... /\*临界区\*/  spin\_**unlock**(&lock); /\*解锁\*/ |

**Sample：自旋锁的使用二：**

|  |
| --- |
| int **xxx\_count** = 0;  static int xxx\_**open**(struct inode\*inode,struct file\*filp)  {  spin\_**lock**(&xxx\_lock);  if(**xxx\_count**){ /\*已经打开\*/  spin\_**unlock**(&xxx\_lock);  return -EBUSY;  }  **xxx\_count++**; /\*增加使用计数\*/  spin\_**unlock**(&xxx\_lock);  ...  return 0 ; /\*成功\*/  }    static int xxx\_**release**(struct inode\*inode,struct file \*filp)  {  ...  **Spin\_lock**(&xxx\_lock);  **xxx\_count--;**  /\*减少使用计数\*/  spin\_**unlock**(&xxx\_lock);  return 0;  } |

一般应用范围:主要针对SMP或单CPU(内核抢占)的情况。对于单CPU和内核不支持抢占的系统，自旋锁退化为空操作;

注意普通的**自旋锁只能保证临界区不受别的CPU和本CPU和本CPU内的抢占进程打扰，但是得到锁的代码路径在执行临界区的时候还可能受到中断和底半部( BH)的影响**。为了解决这个问题就需要在spin\_lock(&lock)和unspin\_lock(&lock)上衍生出新的自旋锁机制

### 中断自旋锁

|  |
| --- |
| **整套的自旋锁机制:**  spin\_**lock**\_**irq**() = spin\_lock()+local\_irq\_disable()  spin\_**unlock**\_irq() = spin\_unlock()+local\_irq\_enable()  spin\_**lock**\_**irqsave**() = spin\_lock()+local\_irq\_save()  spin\_**unlock**\_irqrestore() = spin\_unlock()+local\_irq\_restore() |

自旋锁所带来的问题:

1.自旋锁是忙等待，当锁不可使用的时候,CPU将会不停的测试该自旋锁直到该锁可以使用。而 **CPU在等待自旋锁时不会做任何有用的工作，因此只有在占用所的时间很少的情况下使用它才是合理的 如果临界区过大，需要长时间占用锁，这样就降低了系统性能**

2.可**能导致死锁，例如:当某个CPU已经拥有了自旋锁A，但是它又一次的想要获得这个A，这是就导致了死锁**

3.自**旋锁锁定期间不能再调用那些可能引起进程调度的函数**。

例如:一进程再获取自旋锁之后，因为调用这些函数copy\_xx\_user()\kmalloc()\msleep()则可能导致内核崩溃

### 读写自旋锁:

普通的自旋锁不关心锁定的内核对象，究竟进行怎样的操作，不管是读还是写他都一视同仁。

然后就衍生出一种新的自旋锁机制--读写自旋锁

定义:可以允许读的并发，最多只能由一个写

|  |
| --- |
| **操作:**  **1.定义读写自旋锁:**  rwlock\_t my\_rwlock = **PW**\_**LOCK**\_**UNLOCKED**; /\*静态初始化\*/    rwlock\_t my\_rwlock;  **rwlock**\_**init**(&my\_rwlock); //动态初始化  **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);  在对共享资源进行读取之前，应该先调用读读锁定函数，完成之后调用解锁函数  **4.写锁定函数**  void **write**\_**lock**(rwlock\_t \*lock);  void **write**\_**lock**\_**irwqsave**(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); //尝试获取自旋锁，不管成功与否都立即返回  **5.写解锁**  void **write**\_**unlock**(rwlock\_t \*lock);  void **write**\_**unlock**\_**irwqsave**(rwlock\_t \*lock,unsigned long flags);  void **write**\_**unlock**\_**irq**(rwlock\_t \*lock); |

Sample : 读写自旋锁使用范例:

|  |
| --- |
| rwlock\_t lock; /\*定义rwlock\*/  rwlcok\_init(&lock); /\*初始化rwlock\*/      read\_lock(&lock); /\*读时获取锁\*/  ... /\*临界资源\*/  read\_unlock(&lock);    write\_lock\_irqsave(&lock,flags); /\*写时获取锁\*/  .. ./\*临界资源\*/  write\_unlock\_irqstore(&lock,flags); |

## 顺序锁:

**机制 :** **读时可写或者写时可读，但是不可同时写**，**并且如果在读执行单元进行读操作时有写执行单元写入，那么读执行单元需要重新读取，以确保数据最新**

**限制:** 顺序锁要求被保护的资源不包含有指针

|  |
| --- |
| **顺序锁操作:**  **1.获取顺序锁**  void **write**\_**seqlock**(seqlock\_t sl);  int **write**\_**tryseqlock**(seqlock\_t \*sl);  **write**\_**seqlock**\_**irqsave**(lock,flags); //= write\_seqlock()+local\_irq\_save()  **write**\_**seqlock**\_**irq**(lock); //spin\_unlock()+local\_irq\_enable()  **write**\_**seqlock**\_**bh**(lock); //spin\_unlock()+local\_bh\_enable()    **2.释放自旋锁，与获取自选锁相对**  /\*读执行单元涉及的循序操作\*/  **1.读开始**  /\*读执行单玉环在对被顺序锁sl保护的共享资源进行访问前，需要调用该函数  ，该函数返回顺序锁sl的当前序号\*/  unsigned **read**\_**seqbegin**(const seqlock\_t \*sl);  **read**\_**seqbegin**\_**irqsave**(lock,flag);  **2.重读**  /\*读执行单元在访问玩被顺序锁sl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操作，如果有写操作，读执行单元就要重新进行读取\*/  int **read**\_**seqretry**(const seqlock\_t \*sl,unsigned iv);  **read**\_**seqretry**\_**irqrestore**(lock,iv,flags);  /\*读执行单元使用模式\*/  do{  segnum = **read**\_**seqbegin**(&seqlock\_a,seqnum);  /\*读操作代码\*/  ...  }while(**read**\_**seqretry**(&seqlock\_a,seqnum)); |

## 自旋锁的衍生----读-拷贝-更新(RCU)

**特点:**

1.对于RCU保护的共享数据，读执行单元不需要任何锁就可以访问它，不实用原子指令

2.用RCU的写执行单元在访问它前需要首先拷贝一个副本，然后对副本进行修改，最后在时机适当的时候指向原来数据的指针，重新指向新的被修改的数据

**缺点:** 如果写操作比较多时，对读执行单元的性能的提高不能弥补写执行单元导致的损失。原因在于使用RCU时写执行单元之间的同步开销会比较大

|  |
| --- |
| **RCU操作:**  **1.读锁定**  **rcu**\_read\_**lock**()  **rcu**\_read\_lock\_**bh**()  **2.读解锁**  rcu\_read\_**unlock**()  rcu\_read\_**unlock**\_**bh**()  **3.同步RCU**  **synchronize**\_rcu()  此操作将阻塞写执行单元，知道所有的读执行单元已经完成读操作，写指定单元才进行下一步操作    **synchronize**\_**sched**()  该函数用于等待所有的CPU都处在可抢占的状态，它能保证正在运行的中断处理函数处理完毕，单不能保证正在运行的软中断处理完毕    **使用模式:**  rcu\_read\_lock(); //获取锁  ... /\*读临界区\*/  rcu\_read\_unlock() |

## 信号量

信号量是保护临界区的另一种有效方法

机制:只有得到信号量的进程才能执行临界区代码，但是与自旋锁不同的是 ，当获取不到信号量时，进程不会在原地打转而是进入休眠状态，这种休眠可以被唤醒

|  |
| --- |
| **1.定义信号量**  struct **semaphore** sem;  **2.初始化信号量**  void **sema**\_**init**(struct semaphore \*sem,int val); //设置信号量的值为val    **相关的宏**  #define init\_**MUTEX**(sem) sem\_init(sem,1); //定义并初始化信号量为1  #define init\_**MUTEX**\_**LOCKED**(sem) sema\_init(sem,0); //定义并初始化信号量为0    **DECLARE**\_**MUTEX**(name); //定义一个名为name的信号量并初始化为1,  **DECLARE**\_**MUTEX**\_**LOCKED**(name); //定义一个名为name的信号量并初始化为0    **3.获得信号量**  void **down**(struct semaphore \* sem);  该函数用于获得信号量，如果获取不到会导致睡眠，因此不能在中断上下文中使用  int **down**\_**interruptible**(struct semaphore \* sem);  与down类似，不同之处在于，由此函数引起的睡眠可以被信号打断，信号也会导致该函数返回非零值  int **down**\_**trylock**(struct semaphore \* sem);  尝试获取信号量，如果能够立即获得，它就获得信号量并返回0，否则返回非零值，不会导致睡眠    注意:使用down\_interrupible()获取信号量的时候，对返回值一般会进行检查，如果非零返回-ERESTARTSYS  if(down\_interrupible(&sem))  return -ERESTARTSYS;    **4.释放信号量**  void **up**(struct semaphore \* sem);  释放信号量唤醒等待者    /\*信号量的一般使用步骤\*/  DECLARE\_MUTEX(mount\_sem); //定义并初始化信号量为1  down(&mount\_sem);  ...  critical section /\*临界区\*/  ...  up(&mount\_sem);/\*释放信号量\*/ |

**实例:用信号量实现，设备只能被一个进程打开的实例**

|  |
| --- |
| static **DELCARE**\_**MUTEX**(xxx\_lock);/\*定义互斥量\*/  static int xxx\_**open**(struct inode\*inode ,struct file\*filp)  {  if(**down**\_**trylock**(&xxx\_lock)) /\*获得信号量,尝试获得\*/  return -EBUSY; /\*设备忙\*/  return 0;  }    static int xxx\_**release**(struct inode\*inode,struct file\*filp)  {  **up**(&xxx\_lock);  return 0;  } |

小知识:如果信号量初始化为零，则可以用于同步

## 完成量

简介:与信号量类似，但是同步机制更好

|  |
| --- |
| **1.定义完成量**  struct **completion** my\_completion;    **2.初始化completion**  (1),**init**\_**completion**(&my\_completion);  (2),**DECLARE**\_**COMPLETION**(my\_completion);    **3.等待完成量**  void **wait**\_**for**\_**completion**(struct completion \*c);  **4.唤醒完成量**  void **complete**(struct completion \*c);  void **complete**\_**all**(struct completion \*c); |

**自选量和信号量的使用原则：**

(1)自旋锁保护的临界区中不能包含会引起阻塞的代码(可能会引起死锁啦)

(2)如果被保护的共享资源需要在中断上下文中使用则只能使用自旋量或者是down\_trylock()

## 互斥体

简介:我们在定义信号量的时候用到了如下宏:

DECLARE\_MUTEX()，从名字中我们可以看出"mutex"互斥的概念，但是这种互斥

机制在Linux内核中其实已经被实现，从某种意义上来说，信号量有点"模仿"的意思

|  |
| --- |
| **1.定义&初始化**  struct **mutex** my\_mutex;  **mutex**\_**init**(&my\_mutex);    **2.等待(获取)互斥体**  void inline \_\_sched **mutex**\_**lock**(struct mutex \*lock);  int \_\_sched **mutex**\_**lock**\_**interruptible**(struct mutex \*lock);  int \_\_sched **mutex**\_**trylock**(struct mutex \*lock);  **3.释放互斥体**  void \_\_sched **mutex**\_**unlock**(struct mutex \*lock); |

# Linux驱动之阻塞/非阻塞

|  |
| --- |
| **等待队列：**  1、等待队列头的定义及初始化：  struct **wait**\_**queue**\_**head\_t** mywaitqueue; //定义  init\_**waitqueue**\_**head**(&mywaitqueue); //初始化  **DECLARE**\_**WAIT**\_**QUEUE**\_**HEAD**(mywaitqueue); //定义+初始化  2、等待队列的定义及初始化。  **DECLARE**\_**WAITQUEUE**(**name**, **tsk**) //定义一个等待队列的节点  3. 添加/移除等待队列  **add**\_**wait**\_**queue**(wait\_queue\_head\_t \*q,wait\_queue\_t \*wait); //添加到等待队列头中。  **remove**\_**wait**\_**queue**(wait\_queue\_head\_t \*q,wait\_queue\_t \*wait); //移除等待节点  4、等待事件  **wait**\_**event**(queue,condition);  **wait**\_**event**\_**interruptible**(queue,condition);  **wait**\_**event**\_**timeout**(wq, condition, timeout)  **wait**\_**event**\_**interruptible**\_**timeout**(wq, condition, timeout)  等待第一个参数queue作为等待队列头的等待队列被唤醒，而且第二个参数condition必须满足，否则阻塞。wait\_event()和wait\_event\_interruptible()的区别在于后者可以被信号打断，而前者不能。加上timeout后的宏意味着阻塞等待的超时时间，以jiffy为单位，在第三个参数的timeout到达时，不论condition是否满足，均返回。  5、唤醒队列  **wait**\_**up**(wait\_queue\_head\_t \*q), //唤醒该等待队列头对应的所有等待。  **wait**\_**up**\_**interruptible**(wait\_queue\_head\_t \*q) //唤醒处于TASK\_INTERRUPTIBLE的等待进程。  1. wait\_up()应该与 wait\_event(), wait\_event\_timeout()成对使用；  2. wait\_up\_interruptible()应该与 wait\_event\_interruptible()， wait\_event\_interruptible\_timeout()成对使用。  3. wake\_up()可以唤醒处于TASK\_INTERRUPTIBLE和TASK\_UNINTERRUPTIBLE的进程  4. wake\_up\_interruptble()只能唤醒处于TASK\_INTERRUPTIBLE的进程。 |

**阻塞：** 进程在执行设备操作时若不能获得资源则挂起操作，直到满足可操作的条件后再进行操作，被挂起的进程进入休眠状态，被从调度器的运行队列移走，直到等待的条件满足。

**非阻塞：** 进程在不能进行设备操作时并不挂起，它或者放弃，或者不停的查询，直到可以进行位置。

|  |  |
| --- | --- |
| **阻塞地取串口一个字符** | **非阻塞地取串口一个字符** |
| char buf;  fd = open("/dev/ttys",O\_RDWR);  //当串口上有输入时才返回  res = read(fd,&buf,1);  if(res == 1)  printf("%c\n",buf); | char buf;  fd = open("/dev/ttys",O\_RDWR | **O\_NONBLOCK**);  //当串口上无输入也返回,所以要循环尝试读取串口  while( read(fd,&buf,1) !=1);  printf("%c\n",buf); |

## 等待队列

Linux驱动程序中可以使用 等待队列(wait queue) 实现阻塞进程的唤醒。

**Sample :**

1、在设备结构体中添加等待队列头，由于读写都需要堵塞，所以添加两个队列头，分别用来堵塞写操作，写操作。

2、然后在模块初始化中初始化队列头

3、确定一个具体的条件，比如数据有无，具体的条件根据实际的情况设计。

4、在需要堵塞的读函数，写函数中分别实现堵塞，首先定义等待队列的节点，并添加到队列中去，然后等待事件的唤醒进程。但是由于读写操作的两个等待队列都是基于条件havedata的，所以在读完成以后需要唤醒写，写完成以后需要唤醒读操作，同时更新条件havedata，最后还要移除添加的等待队列节点。

5、应用程序采用两个不同的进程分别进行读、写，然后检测顺序是否可以调换，检查等待是否正常。

|  |
| --- |
| #include<linux/wait.h>  struct mem\_dev  {  char \*data;  unsigned long size;  spinlock\_t lock; /\*添加一个并行机制\*/  Step 1. 定义等待队列头  **wait\_queue\_head\_t rdqueue;**  /\*定义读等待队列头\*/  **wait\_queue\_head\_t wrqueue;** /\*定义写等待队列头\*/  };  static bool havedata = false; /\*等待条件\*/ |
| /\*初始化函数\*/  static int memdev\_init(void)  {  for(i = 0; i < MEMDEV\_NR\_DEVS; i)  {  /\*对设备的数据空间分配空间\*/  mem\_devp[i].size = MEMDEV\_SIZE;  mem\_devp[i].data = kmalloc(MEMDEV\_SIZE,GFP\_KERNEL);  memset(mem\_devp[i].data,0,MEMDEV\_SIZE);  spin\_lock\_init(&(mem\_devp[i].lock)); //初始化自旋锁  Step 2. 初始化等待队列头  **init\_waitqueue\_head(&(mem\_devp[i].rdqueue));**  **init\_waitqueue\_head(&(mem\_devp[i].wrqueue));**  }  } |
| /\*read函数的实现\*/  static ssize\_t mem\_read(struct file \*filp,char \_\_user \*buf, size\_t size,loff\_t \*ppos)  {  unsigned long p = \*ppos;  unsigned int count = size;  struct mem\_dev \*dev = filp->private\_data;  if(p >= MEMDEV\_SIZE) /\*参数的检查，首先判断文件位置\*/  return 0;  if(count > MEMDEV\_SIZE - p) /\*改正文件大小\*/  count = MEMDEV\_SIZE - p;  Step 3. 定义等待队列节点wait\_r  **DECLARE**\_**WAITQUEUE**(**wait\_r**,current);  Step 4. 将等待队列节点wait\_r加到等待队列头requeue中  **add**\_**wait**\_**queue**(&dev->**rdqueue**,&**wait\_r**);  while(!havedata)  {  /\*判断用户是否设置为非堵塞模式读,告诉用户再读\*/  if(filp->f\_flags & O\_NONBLOCK)  return -EAGAIN;  Step 5. 等待havedata == true && 调用wake\_up\_interruptible(rdqueue)唤醒程序  **wait**\_**event**\_**interruptible**(dev->**rdqueue**,**havedata**);  }  spin\_lock(&dev->lock);  if(copy\_to\_user(buf,(void \*)(dev->data p),count)) { /\*从内核读数据到用户空间\*/  ret = -EFAULT; /\*出错误\*/  }  else {  \*ppos = count; /\*移动当前文件光标的位置\*/  ret = count;  }  spin\_unlock(&dev->lock);  Step 6. 删除等点队列节点  **remove**\_**wait**\_**queue**(&dev->**rdqueue**,&**wait\_r)**;  Step 7. 设置条件变量havedata=false， 调用wake\_up\_interruptible(wrqueue)唤醒write()函数  havedata = false;  **wake**\_**up**\_**interruptible**(&dev->wrqueue); /\*唤醒写等待队列\*/  return ret;  } |
| /\*write函数的实现\*/  static ssize\_t mem\_write(struct file \*filp,const char \_\_user \*buf,size\_t size,loff\_t \*ppos)  {  unsigned long p = \*ppos;  unsigned int count = size;  struct mem\_dev \*dev = filp->private\_data;  if(p >= MEMDEV\_SIZE) /\*参数的检查，首先判断文件位置\*/  return 0;  if(count > MEMDEV\_SIZE - p) /\*改正文件大小\*/  count = MEMDEV\_SIZE - p;  **DECLARE**\_**WAITQUEUE**(**wait\_w,**current); /\*定义并初始化一个等待队列节点 \*/  **add**\_**wait**\_**queue**(&dev->**wrqueue**,&**wait\_w**); /\*将节点添加到等待队列中\*/  /\*采用循环是为了防止信号等其他原因导致唤醒\*/  while(havedata)  {  if(filp->f\_flags & O\_NONBLOCK) /\*如果是以非堵塞方式\*/  return -EAGAIN;    /\*分析源码发现，wait\_event\_interruptible 中存在DECLARE\_WAITQUEUE和add\_wait\_queue的操作，因此不需要手动添加等待队列节点\*/  **wait**\_**event**\_**interruptible**(&dev->**wrqueue**,(!**havedata**));  }  spin\_lock(&dev->lock);  if(copy\_from\_user(dev->data p,buf,count))  ret = -EFAULT;  else {  \*ppos = count; /\*改变文件位置\*/  ret = count;  }  spin\_unlock(&dev->lock);  **remove\_wait\_queue(**&dev->**wrqueue**,&**wait\_w**); /\*将该等待节点移除\*/  havedata = true; /\*更新条件\*/  **wake\_up\_interruptible**(&dev->**rdqueue**); /\*唤醒读等待队列\*/  return ret;  } |

## 轮询操作 : poll() – 内核空间

**unsigned int (\*poll)(struct file \*filp, struct poll\_table \*wait);** //轮询函数

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| poll()函数的关键内容:  1. 把当前进程添加到wait参数指定的等待列表(poll\_table)中  **void poll\_wait(struct file \*filp, wait\_queue\_head\_t \*queue, poll\_table \*wait);**  //需要注意的是这个函数是不会引起阻塞的。  2. 返回表示是否能对设备进行无阻塞读写访问的掩码   |  |  | | --- | --- | | **常量** | **说明** | | POLLIN | 普通或优先级带数据可读 | | POLLRDNORM | 普通数据可读 | | POLLRDBAND | 优先级带数据可读 | | POLLPRI | 高优先级数据可读 | | POLLOUT | 普通数据可写 | | POLLWRNORM | 普通数据可写 | | POLLWRBAND | 优先级带数据可写 | | POLLERR | 发生错误 | | POLLHUP | 发生挂起 | | POLLNVAL | 描述字不是一个打开的文件 | |
| **Sample :**  static unsigned int XXX\_poll(struct file \*filp, poll\_table \*wait)  {  unsigned int mask = 0;  struct XXX\_dev \*dev = filp->private\_data; //获得设备结构指针  ...  poll\_wait(filp, &dev->r\_wait, wait); //加读等待对列头  poll\_wait(filp ,&dev->w\_wait, wait); //加写等待队列头    if(...)//可读  ｛  mask |= POLLIN | POLLRDNORM; //标识数据可获得  ｝  if(...)//可写  ｛  mask |= POLLOUT | POLLRDNORM; //标识数据可写入  ｝  ..  return mask;  } |

## 轮询操作 : select() – 用户空间

|  |
| --- |
| **int select(int numfds, fd\_set \*readfds, fd\_set \*writefds, fd\_set \*exceptionfds, struct timeval \*timeout);**  readfs，writefds，exceptfds分别是**读，写和异常处理**的文件描述符集合。  numfds的值是需要检查的号码最高的文件描述符加1。  timeout则是一个时间上限值，超过该值后，即使仍没有描述符准备好也会返回。  涉及到文件描述符集合的操作主要有以下几种:  1)清除一个文件描述符集 FD\_ZERO(fd\_set \*set);  2)将一个文件描述符加入文件描述符集中 FD\_SET(int fd,fd\_set \*set);  3)将一个文件描述符从文件描述符集中清除 FD\_CLR(int fd,fd\_set \*set);  4)判断文件描述符是否被置位 FD\_ISSET(int fd,fd\_set \*set); |
| **Sample :**  #define FIFO\_CLEAR 0x1  #define BUFFER\_LEN 20  main()  {  int fd, num;  char rd\_ch[BUFFER\_LEN];  fd\_set rfds,wfds;  /\*以非阻塞方式打开/dev/polltest设备文件\*/  fd = **open**("/dev/polltest", O\_RDONLY | **O\_NONBLOCK**);  if (fd != - 1)  { /\*FIFO清0\*/  if (ioctl(fd, FIFO\_CLEAR, 0) < 0)  {  printf("ioctl command failed\n");  }  while (1)  {  FD\_ZERO(&rfds);  FD\_ZERO(&wfds);  FD\_SET(**fd**, &rfds);  FD\_SET(**fd**, &wfds);  **select**(fd + 1, &rfds, &wfds, NULL, NULL);  /\*数据可获得\*/  if (**FD**\_**ISSET**(**fd**, &**rfds**))  {  printf("Device can be read now\n");  }  /\*数据可写入\*/  if (FD\_ISSET(fd, &**wfds**))  {  printf("Device can be written now\n");  }  }  }  else  {  printf("Device open failure now\n");  }  } |

# Linux驱动之异步通知异步I/O

|  |
| --- |
| **int fasync\_helper(int fd, struct file \*filp, int mode ,struct fasync\_struct \*\*fa);**  //初始化fasync\_struct结构体;  //fd= -1， mode=0 , 则释放fasync\_struct结构体  **void kill\_fasync(struct fasync\_struct \*\*fa, int sig, int band);** //发出SIGIO信号  **用户层函数接口：**  1）**signal(SIGIO, sig\_handler);**  调用signal函数，让指定的信号SIGIO与处理函数sig\_handler对应。  2）**fcntl(fd, F\_SET\_OWNER, getpid());**  指定一个进程作为文件的“属主(filp->owner)”，这样内核才知道信号要发给哪个进程。  3）设置文件标志，添加FASYNC标志  **f\_flags = fcntl(fd, F\_GETFL);**  **fcntl(fd, F\_SETFL, f\_flags | FASYNC);** |

## 异步通知： kill\_fasync () – 内核空间

Fasync机制就是为了使驱动的读写和application的读写分开，使得application可以在驱动读写时去做别的事。

驱动通过调用kill\_fasync(kill\_fasync(&async, SIGIO, POLL\_IN);)发SIGIO信号给应用，应用通过fcntl把自己这个SIGIO的信号换成自己的响应函数，当驱动发(kill\_fasync(&async, SIGIO, POLL\_IN);)给应用时,应用就调用了自己的handler去处理。

|  |
| --- |
| 驱动方面：  1. 在设备抽象的数据结构中增加一个struct fasync\_struct的指针  2. 实现设备操作中的fasync函数，这个函数很简单，其主体就是调用内核的fasync\_helper函数。  3. 在需要向用户空间通知的地方(例如中断中)调用内核的kill\_fasync函数。  4. 在驱动的release方法中调用kpp\_fasync(-1, filp, 0);函数 |
| **Sample:**  #define MEM\_SIZE 256  #define MEM\_NAME "mem"  struct mem\_dev  {  struct cdev dev;  char mem[MEM\_SIZE];  **struct fasync\_struct \*async\_queue;**  };  static struct mem\_dev \*mem\_dev\_p;  static dev\_t mem\_devno;  static struct class \*mem\_class;  static ssize\_t mem\_read(struct file \*filp, char \_\_user \*buf, size\_t count, loff\_t \*pos)  {  struct mem\_dev \*dev\_p = filp->private\_data;  if (count > MEM\_SIZE)  count = MEM\_SIZE;  if (copy\_to\_user(buf, dev\_p->mem, count))  {  return -EFAULT;  }  return count;  }  static ssize\_t mem\_write(struct file \*filp, const char \_\_user \*buf, size\_t count, loff\_t \*pos)  {  struct mem\_dev \*dev\_p = filp->private\_data;  if (count > MEM\_SIZE)  count = MEM\_SIZE;  if (copy\_from\_user(dev\_p->mem, buf, count))  {  return -EFAULT;  }  if (dev\_p->async\_queue)  {  2. 通知用户空间的程序  //当内核执行到kill\_fasync函数，用户空间SIGIO函数的处理函数就会被调用  **kill\_fasync(&dev\_p->async\_queue, SIGIO, POLL\_IN);**  }  return count;  }  //用户空间设置FASYNC 标志时会调用ops的fasync函数:  //fcntl(fd, F\_SETFL, f\_flags | FASYNC);  **static int mem\_fasync(int fd, struct file \*filp, int mode)**  {  struct mem\_dev \*dev\_p = filp->private\_data;  1. 初始化fsaync  return **fasync\_helper(fd, filp, mode, &dev\_p->async\_queue);**  }  static int mem\_open(struct inode \* inode , struct file \* filp)  {  filp->private\_data = mem\_dev\_p;  return 0;  }  static int mem\_release(struct inode \* inode, struct file \*filp)  {  3. 释放fasync  **mem\_fasync(-1, filp, 0);**  return 0;  }  static const struct file\_operations mem\_fops =  {  .owner = THIS\_MODULE,  .open = mem\_open,  .release = mem\_release,  .read = mem\_read,  .write = mem\_write,  **.fasync = mem\_fasync,**  };  static int \_\_init my\_mem\_init(void)  {  int ret = alloc\_chrdev\_region(&mem\_devno, 0, 1, MEM\_NAME);  mem\_dev\_p = kmalloc(sizeof(struct mem\_dev), GFP\_KERNEL);  memset(mem\_dev\_p, 0, sizeof(struct mem\_dev));  cdev\_init(&mem\_dev\_p->dev, &mem\_fops);  mem\_dev\_p->dev.owner = THIS\_MODULE;  mem\_dev\_p->dev.ops = &mem\_fops;  ret = cdev\_add(&mem\_dev\_p->dev, mem\_devno, 1);  mem\_class = class\_create(THIS\_MODULE, "mem\_driver");  device\_create(mem\_class, NULL, mem\_devno, NULL, **"mem\_fasync**");  return 0;  }  static void \_\_exit my\_mem\_exit(void)  {  device\_destroy(mem\_class, mem\_devno);  class\_destroy(mem\_class);  cdev\_del(&mem\_dev\_p->dev);  kfree(mem\_dev\_p);  unregister\_chrdev\_region(mem\_devno, 1);  }  MODULE\_LICENSE("Dual BSD/GPL");  module\_init(my\_mem\_init);  module\_exit(my\_mem\_exit); |

## 异步通知： singal() – 用户空间

|  |
| --- |
| 应用程序步骤:  1）**signal(SIGIO, sig\_handler);**  调用signal函数，让指定的信号SIGIO与处理函数sig\_handler对应。  2）**fcntl(fd, F\_SET\_OWNER, getpid());**  指定一个进程作为文件的“属主(filp->owner)”，这样内核才知道信号要发给哪个进程。  3）设置文件标志，添加FASYNC标志  **f\_flags = fcntl(fd, F\_GETFL);**  **fcntl(fd, F\_SETFL, f\_flags | FASYNC);** |
| **Sample：**  void catch\_sigio(int signu)  {  printf("catch signo\n");  }  int main(void)  {  int fd = open("/dev/**mem\_fasync**", O\_RDWR);  signal(**SIGIO**, catch\_sigio))  fcntl(fd, F\_**SETOWN**, getpid());  int flags = fcntl(fd, F\_GETFL);  fcntl(fd, F\_SETFL, flags | **O\_ASYNC**);  while (1)  {  Sleep(100);  }  return 0;  } |

# Linux驱动之中断与时钟

|  |
| --- |
| **设备申请释放中断：**  **request\_threaded\_irq**(unsigned int irq, irq\_handler\_t handler,  irq\_handler\_t thread\_fn,  unsigned long flags, const char \*name, void \*dev);  static inline int \_\_must\_check **request**\_**irq**(unsigned int irq, irq\_handler\_t handler, unsigned long flags, const char \*name, void \*dev) //申请irq  //等价于 request\_threaded\_irq(irq, handler, NULL, flags, name, dev);  **free\_irq**(unsigned int irq, void \*dev\_id); //释放中断  irq是要申请的中断号;  handler() : 中断回调函数  irqflags： 中断处理的属性  =SA\_INTERRUPT, 表示中断处理程序是快速处理程序，它被调用时屏蔽所有中断。=SA\_SHIRQ, 表示多个设备共享中断，  dev\_id： 中断共享时会用到，一般设置为这个设备的设备结构体或者NULL.  返回值：  0表示成功，  -INVAL表示中断号无效或处理函数指针为NULL,  EBUSY表示中断已经被占用且不能共享。  **使能和屏蔽中断：**  void **disable**\_**irq**(int irq); //这个会立即返回  void **disable**\_**irq**\_**nosync**(int irq);//等待目前的中断处理完成再返回。  void **enable**\_**irq**(int irq);  上述三个函数作用于可编程中断处理器，因此对系统内所有的CPU都生效。  void **local**\_**irq**\_**save**(unsigned long flags);//会将目前的中断状态保留在flags中  void **local**\_**irq**\_**disable**(void);//直接中断  void **local**\_**irq**\_**restore**(unsigned long flags);  void **local**\_**irq**\_**enable**(void);  //local\_开头的方法只适用于本CPU中  **tasklet：**  void **my**\_**tasklet**\_**function**(unsigned long); //定义一个处理函数  **DECLARE**\_**TASKLET**(**my**\_**tasklet**, **my**\_**tasklet**\_**function**, data);  //定义了一个名叫my\_tasklet的tasklet并将其与处理函数绑定，而传入参数为data  **tasklet**\_**schedule**(&**my**\_**tasklet**); //调度tasklet处理函数  **工作队列：**  struct **work**\_**struct** my\_wq; //定义一个工作队列  void **my**\_**wq**\_**func**(unsigned long); //定义一个处理函数  **INIT**\_**WORK**(&my\_wq, (void (\*)(void \*))**my**\_**wq**\_**func**, NULL); //初始化工作队列  **schedule**\_**work**(&**my**\_**irq**)；//调度工作队列  **软中断： 软件方式模拟硬件中断，tasklet也是基于软中断实现的。**  **softirq**\_**action**//软中断结构体，这包含软中断处理函数指针和传递给函数的参数，  **open**\_**softirq**() //注册软中断对应的处理函数  **raise**\_**softirq**() //触发软中断。 |
| **定时器：**  void **init**\_**timer**(struct timer\_list \*timer); //初始化定时器：  void **add**\_**timer**(struct timer\_list \*timer); //注册内核定时器  int **del**\_**timer**(struct timer\_list \*timer); //删除定时器  int **mod**\_**timer**(struct timer\_list \*timer, unsigned long expires); //修改定时器的expires  **短延迟： 不建议用，浪费CPU资源**  void **ndelay**(unsigned long nsecs); //纳秒延迟  void **udelay**(unsigned long usecs); //微秒延迟  void **mdelay**(unsigned long msecs); //毫秒延迟  **长延迟: 比较当前的jiffies和目标jiffies，直到未来的jiffies达到目标jiffies。**  unsigned long delay = jiffies + 100; //延迟100个jiffies  while(time\_before(**jiffies**, **delay**));  **睡着延迟：进程会处于睡眠状态，把CPU空出来。**  void **msleep**(unsigned int msecs); //毫秒延迟  unsigned long **msleep**\_**interruptible**(unsigned int msecs); //毫秒延迟, 可以倍signel唤醒  **time**\_**on**\_**timeout**(wait\_queue\_head\_t \*q, unsigned long timeout);  **interruptible**\_**sleep**\_**on**\_**timeout**(wait\_queue\_head\_t \*q, unsigned long timeout);  //这两个将当前进程添加到等待队列，从而在等待队列上睡眠，当超时发生时，进程将被唤醒。 |

**啥叫中断？**

就是指cpu在执行过程中，出现了某些突发事件时CPU必须暂停执行当前的程序，转去处理突发事件，处理完毕后CPU有返回原程序被中断的位置并继续执行。

**中断的分法：**

按照中断入口跳转方法的不同，可分为向量中断和非向量中断。

1. 采用向量中断的CPU通常为不同的中断分配不同的中断号，当检测到某中断号的中断到来后，就自动跳转到与该中断号对应的地址执行。不同的中断号有不同的中断地址（即入口）。

2. 非向量中断的多个中断共享一个入口地址。进入后根据软件判断中断标志来识别具体是哪个中断。也就是说，向量中断是由硬件提供中断服务程序入口地址，非向量中断由软件提供中断服务程序入口地址。

**时钟定时器：**

它也是通过中断来实现的。它的原理很简单，嵌入式微处理器它接入一个时钟输入，当时钟脉冲到来时，就将目前的计数器值加1并和预先设置的计数值比较，若相等，证明计数周期满，产生定时器中断并复位目前计数器值。

**中断的顶半部和底半部：**

设备的中断会打断内核中进程的正常调度和运行，会影响系统的性能。为了在中断执行时间尽可能短和中断处理需完成大量工作之间找到一个平衡点，Linux将中断处理程序分解成两个半部：顶半部和底半部。其中顶半部尽可能完成尽可能少的比较紧急的功能。而底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。

在linux系统中主要通过tasklet，工作队列，软中断实现。

**软中断，tasklet，工作队列区别：**

软中断和tasklet仍然运行与中断上下文，而工作队列则运行于进程上下文。因此，软中断和tasklet的处理函数不能休眠，但工作队列是可以的。

local\_bh\_disable()/local\_bh\_enable()是内核用于禁止和使能软中断和tasklet底半部机制的函数。

## 中断处理函数模板

|  |
| --- |
| **irq\_handler**()  {  ...  int int\_src = **read**\_**int**\_**status**(); //读硬件的中断相关寄存器  **switch(int\_src)**  //判断中断源  ｛  case DEV\_A:  dev\_a\_handler();  break;  case DEV\_B:  dev\_b\_handler();  break;  ....  default:  break;  ｝  } |

## tasklet模版

|  |
| --- |
| void **xxx**\_**do**\_**tasklet**(unsigned long);  Step2. 定义tasklet的名字和处理函数  **DECLARE**\_**TASKLET**(XXX\_tasklet\_name, **xxx**\_**do**\_**tasklet**, 0);  void **xxx\_do\_tasklet**(unsigned long) //中断处理底半部  {  Step5. 执行中断处理底半部  .....  }  irqreturn\_t **xxx\_interrupt**(int irq, void \*dev\_id, struct pt\_regs \*regs) //中断处理顶半部  {  Step3. 中断触发，执行中断处理顶半部  …  Step4. 唤醒tasklet  **tasklet**\_**schedule**(&xxx\_tasklet\_name);  }  int \_\_init **xxx\_init**(void) //设备驱动模块加载函数  {  Step1. 申请中断。 中断号(xxx\_irq), 中断处理函数(xxx\_interrupt)  result= **request**\_**irq**(xxx\_**irq**, xxx\_**interrupt**, SA\_**INTERRUPT**, "XXX",NULL);  }  void \_\_exit **xxx\_exit**(void) //设备驱动卸载模块  {  Step6. 释放中断  **free**\_**irq**(xxx\_**irq**, xxx\_**interrupt**);  } |

## 工作队列模版

|  |
| --- |
| Step2. 定义工作队列  struct work\_struct xxx\_wq;  void xxx\_do\_**work**(unsigned long);  void **xxx\_do\_work**(unsigned long) //中断处理底半部  {  Step5. 执行中断处理底半部  .....  }  irqreturn\_t **xxx\_interrupt**(int irq, void \*dev\_id, struct pt\_regs \*regs) //中断处理顶半部  {  Step4. 中断触发，执行中断处理顶半部  …  Step5. 唤醒工作队列  **schedule**\_**work**(&xxx\_wq);  }  int **xxx\_init**(void) //设备驱动模块加载函数  {  Step1. 申请中断。 中断号(xxx\_irq), 中断处理函数(xxx\_interrupt)  result= **request**\_**irq**(xxx\_**irq**, xxx\_interrupt, SA\_**INTERRUPT**, "XXX",NULL);  Step3. 定义工作队列的处理函数  **INIT**\_**WORK**(&xxx\_wq, (void (\*)(void \*))**xxx**\_**do**\_**work**, NULL);  }  void \_\_exit **xxx\_exit**(void) //设备驱动卸载模块  {  Step6. 释放中断  **free**\_**irq**(xxx\_**irq**, xxx\_**interrupt**);  } |

## 共享中断处理函数模板

|  |
| --- |
| irqreturn\_t **xxx\_interrupt**(int irq, void \*dev\_id, struct pt\_regs \*regs) //中断处理顶半部  {  Step2. 中断触发，执行中断处理函数  int status = read\_int\_status(); //获取终端源  if(!is\_myint(dev\_id, status)) //判断是否是本设备的中断  {  return IRQ\_NONE: //立即返回  }  ..  return IRQ\_HANDLED;  }  int \_\_init **xxx\_init**(void) //设备驱动模块加载函数  {  Step1. 申请共享中断。中断号(xxx\_irq), 中断处理函数(xxx\_interrupt)  result= **request**\_**irq**(xxx\_irq, xxx\_interrupt, SA\_**SHIRQ**, "XXX",xxx\_dev);  }  void \_\_exit **xxx\_exit**(void) //设备驱动卸载模块  {  Step3. 释放中断  **free**\_**irq**(xxx\_irq, xxx\_interrupt);  } |

## 定时器模板

|  |
| --- |
| struct xxx\_dev  {  struct cdev cdev; /\*cdev结构体\*/  Step1. 定义定时器结构体  **struct timer\_list xxx\_timer;**  };  int xxx\_func1(...) //xxx驱动中某函数  {  struct xxx\_dev \*dev = filp->private\_data;  ...  Step2. 初始化定时器: 回调函数(xxx\_do\_handle), 超时时间(expires)  **init**\_**timer**(&dev->xxx\_timer);  dev->xxx\_timer.**function** = &**xxx**\_**do**\_**handle**;  dev->xxx\_timer.**data** = (unsigned long)dev;  dev->xxx\_timer.**expires** = jiffies + delay;  Step3. 注册定时器，等待时间超时触发处理函数  **add**\_**timer**(&dev->xxx\_timer);  ...  return 0;  }  int xxx\_func2(...) //驱动中某函数  {  Step6. 删除定时器  **del**\_**timer**(&second\_devp->s\_timer);  ...  }  static void xxx\_do\_timer(unsigned long arg) //定时器处理函数  {  struct xxx\_device \*dev = (struct xxx\_device \*)(arg);  Step4. 处理定时器中断  ...  Step5. 注册定时器，等待时间超时触发处理函数  dev->xxx\_timer.**expires** = jiffies + delay;  **add**\_**timer**(&dev->xxx\_timer);  } |

# Linux驱动之内存与I/O访问

**MMU(内存管理单元)的几个概念：**

1）**TLB:Translation Lookaside Buffer**，即转换旁路缓存。也称快表，是转换表的cache，缓存少量的虚拟地址与物理地址的对应关系。

2）**TTW:Translation Table walk**，即转换表漫游。当TLB中没有缓冲对应的地址转换关系时，需要通过对内存中转换表的访问来获得虚拟地址和物理地址的对应关系。TTW成功后，结果应写入TLB.

**用户空间和内核空间**

前者一般分布为**0~3GB**(即PAGE\_OFFSET,在0x86中等于0xC0000000),剩下的3~4GB是内核空间。通常情况下，用户进程只有通过系统调用等方式才可以访问到内核空间。

linux**内核空间3~4GB**是还可以在分的,从低到高依次是：物理内存映射区->隔离带->vmalloc虚拟内存分配器->隔离带->高端内存映射区->专用页面影视区->保留区。

**Mmap()函数**

mmp()必须以PAGE\_SIZE为单位进行映射

mmap()实现的机制是**建立页表**，并填充VMA结构体中.

vm\_area\_struct用于描述一个**虚拟内存**区域。

**DMA**

DMA是一种无须CPU的参与就可以让外设与系统内存之间进行双向数据传输的硬件机制。DMA方式的数据传输由**DMA控制器**(DMAC)控制，在传输期间，CPU可以并发地执行其他任务，当DMA结束后，DMAC通过**中断**通知CPU数据传输已经结束，然后由CPU执行相应的中断服务程序进行后续处理。

说到**DMA**，就会想到**Cache**，两者本身似乎是好不相关的事物。的确，假设DMA针对内存的目的地址和Cache缓存的对象没有重叠区域，DMA和Cache之间就相安无事，但是，如果有**重叠**呢，经过DMA操作，Cache缓存对应的内存的数据已经被修改，而CPU本身并不知道，它仍然认为Cache中的数据仍然还是内存中的数据，以后访问Cache映射的内存时，它仍然使用陈旧的Cache数据，这就会发生Cache与内存之间数据“不一致性”的错误。一旦出现这样的情况，没有处理好，驱动就将无法正常运行。那么怎样解决呢？最简单的方法是直接**禁止DMA目标地址范围内内存的Cache功能**，当然这是牺牲性能的，但却高可靠。

DMA的硬件使用**总线地址**而非物理地址，**总线**地址是**从设备角度上看到的内存地址**，**物理**地址是**从CPU角度上**看到的未经转换的内存地址(经过转换的那叫虚拟地址)。在PC上，对于ISA和PCI而言，总线即为物理地址，但并非每个平台都是如此。由于有时候接口总线是通过桥接电路被连接，桥接电路会将IO地址映射为不同的物理地址。例如，在PRep(PowerPC Reference Platform)系统中，物理地址0在设备端看起来是0X80000000，而0通常又被映射为虚拟地址0xC0000000,所以同一地址就具备了三重身份：物理地址0,总线地址0x80000000及虚拟地址0xC0000000,还有一些系统提供了页面映射机制，它能将任意的页面映射为连续的外设总线地址。

|  |
| --- |
| **申请内存：**  void \***kmalloc**(unsigned int len, int priority);  void **kfree**(void \*ptr);  //kmalloc()用于分配在物理上连续的内存，虚拟地址也是连续的，是基于slab分配实际上存在的内存。  //但是kmalloc最多只能开辟大小为32XPAGE\_SIZE的内存,一般的PAGE\_SIZE=4kB,也就是128kB的大小的内存。  static inline void \* **kcalloc** (size\_t n, size\_t size, gfp\_t flags);  void **kfree**(void \*ptr);  //kcalloc()是基于slab分配实际上存在的内存，并且在分配后将内存中的内容都初始化为0，但kcalloc()主要为一个数组分配内存空间，数组中的一个元素对应一个内存对象  static inline void \* **kzalloc** (size\_t size, gfp\_t flags);  void **kfree**(void \*ptr);  //kzalloc()是基于slab分配实际上存在的内存，在分配内存后将内存中的内容都初始化为0,和kcalloc()有些相似，但不针对数组。  void \* \_\_must\_check **krealloc** (const void \*p, size\_t new\_size, gfp\_t flags);  //krealloc()重新分配内存，但不改变原地址空间中的内容  void \***vmalloc**(unsigned long size);  **vfree**()  //如果要申请一片连续的虚拟内存，需要使用vmalloc函数。vmalloc返回的虚拟内存虽然是连续的，但是映射到的物理内存是不连续的，而且可能与物理地址不是一一对应的（不同于kmalloc和\_\_get\_free\_pages）。所以在使用它分配的内存时，页表的查询比较频繁，所以效率相对较低。  struct page \***vmalloc**\_**to**\_**page**(const void \*addr);  //找到vmalloc()所分配内存的虚拟地址所映射的物理页，并返回该页的指针描述符  void \***vmalloc**\_**user**(unsigned long size);  vfree();  //功能类似于vmalloc(),在vmalloc()的基础上将地址空间清零，这样该地址空间被映射到用户空间不会发生数据泄露  **分配页以上的大内存**  unsigned long **get**\_**zeroed**\_**page**(unsigned int gfp\_mask)；  //返回一个单个的, 零填充的页  unsigned long \_\_**get**\_**free**\_**pages**(unsigned int gfp\_mask, unsigned int order)；  //直接获取整页的内存（页数是2的幂）  void **free**\_**pages**(unsigned long addr, unsigned int order)；  //释放面向页分配的函数  **分配DMA内存**  #define \_\_**get**\_**dma**\_\_**pages**(gfp\_mask, order) \_\_**get**\_**free**\_**pages**((gfp\_mask) | **GFP**\_**DMA**, (order))  static unsigned long **dma**\_**mem**\_**alloc**(int size) //简单版  {  int order = get\_order(size); //大小->指数  return \_\_get\_dma\_pages(GFP\_KERNEL, order);  }  // DMA映射必须考虑Cache一致性问题。内核中提供了以下函数用于分配一个DMA一致性的内存区域  void \***dma\_alloc\_coherent**(struct device \*dev, size\_t size, dma\_addr\_t \*handle, gfp\_t gfp);  void **dma**\_**free**\_**coherent**(struct device \*dev, size\_t size, void \*cpu\_addr, dma\_addr\_t handle);  //分配一个写合并(writecombinbing)的DMA缓冲区：  void \***dma**\_**alloc**\_**writecombine**(struct device \*dev, size\_t size, dma\_addr\_t \*handle, gfp\_t gfp);  dma\_free\_writecombine();  //PCI设备申请DMA缓冲区的函数  void \***pci**\_**alloc**\_**consistent**(struct pci\_dev \*dev, size\_t size, dma\_addr\_t \*dma\_addrp);  void **pci**\_**free**\_**consistent**(struct pci\_dev \*pdev, size\_t size, void \*cpu\_addr, dma\_addr\_t dma\_addr);  //申请DMA通道的函数如下：  int **request**\_**dma**(unsigned int dmanr, const char \* device\_id);  void **free**\_**dma**(unsinged int dmanr);  **虚拟地址和总线地址的互换：**  unsigned long **virt**\_**to**\_**bus**(volatile void \*address);  void \***bus**\_**to**\_**virt**(unsigned long address);  **虚拟地址和物理地址的互换：**  **virt**\_**to**\_**phys**(); //实现虚拟地址转换为物理地址  **phys**\_**to**\_**virt**(); //将物理地址转化为虚拟地址  //上述方法仅适用与常规内存，高端内存的虚拟地址与物理地址之间不存在这样简单的换算关系  **Mmap():将用户空间的一段内存与设备内存关联**  int (\***mmap**)(struct file \*, struct vm\_area\_struct \*);  **I/O内存：**  void \***ioremap**(unsigned long offset, unsigned long size); //物理地址映射到虚拟地址  void **iounmap**(void \*addr); //释放内存  **I/O端口操作： port**  unsigned **inb**(unsigned port); //读端口， 8字节  void **outb**(unsigned char byte, unsigned port); //写端口， 8字节  unsigned **inw**(unsigned port); //读， 16字节  void **outw**(unsigned short word, unsigned port); //写， 16字节  unsigned **inl**(unsigned port); //读， 32字节  voi **outl**(unsigned longword, unsigned port); //写， 32字节  unsigned **insb**(unsigned port， void \*addr， unsigned long count); //读字串，字节  voi **outsb**(unsigned port, void \*addr, unsigned long count); //写字串, 字节  unsigned **insw**(unsigned port， void \*addr，unsigned long count); //读字串，字  voi **outsb**(unsigned port, void \*addr, unsigned long count); //写字串，字  unsigned **insl**(unsigned port， void \* addr， unsigned long count); //读字串，长字  voi **outsb**(unsigned port, void \*addr, unsigned long count); //写字串，长字  //上述各函数中I/O端口port的类型长度依赖与具体的硬件平台，所以只是写出了unsigned  //读写内存: 字，字节类型  unsigned int **ioread8**(void \*addr);  unsigned int **ioread16**(void \*addr);  unsigned int **ioread32**(void \*addr);  void **iowrite8**(u8 value，void \*addr);  void **iowrite16**(u16 value, void \*addr);  void **iowrite32**(u32 value, void \*addr);  //读写内存: 字符串类型  void **ioread8**\_**rep**(void \*addr， void \*buf, unsigned long count);  void **iowrite8**\_**rep**(void \*addr， void \*buf, unsigned long count);  void **ioread16**\_**rep**(void \*addr， void \*buf, unsigned long count);  void **iowrite8**\_**rep**(void \*addr， void \*buf, unsigned long count);  void **ioread32**\_**rep**(void \*addr， void \*buf, unsigned long count);  void **iowrite8**\_**rep**(void \*addr， void \*buf, unsigned long count);  //复制IO内存  void **memcpy**\_**fromio**(void \*dest, void \*source, unsigned int count);  void **memcpy**\_**toio**(void \*dest, void \*source, unsigned int count);  //设置IO内存  void \***ioport**\_**map**(unsigned long port, unsigned int count);  //把IO端口映射到内存空间  void \***ioport**\_**map**(unsigned long port, unsigned int count);  //把port开始的count个连续的IO端口重映射为一段“内存空间”  void **ioport**\_**unmap**(void \*addr); //撤销这种映射  //IO端口申请  struct resource \***request**\_**region**(unsigned long first, unsigned long n, const char \*name);  //内核申请n个端口，这些端口从first开始，name参数为设备的名称  void **release**\_**region**(unsigned long start, unsigned long n);  //IO内存申请  struct resource \***request**\_**mem**\_**region**(unsigned long start, unsigned long len, char \*name);  //向内核申请n个内存，这些地址从first开始，name为设备的名称  void **release**\_**mem**\_**region**() ; //释放归回给系统 |

## Kmalloc()之flag参数

|  |  |  |
| --- | --- | --- |
| 参  数  值 | 含    义 | 备    注 |
| GFP\_KERNEL | 运行在内核空间的进程使用。当空闲内存较少时，可能进入休眠来等待一个页面 | 使用 GFP\_KERNEL来分配内存的函数必须是可重入的，且不能在原子上下文中运行 |
| GFP\_ATOMIC | 原子性的内核空间分配。进程不能被置为睡眠时，应使用 GFP\_ATOMIC | 常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，不会导致睡眠 |
| GFP\_USER | 为用户空间分配内存页 | 可能导致睡眠 |
| GFP\_HIGHUSER | 类似 GFP\_USER，如果有高端内存，就从高端内存分配 |  |
| GFP\_NOIO | 类似 GFP\_KERNEL。但禁止任何 I/O 初始化 |  |
| GFP\_DMA | 获取具有DMA功能的地址空间 |  |
| GFP\_NOFS | 类似 GFP\_KERNEL。但不允许执行任何文件系统调用 | 主要用在文件系统 |

## Mmap() – 内核空间

mmp()必须以PAGE\_SIZE为单位进行映射

mmap()实现的机制是建立页表，并填充VMA结构体中.

vm\_area\_struct用于描述一个虚拟内存区域。

|  |
| --- |
| **int (\*mmp)(struct file \*, struct vm\_area\_struct \*);**  **int munmap(caddr\_t addr, size\_t len);**  **1. vm\_area\_struct用于描述一个虚拟内存区域。结构体如下：**  struct vm\_area\_struct {  struct mm\_struct \* vm\_mm; //所处的地址空间  unsigned long vm\_start;//开始虚拟地址  unsigned long vm\_end;//结束虚拟地址  struct vm\_area\_struct \*vm\_next;  pgprot\_t vm\_page\_prot; //访问权限  unsigned long vm\_flags; //标志  ...  struct vm\_operations\_struct \* vm\_ops; //操作VMA的函数集指针  unsigned long vm\_pgoff; //偏移(页帧号)  struct file \* vm\_file;  void \* vm\_private\_data;  ...  };  **2. vm\_operations\_struct定义虚拟内存的操作函数，结构体定义如下：**  struct vm\_operations\_struct {  void (\*open)(struct vm\_area\_struct \* area);  void (\*close)(struct vm\_area\_struct \* area);  struct page \* (\*nopage)(struct vm\_area\_struct \* area, unsigned long address, int \*type）；  int (\*populate)(struct vm\_area\_struct \* area, unsigned long address, unsigned long len, pgprot\_t prot, unsigned long pgoff, int nonblock);  ...  };  **3. 建立页表。即映射。**  int remap\_**page**\_range(unsigned long **start**, unsigned long **phy\_addr**, unsigned long **size**, pgprot\_t **prot**);  int remap\_**pfn**\_range(struct vm\_area\_struct \***vma**, unsigned long **addr**, unsigned long **pfn**, unsigned long **size**, pgprot\_t **prot**);  //remap\_page\_range和 remap\_pfn\_range的区别就是前者用实际的物理地址（第二个参数）建立页表，后者用页桢号建立页表。页桢号实际上是实际的物理地址右移PAGA\_SIZE得到的。  驱动中的.mmap函数将在用户进行mmap()系统调用时最终被调用。  当用户调用mmap()时候内核会进行如下处理：  1)在进程的虚拟空间查找一块VMA.  2)将这块VMA进行映射。  3)如果设备驱动程序或文件系统的file\_operations定义了mmap()操作，则调用它。  4)将这个VMA插入到进程的VMA链表中。  **注意的是：**  **当用户进行mmap()系统调用后，尽管VMA在设备驱动文件操作结构体的mmap()被调用前就已经产生，内核却不会调用VMA的open函数，通常需要在驱动的mmap()函数中先上调用vma->vm\_ops->open().** |

## Mmap() – 用户空间

|  |
| --- |
| **caddr\_t mmap(caddr\_t addr,size\_t len,int prot,int flags,int fd,off\_t offset);**  prot : 指定访问权限，  PROT\_READ(可读)，  PROT\_WRITE(可写)  PROT\_EXEC（可执行）  PROT\_NONE（不可访问）  caddr\_t: 实际上是 void \*; |

## Mmap模板1

mmp()必须以PAGE\_SIZE为单位进行映射

mmap()实现的机制是**建立页表**，并填充VMA结构体中.

vm\_area\_struct用于描述一个**虚拟内存**区域。

|  |
| --- |
| static int **xxx\_mmap**(struct file \*filp, struct vm\_area\_struct \*vma)  {  if(remap\_**pfn**\_range(vma, vma->vm\_start, vm->vm\_pgoff, vma->vm\_end - vma->start, vma->vm\_page\_prot)) //建立页表, 即映射  return - EAGAIN;  vma->vm\_ops = &**xxx\_remap\_vm\_ops**;  **xxx**\_**vma**\_**open**(vma);  return 0;  }  void xxx\_vma\_open(struct vm\_area\_struct \*vma) //VMA打开函数  {  ...  printk(KERN\_NOTICE "xxx VMA open, virt %lx, phys %1x\n",vma->vm\_start, vma->vm\_pgoff 《PAGE\_SHIFT);  }  void xxx\_vma\_close(struct vm\_area\_struct \*vma) //VMA关闭函数  {  ...  printk(KERN\_NOTICE "xxx VMA close. \n");  }  static struct **vm\_operation\_struct xxx**\_**remap**\_**vm**\_**ops** = //VM操作结构体  {  .open=**xxx**\_**vma**\_**open**,  .close=xxx\_vma\_close,  ...  } |

## Mmap模板2

remap\_pfn\_range函数的一个限制是：它只能访问保留页和超出物理内存的物理地址。（Linux中，在内存映射时，物理地址页被标记为reserved，表示内存管理对其不起作用。如在PC中，640KB-1MB之间的内存被标记为保留的，因为这个范围位于内核自身代码的内部。）所以remap\_pfn\_range不允许重新映射常规地址。包括调用get\_free\_page函数所获得的地址。相反，他能映射零内存页。（引用来自网络）

kmalloc()申请的内存若要被映射到用户空间可以通过mem\_map\_reserve()设置为保留后进行。

kmalloc()申请的内存若要被映射到用户空间可以通过mem\_map\_reserve()设置为保留后；在mmap()中调用remap\_page\_range()建立页表，将kmalloc()的空间地址映射到用户空间。

|  |
| --- |
| // 内核模块加载函数  int \_\_init kmalloc\_map\_init(void)  {  ../申请设备号，添加cedv结构体  buffer = kmalloc(BUF\_SIZE, GFP\_KERNEL); //申请buffer  for(page = virt\_to\_page(buffer); page< virt\_to\_page(buffer+BUF\_SIZE); page++)  {  mem\_map\_reserve(page); //置业为保留  }  }  //mmap()函数  static int kmalloc\_map\_mmap(struct file \*filp, struct vm\_area\_struct \*vma)  {  unsigned long page, pos;  unsigned long start = (unsigned long)vma->start;  unsigned long size = (unsigned long)(vma->end - vma->start);  printk(KERN\_INFO, "mmaptest\_mmap called\n");  if(size > BUF\_SIZE) //用户要映射的区域太大  return - EINVAL;  pos = (unsigned long)buffer;  while(size > 0) //映射buffer中的所有页  {  page = virt\_to\_phys((void \*)pos);  if(remap\_**page**\_range(start, page, PAGE\_SIZE, PAGE\_SHARRED))  return -EAGAIN;  start += PAGE\_SIZE;  pos +=PAGE\_SIZE;  size -= PAGE\_SIZE;  }  return 0;  } |

# End