Linux 设备驱动之并发控制(一)

——编写者:草根老师(程姚根)

一、并发和竞态

并发:多个执行单元同时进行 或 多个执行单元微观串行执行，宏观并行执行
竞态:并发的执行单元对共享资源(硬件资源和软件上的全局变量)的访问而导致的竞争状态

了解完概念之后，我们来看看在操作系统运行的过程中，什么时候会出现并发呢?

(1)多CPU之间的并发

例如:
CPU_A 运行的程序访问串口资源，CPU_B运行的程序也访问串口资源

(2)单CPU之间进程间的并发

由于现在的操作系统都是多任务的，即多进程，一个进程在运行的时候，不是一直占用CPU资源直到它结束，而是每个进程都有一个时间片，如果时间片用完了操作系统就调度另外一个进程执行。

例如:
如果A进程访问了串口资源，由于时间片用完了，操作系统调度B进程访问串口资源。这种微观串行，宏观并行的也称为并发。

(3)单CPU上进程和中断之间的并发

在一个进程执行的过程中，如果硬件上产生了中断请求，此时操作系统必须放弃进程的执行，去执行中断处理函数，进行中断处理。

例如:
A进程正在访问串口资源,此时产生了中断，在中断处理函数中，也访问了串口资源。

(4)单CPU上中断之间的并发

在硬件上设计中断的时候，每个中断都会有一个优先级。如果CPU在处理中断的时候（执行这个中断对应的中断处理函数），此时如果来了一个高优先级的中断，此时CPU就会放弃此次中断处理而去响应高优先级的中断。

例如:
CPU正在执行中断A的中断处理函数访问串口资源，此时中断B产生，中断B的优先级高于中断A，CPU会立即响应中断B的而放弃中断A。悲催的是中断B的中断处理函数也是访问串口资源。

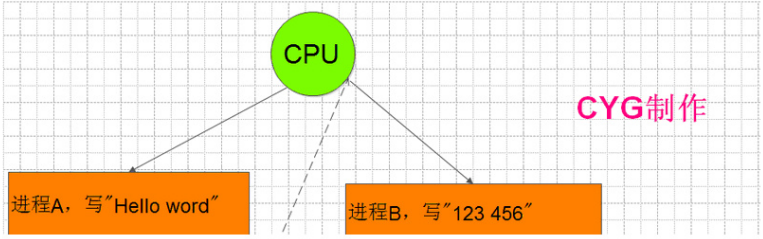
二、为什么需要进行并发控制？

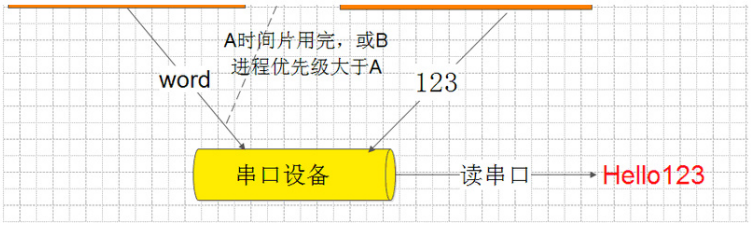
我们了解过了并发，大家应该明白，操作系统之所以要并发执行的目的只有一个，那就是提高效率。大家一起干活效率就会很高，时间就会缩短，效率是用户永远的追求。

如果我们在写程序的时候，什么都不管，那操作系统在并发执行的时候又出现什么效果呢?

最近东莞事件挺火的，那是一个神奇的地方，江湖传言哪里汇集了全宇宙的MM，是男人的天堂。试想如果两个哥们同时或先后看上一个MM,如果老板啥都不管，那会出现什么情况,我没去过不知道，去过了人应该很清楚....

在上面的例子中，我们会发现多个执行单元都会访问串口资源，他们共同的特点(除了多CPU)是,A执行单元没有完全使用完串口资源的时候，就被B执行单元打断，而B执行单元也访问了串口资源。如果我们写程序的时候，什么都不管，就极有可能出现以下情况。





现在明白，为什么要进行并发控制了吧!不进行并发控制就会乱套。基于上图而言，如何才能不乱套呢?

解决方案:
进程A在访问串口设备期间，不允许B进程访问，访问完串口设备后，才允许B进程访问。

此时B进程情何以堪?
(1)B进程一直在激情的奋斗着去占用串口设备，通过N个奋斗拿到了串口设备(A进程访问完了),开始向串口中写数据。
(2)B进程觉得反正现在不能使用串口设备，我先休息呀，小A进程你访问完串口设备后，你叫醒我。
(3)B进程立即访返回，告诉用户不是我偷懒，是你的设备正在被别人访问，此时我没办法访问

三.并发控制之中断屏蔽

在操作系统中，进程调度的操作是基于中断来实现的。也就是说一旦我们把中断屏蔽了，也就不存在进程抢占和进程调度了，更不可能出现中断。这样当前的进程就会一直执行，直到它取消中断屏蔽。

现在我们来想，它是否能避竞态的产生呢?
(1)对于单CPU而言，是可以的
(2)对于多CPU而言是不行的。因为每个CPU上都运行了一个进程，此时虽然中断屏蔽了，但是每个CPU上还是有进程在运行的，如果这些进程碰巧访问都是同一个设备，一样会导致竞态。

好了，接下来我们来看看在Linux 系统中如何实现中断屏蔽吧!

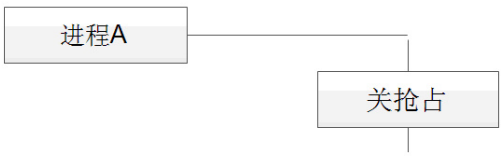
```
#include <linux/irqflags.h>

//关闭中断
local_irq_disable();

访问共享资源

//使能中断
local_irq_enable();

------------------------------------------------------

//保存当前CPSR的值并且关闭中断
local_irq_save(unsigned long flags);

访问共享资源

//回复中断之前的状态
local_irq_restore(unsigned long flags);
```

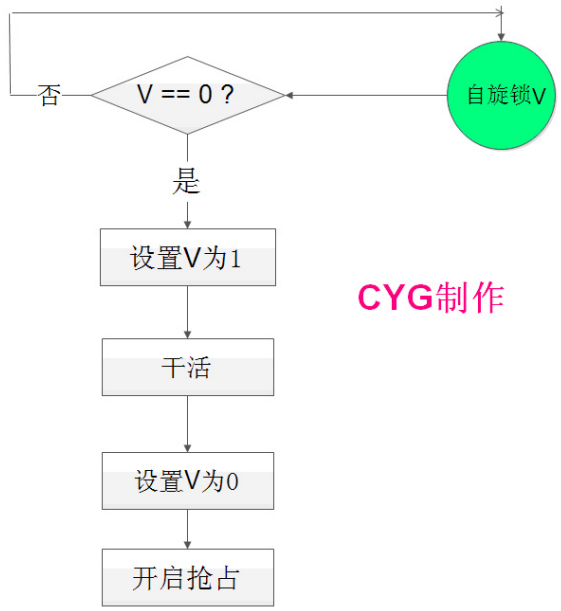
虽然屏蔽中断可以解决竞态,但是请慎用，因为一旦中断屏蔽，很多事情都做不了了,而且需要注意访问共享资源的时间需要尽可能的短.以便尽快的回复中断。

四.并发控制之自旋锁(spinlock)

自旋锁刚开始引入Linux 内核的目的是为了解决多处理器上对共享资源的访问产生的竞态,它是一种神奇的锁，有锁则开门，无锁则自旋,所以此锁称为自旋锁。如果一个进程如果在获取自旋锁的时候，发现自旋锁已经被其他人获取了，此时它就会自旋(所谓的自旋就是不断的测试锁是否可以获得锁，直到获得锁)。

知道概念后我们来看看，Linux 内核实现自旋锁的方法:





从上图可以看出Linux 内核在实现自旋锁为了避免竞态只是使用了一个全局变量。当这个全局变量的值为0时，代表自旋锁空闲，可以获得锁，当这个全局变量的值为1时，代表自旋锁正在被别人占用。

在这里需要理解一下关抢占的概念。

在2.5.4的Linux 内核版本之前，Linux 内核是不可抢占的，高优先级的进程不能中止正在内核中运行的低优先级的进程而抢占上CPU运行。进程一旦处于内核态(例如用户进程调用系统调用而进入内核空间)，则除非进程自愿放弃CPU,否则该进程将一直运行下去，直到完成内核空间程序的执行。与此相反，一个可抢占的Linux内核，不管当前进程处于用户态还是内核态，都会调度优先级高的进程运行,而停止当前进程运行。

在自旋锁使用的期间，抢占是关闭的，即此时内核不会调度其他的进程运行。但是也不是绝对会一直执行

(1)获得互斥锁的进程在操作过程中，显示的调用schedule（）函数，内核就会调度其他进程运行。
(2)获得互斥锁的进程在执行的过程中，中断发生了，此时会被打断，去执行中断处理函数。
(3)获得互斥锁的进程在执行的过程中，由于资源不可用阻塞了，此时内核就会调度其他进程运行。

所以在使用自旋锁的时候要特别小心，谨防死锁发生，一旦发生将是致命的。下面举几个死锁的情况，帮助大家理解自旋锁。

(1)拥有自旋锁的进程A在内核态阻塞了,此时Linux 内核就调度其B进程去运行,碰巧这个进程也需要获得自旋锁，然后去访问共享资源。由于自旋锁已经被A获得了，此时B只能自旋转。不要忘记此时抢占已经关闭，不会调度A进程运行了，A进程持有的锁将永远不会释放，这也会导致B进程永远自旋,产生死锁。

例如: 调用了copy_from_user()、copy_to_user(),kmalloc()等函数都有可能引起*阻塞*。

copy_from_user()和copy_to_user发生阻塞的原因:
当包含用户数据的页被换出到硬盘上而不是在物理内存上的时候，这种情况就会发生。此时，进程就会休眠，直到缺页处理程序将该页从硬盘重新换回物理内存。

kmalloc()函数阻塞的原因:没有可用的内存的时候，就会阻塞调用者。

(2)拥有锁的进程在执行的过程中，中断到来，CPU执行中断处理函数，中断处理函数需要获得自旋锁然后访问共享资源。此时不能获得自旋锁, 只能自旋，产生死锁。

说了这么多，我们来总结一下自旋锁使用的场合以及需要注意的地方:

<1> 可以解决多CPU之间的并发引起的竞态(实现多CPU之间的互斥访问)
<2> 可以解决单CPU上并发引起的竞态
　　方法：
　　a. 单CPU上，进程之间的并发引起的竞态，解决办法是直接加锁(消除单CPU上进程之间的并发)
　　b. 单CPU上，进程与中断之间并发和中断与中断之间并发引起的竞态，解决方法是加锁且关闭中断
<3> 加锁时间不能太长
<4> 获得自旋锁期间，不能有引起调度的函数，自己放弃cpu(休眠是典型的代表)

最后我们来看看在Linux 内核中使用自旋锁的流程:

```
1.定义自旋锁
spinlock_t lock;

2.初始化自旋锁
void spin_lock_init(spinlock_t *lock);

3.获得自旋锁
void spin_lock(spinlock_t *lock);
或
int spin_trylock(spinlock_t *lock);
功能:尝试获得锁，获得返回真，未获得返回假

4.干活(....)

5.释放互斥锁
void spin_unlock(spinlock_t *lock);
```

五、linux抢占发生的时间

最后在了解下linux抢占发生的时间，抢占分为用户抢占和内核抢占。

用户抢占在以下情况下产生：

- 从系统调用返回用户空间
- 从中断处理程序返回用户空间

内核抢占会发生在：

- 当从中断处理程序返回内核空间的时候，且当时内核具有可抢占性；
- 当内核代码再一次具有可抢占性的时候。（如:spin_unlock时）
- 如果内核中的任务显式的调用schedule()
- 如果内核中的任务阻塞。

基本的进程调度就是发生在时钟中断后，并且发现进程的时间片已经使用完了，则发生进程抢占。通常我们会利用中断处理程序返回内核空间的时候可以进行内核抢 占这个特性来提高一些I/O操作的实时性，如：当I/O事件发生的是时候，对应的中断处理程序被激活，当它发现有进程在等待这个I/O事件的时候，它会激 活等待进程，并且设置当前正在执行进程的need_resched标志，这样在中断处理程序返回的时候，调度程序被激活，原来在等待I/O事件的进程（很 可能）获得执行权，从而保证了对I/O事件的相对快速响应（毫秒级）。可以看出，在I/O事件发生的时候，I/O事件的处理进程会抢占当前进程，系统的响 应速度与调度时间片的长度无关。