LINUX的并发与竞争（驱动）

一、简介：

- 并发就是多个“用户 ”同时访问同一个共享资源。Linux系统是个多任务操作系统，会存在多个任务同时访问同一片内存区域，这些任务可

  能会相互覆盖这段内存中的数据，造成内存数据混乱。
- 造成linux系统并发产生的原因：
  - ①、多线程并发访问， Linux是多任务 (线程)的系统，所以多线程 访问是最基本的原因。
  - ②、抢占式并发访问，从 2.6版本内核开始， Linux内核支持抢占，也就是说调度程序可以在任意时刻抢占正在运行的线程，从而运行其他的线程。

    ③、中断程序并发访问，这个无需多说，学过 STM32的同学应该知道，硬件中断的权利是很大的

    ④、 SMP(多核)核间并发访问，现在 ARM架构的多核 SOC很常见，多核 CPU存在核间并发访问。

（之前stm32中FreeRTos中临界区就是为了防止并发的一种手段）

- LINUX中用原子操作来防止并发访问。

二、原子操作：

2.1 、原子操作简介：

原子操作就是指不能 再 进一步分割的 操作，在C语言中一个变量的幅值会借助几个寄存器来完成操作，进而产生并发的风险。

2.2、原子整形操作API 函数：

Linux 内核定义了叫做atomic_t 的结构体来完成整形数据的原子操作，在使用中用原子变量来代替整形变量，此结构体定义在include/linux/types.h 文件中，定义如下：

```
typedef struct{
    int counter;
}atomic_t;
```

如果要使用原子操作API 函数，首先要先定义一个atomic_t 的变量，如下所示：

```
1 atomic_t a; //定义a
```

也可以在定义原子变量的时候给原子变量赋初值，如下所示：

```
1 atomic_t b = ATOMIC_INIT(0); //定义原子变量b 并赋初值为0
```

可以通过宏 ATOMIC_INIT向原子变量赋初值。

原子操作的API函数

函数	描述
ATOMIC_INIT(int i)	定义原子变量的时候对其初始化。
int atomic_read(atomic_t *v)	读取 v 的值，并且返回。
void atomic_set(atomic_t *v, int i)	向 v 写入 i 值。
void atomic_add(int i, atomic_t *v)	给 v 加上 i 值。
void atomic_sub(int i, atomic_t *v)	从 v 减去 i 值。
void atomic_inc(atomic_t *v)	给 v 加 1，也就是自增。
void atomic_dec(atomic_t *v)	从 v 减 1，也就是自减
int atomic_dec_return(atomic_t *v)	从 v 减 1，并且返回 v 的值。
int atomic_inc_return(atomic_t *v)	给 v 加 1，并且返回 v 的值。
int atomic_sub_and_test(int i, atomic_t *v)	从 v 减 i，如果结果为 0 就返回真，否则返回假
int atomic_dec_and_test(atomic_t *v)	从 v 减 1，如果结果为 0 就返回真，否则返回假
int atomic_inc_and_test(atomic_t *v)	给 v 加 1，如果结果为 0 就返回真，否则返回假
int atomic_add_negative(int i, atomic_t *v)	给 v 加 i，如果结果为负就返回真，否则返回假

当使用64位的SOC时，函数有用法一样，只是将“ atomic_”前缀换为 atomic64_”，将 int换为 long long.

2.3、原子位操作：

原子位操作直接对内存进行操作，API如下：

函数	描述
void set_bit(int nr, void *p)	将 p 地址的第 nr 位置 1。
void clear_bit(int nr,void *p)	将 p 地址的第 nr 位清零。
void change_bit(int nr, void *p)	将 p 地址的第 nr 位进行翻转。
int test_bit(int nr, void *p)	获取 p 地址的第 nr 位的值。
int test_and_set_bit(int nr, void *p)	将 p 地址的第 nr 位置 1，并且返回 nr 位原来的值。
int test_and_clear_bit(int nr, void *p)	将 p 地址的第 nr 位清零，并且返回 nr 位原来的值。
int test_and_change_bit(int nr, void *p)	将 p 地址的第 nr 位翻转，并且返回 nr 位原来的值。

三、自旋锁：

3.1、自旋锁简介：

原子操作只能对整形变量或者位进行保护。设备结构体变量就不是整型变量，所以引入自旋锁。当一个线程要访问某个 共享资源 的时候首先要先获取相应的锁， 锁

只能被一个线程持有，只要此线程不释放持有的锁，那么其他的线程就不能获取此锁。对于自旋锁而言，如果自旋锁正在被线程 A持有，线程 B想要获取自旋锁，那么线程 B就会处于忙循环 -旋转 -等待状态，线程 B不会进入休眠状态或者说去做其他的处理，而是会一直傻傻的在那里“转圈圈”的等待锁可用。

Linux内核使用结构体 spinlock_t表示自旋锁，结构体定义如下所示：

```
typedef struct spinlock{
    union{
        struct raw_spinlock rlock;
        #ifdef CONFIG_DEBUG_LOCK_A



        LLOC
        # define LOCK_PADSIZE (offsetof(struct raw_spinlock, dep_map))
        struct{
            u8 __padding[LOCK_PADSIZE];
            struct lockdep_map dep_map;
        };
        #endif
    };
}spinlock_t;
```

在使用自旋锁之前，肯定要先定义一个自旋锁变量，定义方法如下所示：

```
spinlock_t lock; //定义自旋锁
```

3.2、自旋锁API 函数：

函数	描述
DEFINE_SPINLOCK(spinlock_t lock)	定义并初始化一个自选变量。
int spin_lock_init(spinlock_t *lock)	初始化自旋锁。
void spin_lock(spinlock_t *lock)	获取指定的自旋锁，也叫做加锁。
void spin_unlock(spinlock_t *lock)	释放指定的自旋锁。
int spin_trylock(spinlock_t *lock)	尝试获取指定的自旋锁，如果没有获取到就返回 0
int spin_is_locked(spinlock_t *lock)	检查指定的自旋锁是否被获取，如果没有被获取就返回非 0，否则返回 0。

表 47.3.2.1 自旋锁基本 API 函数表

- 被自旋锁保护的临界区一定不能调用任何能够引起睡眠和阻塞的API 函数，否则的话会可能会导致死锁现象的发生。。如果线程A 在持有锁期间进入了休眠状态，那么线程A 会自动放弃CPU 使用权。线程B 开始运行，线程B 也想要获取锁，但是此时锁被A 线程持有，而且内核抢占还被禁止了！线程B 无法被调度出去，那么线程A 就无法运行，锁也就无法释放，好了，死锁发生了！

- 获取锁之前，要先关闭本地中断，