当两个进程同时进入一段代码时会产生并发，并发情况下会出现进程竞争资源，如下代码

If(a < 0){ a = 0; }

Else{ a++; }

如果A、B进程同时进入if内，那么a等于0而不是1

如果某个代码段不允许并发，那么我们称这段代码为临界区

1. **信号量**

进程进入临界区时必须要获取信号，进程退出临界区时必须释放信号，头文件<asm/semaphore.h>.

**创建信号量**

struct semaphore sem;

void sema\_init(struct semaphore \*sem, int val);

Val：信号量初始值（可以理解为信号的数量）

**创建互斥体**

互斥体是一种特殊的信号量，其值为1，如下宏定义并初始化互斥体

DECLARE\_MUTEX(name);

DECLARE\_MUTEX\_LOCKED(name);

初始化为 1 ( 使用 DECLARE\_MUTEX )

初始值为 0 (使用 DECLARE\_MUTEX\_LOCKED ).

动态初始化使用如下：

void init\_MUTEX(struct semaphore \*sem);

void init\_MUTEX\_LOCKED(struct semaphore \*sem);

**获取信号量**

void down(struct semaphore \*sem);

int down\_interruptible(struct semaphore \*sem);

int down\_trylock(struct semaphore \*sem);

down 递减旗标值并且等待需要的时间.

down\_interruptible 同上, 但是操作是可中断的.

down\_tryloc 从不睡眠; 如果没有信号量, down\_trylock 立刻返回一个非零值.

以上操作成功后，信号量的值减1，如果信号量为0表示没有信号

**释放信号量**

void up(struct semaphore \*sem);

1. **读写信号量**

有时候我们可以允许并发多个读操作，但不允许并发读和写、写和写，头文件<linux/rwsem.h>

**初始化读写信号量**

void init\_rwsem(struct rw\_semaphore \*sem);

**获取读信号和释放读信号**

void down\_read(struct rw\_semaphore \*sem);

int down\_read\_trylock(struct rw\_semaphore \*sem);

void up\_read(struct rw\_semaphore \*sem);

**获取和释放写信号**

void down\_write(struct rw\_semaphore \*sem);

int down\_write\_trylock(struct rw\_semaphore \*sem);

void up\_write(struct rw\_semaphore \*sem);

void downgrade\_write(struct rw\_semaphore \*sem);

1. **Completions 机制**

在一些情况下，我们一个或多个线程会等待另一个线程完成某个任何后才接着往下执行，头文件<linux/completion.h>

**创建completion**

使用宏:

DECLARE\_COMPLETION(my\_completion);

如果 completion 必须动态创建和初始化:

struct completion my\_completion;

init\_completion(&my\_completion);

**等待completion**

void wait\_for\_completion(struct completion \*c);

**触发completion**

void complete(struct completion \*c);

void complete\_all(struct completion \*c);

complete 只唤醒一个等待的线程

complete\_all 唤醒所有，使用complete\_all后必须重新初始化completion才能再次使用

1. **自旋锁**

信号量在进程未获取到信号时会进入休眠，但自旋锁在进程未获取到锁时会一直循环尝试获取锁

驱动程序持有自旋锁时，应知道如下规则

程序不应该休眠

程序持有自旋锁时，内核会禁止内核抢占

程序持有自旋锁时，内核会禁止中断

程序应该尽可能短时间持有自旋锁

**自旋锁的使用**

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);

为释放一个你已获得的锁, 传递它给:

void spin\_unlock(spinlock\_t \*lock);

**自旋锁的其他函数**

void spin\_lock(spinlock\_t \*lock);

void spin\_lock\_irqsave(spinlock\_t \*lock, unsigned long flags);

void spin\_lock\_irq(spinlock\_t \*lock);

void spin\_lock\_bh(spinlock\_t \*lock)

1. **原子变量**

原子变量和其操作是原子操作，因此不需要加锁，头文件<asm/atomic.h>

原子变量保存的是个int值

**创建原子变量**

atomic\_t v = ATOMIC\_INIT(0);

**读取原子变量**

int atomic\_read(atomic\_t \*v);

**原子变量加i**

void atomic\_add(int i, atomic\_t \*v);

**原子变量减i**

void atomic\_sub(int i, atomic\_t \*v);

1. **位操作**

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); 清除第 nr 位在 addr 指向的数据项中.

void change\_bit(nr, void \*addr); 翻转这个位.

test\_bit(nr, void \*addr); 它简单地返回nr位的值.