## 定义

如果这个操作所处的层(layer)的更高层不能发现其内部实现与结构，那么这个操作是一个原子(atomic)操作。

原子操作可以是一个步骤，也可以是多个操作步骤，但是其顺序不可以被打乱，也不可以被切割而只执行其中的一部分。

将整个操作视作一个整体是原子性的核心特征。

## 简介

在[多进程](https://baike.baidu.com/item/%E5%A4%9A%E8%BF%9B%E7%A8%8B" \t "_blank)（[线程](https://baike.baidu.com/item/%E7%BA%BF%E7%A8%8B)）访问共享资源时，能够确保所有其他的进程（线程）都不在同一时间内访问相同的资源。原子操作（atomic operation）是不需要synchronized，这是Java多线程编程的老生常谈了。所谓原子操作是指不会被[线程调度](https://baike.baidu.com/item/%E7%BA%BF%E7%A8%8B%E8%B0%83%E5%BA%A6)机制打断的操作；这种操作一旦开始，就一直运行到结束，中间不会有任何 context switch （切换到另一个线程）。通常所说的原子操作包括对非long和double型的primitive进行赋值，以及返回这两者之外的primitive。之所以要把它们排除在外是因为它们都比较大，而JVM的设计规范又没有要求读操作和赋值操作必须是原子操作（JVM可以试着去这么作，但并不保证）。

首先处理器会自动保证基本的内存操作的原子性。处理器保证从系统内存当中读取或者写入一个字节是原子的，意思是当一个处理器读取一个字节时，其他处理器不能访问这个字节的内存地址。奔腾6和最新的处理器能自动保证单处理器对同一个缓存行里进行16/32/64位的操作是原子的，但是复杂的内存操作处理器不能自动保证其原子性，比如跨总线宽度，跨多个缓存行，跨页表的访问。但是处理器提供总线锁定和缓存锁定两个机制来保证复杂内存操作的原子性。[1]

## 特性

原子操作：UP和SMP的异同  
原子操作是不可分割的，在执行完毕不会被任何其它任务或事件中断。在单处理器系统（UniProcessor）中，能够在单条指令中完成的操作都可以认为是"原子操作"，因为中断只能发生于指令之间。这也是某些CPU指令系统中引入了test\_and\_set、test\_and\_clear等指令用于临界资源互斥的原因。但是，在对称多处理器（Symetric Multi-Processor）结构中就不同了，由于系统中有多个处理器在独立地运行，即使能在单条指令中完成的操作也有可能受到干扰。我们以decl（递减指令）为例，这是一个典型的"读－改－写"过程，涉及两次内存访问。设想在不同CPU运行的两个进程都在递减某个计数值，可能发生的情况是：  
1． CPU A（上的进程，以下同）从内存单元把当前计数值（2）装载进它的寄存器中；  
2． CPU B从内存单元把当前计数值（2）装载进它的寄存器中。  
3． CPU A在它的寄存器中将计数值递减为1；  
4． CPU B在它的寄存器中将计数值递减为1；  
5． CPU A把修改后的计数值（1）写回内存单元。  
6． CPU B把修改后的计数值（1）写回内存单元。  
我们看到，内存里的计数值应该是0，然而它却是1。如果该计数值是一个共享资源的引用计数，每个进程都在递减后把该值与0进行比较，从而确定是否需要释放该共享资源。这时，两个进程都去掉了对该共享资源的引用，但没有一个进程能够释放它--两个进程都推断出：计数值是1，共享资源仍然在被使用。  
原子性不可能由软件单独保证--必须需要硬件的支持，因此是和架构相关的。在x86平台上，CPU提供了在指令执行期间对总线加锁的手段。CPU芯片上有一条引线#HLOCK pin，如果汇编语言的程序中在一条指令前面加上前缀"LOCK"，经过汇编以后的机器代码就使CPU在执行这条指令的时候把#HLOCK pin的电位拉低，持续到这条指令结束时放开，从而把总线锁住，这样同一总线上别的CPU就暂时不能通过总线访问内存了，保证了这条指令在多处理器环境中的原子性。  
Linux内核中的原子操作  
Linux 2.4.21中，原子类型的定义和原子操作API都放在内核源码树的include/asm/atomic.h文件中，大部分使用汇编语言实现，因为c语言并不能实现这样的操作。  
在x86的原子操作实现代码中，定义了LOCK宏，这个宏可以放在随后的内联汇编指令之前。如果是SMP，LOCK宏被扩展为lock指令；否则被定义为空--单CPU无需防止其它CPU的干扰，锁内存总线完全是在浪费时间。  
#ifdef CONFIG\_SMP  
#define LOCK "lock ; "  
#else  
#define LOCK ""  
#endif  
typedef struct { volatile int counter; } atomic\_t;  
在所有支持的体系结构上原子类型atomic\_t都保存一个int值。在x86的某些处理器上，由于工作方式的原因，原子类型能够保证的可用范围只有24位。volatile是一个类型描述符，要求编译器不要对其描述的对象作优化处理，对它的读写都需要从内存中访问。  
#define ATOMIC\_INIT(i) { (i) }  
用于在定义原子变量时，初始化为指定的值。如：  
static atomic\_t count = ATOMIC\_INIT(1);  
#define atomic\_read(v) ((v)->counter)  
读取v指向的原子变量的值。由于该操作本身就是原子的，只需要一次内存访问就能完成，因此定义为一个宏，并用C代码实现。  
#define atomic\_set(v,i) (((v)->counter) = (i))  
设置v指向的原子变量为i。由于该操作本身就是原子的，只需要一次内存访问就能完成，因此定义为一个宏，并用C代码实现。  
static \_\_inline\_\_ void atomic\_add(int i, atomic\_t \*v)  
将v指向的原子变量加上i。该函数不关心原子变量的新值，返回void类型。在下面的实现中，使用了带有C/C++表达式的内联汇编代码，格式如下（参考《AT&T ASM Syntax》）：  
\_\_asm\_\_　\_\_volatile\_\_("Instruction List" : Output : Input : Clobber/Modify);  
\_\_asm\_\_ \_\_volatile\_\_指示编译器原封不动保留表达式中的汇编指令系列，不要考虑优化处理。涉及的约束还包括：  
1. 等号约束（=）：只能用于输出操作表达式约束，说明括号内的左值表达式v->counter是write-only的。  
2. 内存约束（m）：表示使用不需要借助寄存器，直接使用内存方式进行输入或输出。  
3. 立即数约束（i）：表示输入表达式是一个立即数（整数），不需要借助任何寄存器。  
4. 寄存器约束（r）：表示使用一个通用寄存器，由GCC在%eax/%ax/%al、%ebx/%bx/%bl、%ecx/%cx/%cl和%edx/%dx/%dl中选取一个合适的。  
{  
\_\_asm\_\_ \_\_volatile\_\_(  
    LOCK "addl %1,%0"  
    :"=m" (v->counter)  
    :"ir" (i), "m" (v->counter));  
}  
static \_\_inline\_\_ void atomic\_sub(int i, atomic\_t \*v)  
从v指向的原子变量减去i。   
static \_\_inline\_\_ int atomic\_sub\_and\_test(int i, atomic\_t \*v)  
从v指向的原子变量减去i，并测试是否为0。若为0，返回真，否则返回假。由于x86的subl指令会在结果为0时设置CPU的zero标志位，而且这个标志位是CPU私有的，不会被其它CPU影响。因此，可以执行一次加锁的减操作，再根据CPU的zero标志位来设置本地变量c，并相应返回。  
{  
unsigned char c;  
\_\_asm\_\_ \_\_volatile\_\_(  
    LOCK "subl %2,%0; sete %1"  
    :"=m" (v->counter), "=qm" (c)  
    :"ir" (i), "m" (v->counter) : "memory");  
return c;  
}  
static \_\_inline\_\_ void atomic\_inc(atomic\_t \*v)  
递增v指向的原子变量。  
static \_\_inline\_\_ void atomic\_dec(atomic\_t \*v)  
递减v指向的原子变量。  
static \_\_inline\_\_ int atomic\_dec\_and\_test(atomic\_t \*v)  
递减v指向的原子变量，并测试是否为0。若为0，返回真，否则返回假。  
static \_\_inline\_\_ int atomic\_inc\_and\_test(atomic\_t \*v)  
递增v指向的原子变量，并测试是否为0。若为0，返回真，否则返回假。  
static \_\_inline\_\_ int atomic\_add\_negative(int i, atomic\_t \*v)  
将v指向的原子变量加上i，并测试结果是否为负。若为负，返回真，否则返回假。这个操作用于实现semaphore。  
原子操作的应用  
在Linux内核中，原子操作被大量用于实现资源的引用计数。待补充...  
参考文献：  
1. Linux内核注释  
2. LINUX内核源代码情景分析 

Linux内核的同步机制（二）：自旋锁   
  
自旋锁：单处理器非抢占式内核和对称多处理器或抢占式内核  
Linux 2.4.x及以前的版本都是非抢占式内核方式，如果编译成单处理器系统，在同一时间只有一个进程在执行，除非它自己放弃，不然只有通过"中断"才能中断其执行。因此，在单处理器非抢占式内核中，如果需要修改某个重要的数据结构，或者执行某些关键代码，只需要禁止中断。  
但是在对称多处理器，仅仅禁止某个CPU的中断是不够的，当然我们也可以将所有CPU的中断都禁止，但这样做开销很大，整个系统的性能会明显下降。此外，即使在单处理器上，如果内核是抢占式的，也可能出现不同进程上下文同时进入临界区的情况。为此，Linux内核中提供了"自旋锁（spinlock）"的同步机制。   
自旋锁与互斥锁有点类似，只是自旋锁不会引起调用者睡眠，如果自旋锁已经被别的执行单元保持，调用者就一直循环在那里看是否该自旋锁的保持者已经释放了锁，"自旋"因此而得名。  
因此，中断（或软中断）禁止用于防止同一CPU上中断（或软中断）对共享资源的非同步访问。而自旋锁则防止在不同CPU上的执行单元对共享资源的同时访问，以及不同进程上下文互相抢占导致的对共享资源的非同步访问。  
我们Linux 2.4.21为基础，分析x86平台下自旋锁的类型及应用方式，相关代码在源代码树的include/linux/spinlock.h以及include/asm-i386/spinlock.h中。  
Linux内核中的自旋锁  
在Linux内核中，自旋锁的基本使用方式如下：  
先声明一个spinlock\_t类型的自旋锁变量，并初始化为"未加锁"状态。在进入临界区之前，调用加锁函数获得锁，在退出临界区之前，调用解锁函数释放锁。例如：  
spinlock\_t lock = SPIN\_LOCK\_UNLOCKED;  
spin\_lock(&lock);  
/\* 临界区 \*/  
spin\_unlock(&lock);  
获得自旋锁和释放自旋锁的函数有多种变体。  
spin\_lock\_irqsave/spin\_unlock\_irqrestore  
相对于自旋锁的其它函数组，这一组函数是最"安全"的，使用频率也最多。在调用spin\_lock\_irqsave之前，我们还需要声明一个unsign long类型的变量（例如flag），该函数可以顺序完成下列操作：  
1. 将CPU的标志寄存器的内容保存在变量flag中；  
2. 禁止CPU的本地中断；  
3. 调用spin\_lock获得自旋锁。  
而spin\_unlock\_irqrestore函数则在调用spin\_unlock释放自旋锁之后，将变量flag保存的值恢复到CPU的标志寄存器中。  
#define spin\_lock\_irqsave(lock, flags)    do { local\_irq\_save(flags);         spin\_lock(lock); } while (0)  
#define local\_irq\_save(x) \_\_save\_and\_cli(x)  
#define \_\_save\_and\_cli(x) do { \_\_save\_flags(x); \_\_cli(); } while(0);  
#define \_\_save\_flags(x)    \_\_asm\_\_ \_\_volatile\_\_("pushfl ; popl %0":"=g" (x): /\* no input \*/)  
保存CPU的标志寄存器方法是：首先调用pushfl将标志寄存器压栈，再调用popl从栈中弹出保存在变量参数中。  
#define \_\_cli()     \_\_asm\_\_ \_\_volatile\_\_("cli": : :"memory")  
禁止CPU的本地中断使用cli汇编指令。  
#define spin\_unlock\_irqrestore(lock, flags) do { spin\_unlock(lock);    local\_irq\_restore(flags); } while (0)  
#define local\_irq\_restore(x) \_\_restore\_flags(x)  
#define \_\_restore\_flags(x)    \_\_asm\_\_ \_\_volatile\_\_("pushl %0 ; popfl": /\* no output \*/ :"g" (x):"memory", "cc")  
恢复CPU的标志寄存器方法是：首先调用pushl将变量参数压到栈中，再调用popfl从栈中弹出保存到标志寄存器。  
需要注意的是，这里没有显式执行开中断的动作。实际上，在标志寄存器中保持了原来的中断状态，在恢复寄存器的同时将中断也恢复到以前的状态。  
spin\_lock\_irq/spin\_unlock\_irq  
和上面一组函数的不同在于，这一组函数并不涉及标志寄存器。spin\_lock\_irq函数首先禁止CPU的本地中断，再调用spin\_lock获得自旋锁。而spin\_unlock\_irq函数则首先调用spin\_unlock释放自旋锁，再打开CPU的本地中断。  
#define spin\_lock\_irq(lock)     do { local\_irq\_disable();           spin\_lock(lock); } while (0)  
#define local\_irq\_disable() \_\_cli()  
#define spin\_unlock\_irq(lock)     do { spin\_unlock(lock);    local\_irq\_enable();         } while (0)  
#define local\_irq\_enable() \_\_sti()  
#define \_\_sti()     \_\_asm\_\_ \_\_volatile\_\_("sti": : :"memory")  
打开CPU的本地中断使用sti汇编指令。  
spin\_lock\_bh/spin\_unlock\_bh/spin\_trylock\_bh  
spin\_lock\_bh函数在得到自旋锁的同时禁止本地软中断，spin\_unlock\_bh函数释放自旋锁的同时，也打开本地的软中断。  
#define spin\_lock\_bh(lock)     do { local\_bh\_disable();            spin\_lock(lock); } while (0)  
#define local\_bh\_disable() cpu\_bh\_disable(smp\_processor\_id())  
#define cpu\_bh\_disable(cpu) /  
    do { local\_bh\_count(cpu)++; barrier(); } while (0)  
#define spin\_unlock\_bh(lock)     do { spin\_unlock(lock);    local\_bh\_enable();          } while (0)  
#define spin\_trylock\_bh(lock)     ({ int \_\_r; local\_bh\_disable();/  
\_\_r = spin\_trylock(lock);        /  
if (!\_\_r) local\_bh\_enable();     /  
\_\_r; })  
spin\_lock/spin\_unlock/spin\_trylock  
上面各组函数最终都需要调用自旋锁操作函数。spin\_lock函数用于获得自旋锁，如果能够立即获得锁，它就马上返回，否则，它将自旋在那里，直到该自旋锁的保持者释放。spin\_unlock函数则释放自旋锁。此外，还有一个spin\_trylock函数。尽力获得自旋锁lock，如果能立即获得锁，它获得锁并返回真，否则不能立即获得锁，立即返回假。它不会自旋等待lock被释放。  
spin\_lock/spin\_unlock/spin\_trylock在UP环境下  
由于Linux 2.4.x为不可抢占的内核，在单处理器环境下，自旋锁什么都不需要做。自旋锁类型spinlock\_t设置为空，除了在GCC的早期版本中不支持内容为空的数据结构。  
typedef struct {  
} spinlock\_t;  
相应地，自旋锁的操作函数不进行任何实质性处理。  
#define spin\_lock\_init(lock) do { } while(0)  
#define spin\_lock(lock)    (void)(lock) /\* Not "unused variable". \*/  
#define spin\_is\_locked(lock) (0)  
#define spin\_trylock(lock) ({1; })  
#define spin\_unlock\_wait(lock) do { } while(0)  
#define spin\_unlock(lock) do { } while(0)  
spin\_lock/spin\_unlock/spin\_trylock在SMP环境下  
在SMP环境下，自旋锁类型spinlock\_t含有一个unsigned int域lock。未加锁时值为1，加锁后值为0或负值。声明时也使用了volatile描述符，要求编译器不要对其作优化处理，对它的读写都需要从内存中访问。  
typedef struct {  
      volatile unsigned int lock;  
} spinlock\_t;  
static inline void spin\_lock(spinlock\_t \*lock)  
{  
      \_\_asm\_\_ \_\_volatile\_\_(  
          spin\_lock\_string  
          :"=m" (lock->lock) : : "memory");  
}  
从上看到，spin\_lock的主要内容是spin\_lock\_string宏，定义如下：  
#define spin\_lock\_string /  
      "/n1:/t" /  
      "lock ; decb %0/n/t" /  
      "js 2f/n" /  
      LOCK\_SECTION\_START("") /  
      "2:/t" /  
      "cmpb $0,%0/n/t" /  
      "rep;nop/n/t" /  
      "jle 2b/n/t" /  
      "jmp 1b/n" /  
      LOCK\_SECTION\_END  
在上面的汇编代码中，首先将lock递减，如果为负（表明锁已被持有），则进入自旋。每次自旋在执行一条空指令后，根据lock和0的比较结果进行跳转。如果lock小于或等于0，则继续自旋；否则（说明锁已被释放），则将lock递减后返回。  
在这里，采取了一种优化手段，因为在大多数情况下，自旋锁是能成功获取的，而自旋部分代码，只是在锁被持有时才执行，因此利用LOCK\_SECTION\_START和LOCK\_SECTION\_END将这些代码放到一个专门的区（.text.lock）中。如果把它跟别的常用指令混在一起，会浪费指令缓存的空间。  
理解这一点，我们也就能明白spin\_lock是如何退出的了。事实上，由于不再同一个区（section），所以"js 2f"的下一条指令并不是"cmpb $0,%0"。  
#define LOCK\_SECTION\_NAME     /  
      ".text.lock." \_\_stringify(KBUILD\_BASENAME)  
#define LOCK\_SECTION\_START(extra)    /  
      ".subsection 1/n/t"     /  
      extra       /  
      ".ifndef " LOCK\_SECTION\_NAME "/n/t" /  
      LOCK\_SECTION\_NAME ":/n/t"    /  
      ".endif/n/t"  
#define LOCK\_SECTION\_END     /  
      ".previous/n/t"  
如果用类C语言来描述上述过程，将是：  
void spin\_lock(int &lock)  
{  
      int flag;

label1:  
      flag = lock--;  
      if (flag >= 0) {  
          return;  
      } else {  
          do {  
              nop;  
          } while(lock <= 0);  
          goto label1;  
      }  
}  
当然，我们需要保证lock--为原子操作，就像在汇编中在decb指令前加上lock前缀一样。释放锁时使用spin\_unlock\_string宏，定义如下：  
static inline void spin\_unlock(spinlock\_t \*lock)  
{  
      \_\_asm\_\_ \_\_volatile\_\_(  
          spin\_unlock\_string  
      );  
}  
#define spin\_unlock\_string /  
      "movb $1,%0" /  
      :"=m" (lock->lock) : : "memory"  
需要强调的是，为了和spin\_lock相配合，我们在释放锁是必须将lock设置为1，而不能使用lock++的方式。  
#define SPIN\_LOCK\_UNLOCKED (spinlock\_t) { 1 SPINLOCK\_MAGIC\_INIT }  
#define spin\_lock\_init(x) do { \*(x) = SPIN\_LOCK\_UNLOCKED; } while(0)  
#define spin\_is\_locked(x) (\*(volatile signed char \*)(&(x)->lock) <= 0)  
#define spin\_unlock\_wait(x) do { barrier(); } while(spin\_is\_locked(x))  
由于spin\_trylock不需要自旋，实现中采用xchgb指令（该指令将自动锁总线，而不需要再使用lock前缀）。若lock原来的值为1，说明未上锁，因此返回为真，表明成功获得锁；否则返回为假。  
static inline int spin\_trylock(spinlock\_t \*lock)  
{  
      char oldval;  
      \_\_asm\_\_ \_\_volatile\_\_(  
          "xchgb %b0,%1"  
          :"=q" (oldval), "=m" (lock->lock)  
          :"0" (0) : "memory");  
          return oldval > 0;  
}  
自旋锁的应用  
在讨论自旋锁的应用时，我们一般区分两种平台：单处理器非抢占式内核和对称多处理器或抢占式内核。在前面我们看到，在单处理器非抢占式内核下，自旋锁根本不存在。这体现了一种出色的设计策略，既然没有别人能够同时刻执行，就没有理由加锁。对于抢占式内核，我们将它等同于对称多处理器来考虑。  
1． 用户上下文之间  
如果数据结构只可能被用户上下文访问，最高效的办法就是使用信号量。（我们在后面将讨论信号量机制）。  
2． 用户上下文与softirq之间  
这种情况下，使用spin\_lock\_bh()/spin\_unlock\_bh()可以满足要求。如果是单处理器非抢占式内核，自旋锁消失了，spin\_lock\_bh等同于local\_bh\_disable，会进制在用户上下文时进制softirq，从而避免用户上下文和softirq同时进入临界区。如果是对称多处理器或者抢占式内核，即使是在不同CPU上的用户上下文和softirq同时运行，自旋锁机制保证了只有一个持有者，只有在它释放锁之后，另一个才能进入临界区。  
3． 用户上下文和Tasklet/Timer之间  
同上。同加锁观点来看，Tasklet和Timer的地位是同样的。  
4． Tasklet或Timer之间  
这里有两点需要说明：（1）同一时刻，一个Tasklet或Timer不会同时在两个CPU上执行；（2）如果CPU已经处在Tasklet或Timer中，它不会同时再执行其它的Tasklet或Timer。因此我们只需要考虑在不同CPU上运行两个不同Tasklet或Timer的情况，而这种情况只需要使用自旋锁机制，即spin\_lock和spin\_unlock函数。  
5． Softirq之间或和Tasklet/Timer之间  
同一个softirq可能在不同的CPU上执行，同上道理，使用spin\_lock和spin\_unlock可以在不同CPU的同一个或不同softirq，或者Softirq与Tasklet或Timer之间保护共享数据。  
6． 硬件中断之间  
如果被保护的共享资源在软中断（包括tasklet和timer）或进程上下文和硬中断上下文访问，那么在软中断或进程上下文访问期间，可能被硬中断打断，从而进入硬中断上下文对共享资源进行访问，因此，在进程或软中断上下文需要使用spin\_lock\_irq和spin\_unlock\_irq来保护对共享资源的访问。   
而在中断处理句柄中使用什么版本，需依情况而定，如果只有一个中断处理句柄访问该共享资源，那么在中断处理句柄中仅需要spin\_lock和spin\_unlock来保护对共享资源的访问就可以了。   
因为在执行中断处理句柄期间，不可能被同一CPU上的软中断或进程打断。但是如果有不同的中断处理句柄访问该共享资源，那么需要在中断处理句柄中使用spin\_lock\_irq和spin\_unlock\_irq来保护对共享资源的访问。  
在使用spin\_lock\_irq和spin\_unlock\_irq的情况下，完全可以用spin\_lock\_irqsave和spin\_unlock\_irqrestore取代，具体应该使用哪一个也需要依情况而定，如果可以确信在对共享资源访问前中断是打开的，那么使用spin\_lock\_irq更好一些，因为它比spin\_lock\_irqsave要快一些。  
但是如果不能确定是否中断使能，那么使用spin\_lock\_irqsave和spin\_unlock\_irqrestore更好，因为它将恢复访问共享资源前的中断标志而不是直接打开中断。 当然，有些情况下需要在访问共享资源时必须禁止中断，而访问完后必须打开中断，这样的情形使用spin\_lock\_irq和spin\_unlock\_irq最好。

Linux内核的同步机制（三）：等待队列   
  
Linux内核中的等待队列  
Linux内核的等待队列是以双循环链表为基础数据结构，与进程调度机制紧密结合，能够用于实现核心的异步事件通知机制。在Linux2.4.21中，等待队列在源代码树include/linux/wait.h中，这是一个通过list\_head连接的典型双循环链表，如下图所示。

在这个链表中，有两种数据结构：等待队列头（wait\_queue\_head\_t）和等待队列项（wait\_queue\_t）。等待队列头和等待队列项中都包含一个list\_head类型的域作为"连接件"。由于我们只需要对队列进行添加和删除操作，并不会修改其中的对象（等待队列项），因此，我们只需要提供一把保护整个基础设施和所有对象的锁，这把锁保存在等待队列头中，为wq\_lock\_t类型。在实现中，可以支持读写锁（rwlock）或自旋锁（spinlock）两种类型，通过一个宏定义来切换。如果使用读写锁，将wq\_lock\_t定义为rwlock\_t类型；如果是自旋锁，将wq\_lock\_t定义为spinlock\_t类型。无论哪种情况，分别相应设置wq\_read\_lock、wq\_read\_unlock、wq\_read\_lock\_irqsave、wq\_read\_unlock\_irqrestore、wq\_write\_lock\_irq、wq\_write\_unlock、wq\_write\_lock\_irqsave和wq\_write\_unlock\_irqrestore等宏。  
等待队列头  
struct \_\_wait\_queue\_head {  
wq\_lock\_t lock;  
struct list\_head task\_list;  
};  
typedef struct \_\_wait\_queue\_head wait\_queue\_head\_t;  
前面已经说过，等待队列的主体是进程，这反映在每个等待队列项中，是一个任务结构指针（struct task\_struct \* task）。flags为该进程的等待标志，当前只支持互斥。  
等待队列项  
struct \_\_wait\_queue {  
unsigned int flags;  
#define WQ\_FLAG\_EXCLUSIVE 0x01  
struct task\_struct \* task;  
struct list\_head task\_list;  
};  
typedef struct \_\_wait\_queue wait\_queue\_t;  
声明和初始化  
#define DECLARE\_WAITQUEUE(name, tsk)       /  
wait\_queue\_t name = \_\_WAITQUEUE\_INITIALIZER(name, tsk)  
#define \_\_WAITQUEUE\_INITIALIZER(name, tsk) {      /  
task:    tsk,        /  
task\_list: { NULL, NULL },       /  
      \_\_WAITQUEUE\_DEBUG\_INIT(name)}  
通过DECLARE\_WAITQUEUE宏将等待队列项初始化成对应的任务结构，并且用于连接的相关指针均设置为空。其中加入了调试相关代码。  
#define DECLARE\_WAIT\_QUEUE\_HEAD(name) /  
wait\_queue\_head\_t name = \_\_WAIT\_QUEUE\_HEAD\_INITIALIZER(name)  
#define \_\_WAIT\_QUEUE\_HEAD\_INITIALIZER(name) {      /  
lock:    WAITQUEUE\_RW\_LOCK\_UNLOCKED,     /  
task\_list: { &(name).task\_list, &(name).task\_list }, /  
     \_\_WAITQUEUE\_HEAD\_DEBUG\_INIT(name)}  
通过DECLARE\_WAIT\_QUEUE\_HEAD宏初始化一个等待队列头，使得其所在链表为空，并设置链表为"未上锁"状态。其中加入了调试相关代码。  
static inline void init\_waitqueue\_head(wait\_queue\_head\_t \*q)  
该函数初始化一个已经存在的等待队列头，它将整个队列设置为"未上锁"状态，并将链表指针prev和next指向它自身。  
{  
      q->lock = WAITQUEUE\_RW\_LOCK\_UNLOCKED;  
      INIT\_LIST\_HEAD(&q->task\_list);  
}  
static inline void init\_waitqueue\_entry(wait\_queue\_t \*q, struct task\_struct \*p)  
该函数初始化一个已经存在的等待队列项，它设置对应的任务结构，同时将标志位清0。  
{  
      q->flags = 0;  
      q->task = p;  
}  
static inline int waitqueue\_active(wait\_queue\_head\_t \*q)  
该函数检查等待队列是否为空。  
{  
      return !list\_empty(&q->task\_list);  
}  
static inline void \_\_add\_wait\_queue(wait\_queue\_head\_t \*head, wait\_queue\_t \*new)  
将指定的等待队列项new添加到等待队列头head所在的链表头部，该函数假设已经获得锁。  
{  
      list\_add(&new->task\_list, &head->task\_list);  
}  
static inline void \_\_add\_wait\_queue\_tail(wait\_queue\_head\_t \*head, wait\_queue\_t \*new)  
将指定的等待队列项new添加到等待队列头head所在的链表尾部，该函数假设已经获得锁。  
{  
      list\_add\_tail(&new->task\_list, &head->task\_list);  
}  
static inline void \_\_remove\_wait\_queue(wait\_queue\_head\_t \*head, wait\_queue\_t \*old)  
将函数从等待队列头head所在的链表中删除指定等待队列项old，该函数假设已经获得锁，并且old在head所在链表中。  
{  
      list\_del(&old->task\_list);  
}  
睡眠和唤醒操作  
对等待队列的操作包括睡眠和唤醒（相关函数保存在源代码树的/kernel/sched.c和include/linux/sched.h中）。思想是更改当前进程（CURRENT）的任务状态，并要求重新调度，因为这时这个进程的状态已经改变，不再在调度表的就绪队列中，因此无法再获得执行机会，进入"睡眠"状态，直至被"唤醒"，即其任务状态重新被修改回就绪态。  
常用的睡眠操作有interruptible\_sleep\_on和sleep\_on。两个函数类似，只不过前者将进程的状态从就绪态（TASK\_RUNNING）设置为TASK\_INTERRUPTIBLE，允许通过发送signal唤醒它（即可中断的睡眠状态）；而后者将进程的状态设置为TASK\_UNINTERRUPTIBLE，在这种状态下，不接收任何singal。以interruptible\_sleep\_on为例，其展开后的代码是：  
void interruptible\_sleep\_on(wait\_queue\_head\_t \*q)  
{  
      unsigned long flags;  
      wait\_queue\_t wait;  
      /\* 构造当前进程对应的等待队列项 \*/  
      init\_waitqueue\_entry(&wait, current);   
      /\* 将当前进程的状态从TASK\_RUNNING改为TASK\_INTERRUPTIBLE \*/  
      current->state = TASK\_INTERRUPTIBLE;  
      /\* 将等待队列项添加到指定链表中 \*/  
      wq\_write\_lock\_irqsave(&q->lock,flags);  
      \_\_add\_wait\_queue(q, &wait);   
      wq\_write\_unlock(&q->lock);  
      /\* 进程重新调度，放弃执行权 \*/  
      schedule();  
      /\* 本进程被唤醒，重新获得执行权，首要之事是将等待队列项从链表中删除 \*/  
      wq\_write\_lock\_irq(&q->lock);  
      \_\_remove\_wait\_queue(q, &wait);  
      wq\_write\_unlock\_irqrestore(&q->lock,flags);  
      /\* 至此，等待过程结束，本进程可以正常执行下面的逻辑 \*/  
}  
对应的唤醒操作包括wake\_up\_interruptible和wake\_up。wake\_up函数不仅可以唤醒状态为TASK\_UNINTERRUPTIBLE的进程，而且可以唤醒状态为TASK\_INTERRUPTIBLE的进程。wake\_up\_interruptible只负责唤醒状态为TASK\_INTERRUPTIBLE的进程。这两个宏的定义如下：  
#define wake\_up(x)     \_\_wake\_up((x),TASK\_UNINTERRUPTIBLE | TASK\_INTERRUPTIBLE, 1)  
#define wake\_up\_interruptible(x) \_\_wake\_up((x),TASK\_INTERRUPTIBLE, 1)  
\_\_wake\_up函数主要是获取队列操作的锁，具体工作是调用\_\_wake\_up\_common完成的。  
void \_\_wake\_up(wait\_queue\_head\_t \*q, unsigned int mode, int nr)  
{  
      if (q) {  
          unsigned long flags;  
          wq\_read\_lock\_irqsave(&q->lock, flags);  
          \_\_wake\_up\_common(q, mode, nr, 0);  
          wq\_read\_unlock\_irqrestore(&q->lock, flags);  
      }  
}  
/\* The core wakeup function.    Non-exclusive wakeups (nr\_exclusive == 0) just wake everything up.    If it's an exclusive wakeup (nr\_exclusive == small +ve number) then we wake all the non-exclusive tasks and one exclusive task.  
There are circumstances in which we can try to wake a task which has already started to run but is not in state TASK\_RUNNING.    try\_to\_wake\_up() returns zero in this (rare) case, and we handle it by contonuing to scan the queue. \*/  
static inline void \_\_wake\_up\_common (wait\_queue\_head\_t \*q, unsigned int mode, int nr\_exclusive, const int sync)  
参数q表示要操作的等待队列，mode表示要唤醒任务的状态，如TASK\_UNINTERRUPTIBLE或TASK\_INTERRUPTIBLE等。nr\_exclusive是要唤醒的互斥进程数目，在这之前遇到的非互斥进程将被无条件唤醒。sync表示？？？  
{  
      struct list\_head \*tmp;  
      struct task\_struct \*p;  
      CHECK\_MAGIC\_WQHEAD(q);  
      WQ\_CHECK\_LIST\_HEAD(&q->task\_list);  
      /\* 遍历等待队列 \*/  
      list\_for\_each(tmp,&q->task\_list) {  
          unsigned int state;  
          /\* 获得当前等待队列项 \*/  
          wait\_queue\_t \*curr = list\_entry(tmp, wait\_queue\_t, task\_list);  
          CHECK\_MAGIC(curr->\_\_magic);  
          /\* 获得对应的进程 \*/  
          p = curr->task;  
          state = p->state;  
          /\* 如果我们需要处理这种状态的进程 \*/  
          if (state & mode) {  
              WQ\_NOTE\_WAKER(curr);  
              if (try\_to\_wake\_up(p, sync) && (curr->flags&WQ\_FLAG\_EXCLUSIVE) && !--nr\_exclusive)  
                  break;  
          }  
      }  
}  
/\* 唤醒一个进程，将它放到运行队列中，如果它还不在运行队列的话。"当前"进程总是在运行队列中的（except when the actual re-schedule is in progress)，and as such you're allowed to do the simpler "current->state = TASK\_RUNNING" to mark yourself runnable without the overhead of this. \*/  
static inline int try\_to\_wake\_up(struct task\_struct \* p, int synchronous)  
{  
      unsigned long flags;  
      int success = 0;  
      /\* 由于我们需要操作运行队列，必须获得对应的锁 \*/  
      spin\_lock\_irqsave(&runqueue\_lock, flags);  
      /\* 将进程状态设置为TASK\_RUNNING \*/  
      p->state = TASK\_RUNNING;  
      /\* 如果进程已经在运行队列中，释放锁退出 \*/  
      if (task\_on\_runqueue(p))  
          goto out;  
      /\* 否则将进程添加到运行队列中 \*/  
      add\_to\_runqueue(p);  
      /\* 如果设置了同步标志 \*/  
      if (!synchronous || !(p->cpus\_allowed & (1UL << smp\_processor\_id())))  
          reschedule\_idle(p);  
      /\* 唤醒成功，释放锁退出 \*/  
      success = 1;  
out:  
      spin\_unlock\_irqrestore(&runqueue\_lock, flags);  
      return success;  
}  
等待队列应用模式  
等待队列的的应用涉及两个进程，假设为A和B。A是资源的消费者，B是资源的生产者。A在消费的时候必须确保资源已经生产出来，为此定义一个资源等待队列。这个队列同时要被进程A和进程B使用，我们可以将它定义为一个全局变量。  
DECLARE\_WAIT\_QUEUE\_HEAD(rsc\_queue); /\* 全局变量 \*/  
在进程A中，执行逻辑如下：  
while (resource is unavaiable) {   
      interruptible\_sleep\_on( &wq );   
}  
consume\_resource();   
在进程B中，执行逻辑如下：  
produce\_resource();  
wake\_up\_interruptible( &wq ); 

Linux内核的同步机制（四）：completion   
  
在Linux内核中，completion是一种简单的同步机制，标志"things may proceed"。要使用completion，必须在文件中包含<linux/completion.h>，同时创建一个类型为struct completion的变量。这个变量可以静态地声明和初始化：  
DECLARE\_COMPLETION(my\_comp);  
或者动态初始化：  
struct completion my\_comp;  
init\_completion(&my\_comp);  
如果驱动程序要在执行后面操作之前等待某个过程的完成，它可以调用wait\_for\_completion ，以要完成的事件为参数：  
void wait\_for\_completion(struct completion \*comp);  
如果其它部分代码可以确定事件已经完成，可以调用下面两个函数之一来唤醒等待该事件的进程：  
void complete(struct completion \*comp);  
void complete\_all(struct completion \*comp); /\* Linux 2.5.x以上版本 \*/  
前一个函数将只唤醒一个等待进程，而后一个函数唤醒等待该事件的所以进程。由于completion的实现方式，即使complete在wait\_for\_competion之前调用，也可以正常工作。  
例如，在MD设备驱动程序实现中，有一个恢复线程md\_recovery\_thread。驱动程序通过md\_register\_thread和md\_unregister\_thread来注册和注销恢复线程。恢复线程的执行逻辑在md\_thread函数中，大致如下：  
int md\_thread(void \* arg)  
{  
      线程初始化;  
      while (运行) {  
          处理逻辑;  
          接收信号;  
      }  
      return 0;  
}  
md\_register\_thread将创建一个恢复线程，它必须在线程真正初始化结束之后才能返回该线程的指针。因此，其逻辑是：  
mdk\_thread\_t \*md\_register\_thread(void (\*run) (void \*), void \*data, const char \*name)  
{  
      mdk\_thread\_t \*thread;  
      ……  
      struct completion event;  
      /\* 为线程分配空间 \*/  
      thread = (mdk\_thread\_t \*) kmalloc (sizeof(mdk\_thread\_t), GFP\_KERNEL);  
      ……  
      init\_completion(&event);  
      ……  
      thread->event = &event;  
      /\* 创建内核线程 \*/  
      ret = kernel\_thread(md\_thread, thread, 0);  
      /\* 等待线程初始化结束 \*/  
      ……  
      wait\_for\_completion(&event);  
      /\* 返回线程指针 \*/  
      return thread;  
}  
而md\_unregister\_thread通过向线程发送SIGKILL信号注销恢复线程，它也需要在线程真正退出后才能释放线程所占用的内存。因此，其逻辑是：  
void md\_unregister\_thread(mdk\_thread\_t \*thread)  
{  
      struct completion event;  
      init\_completion(&event);  
      thread->event = &event;  
      ……  
      /\* 向线程发送SIGKILL信号终止其运行 \*/  
      md\_interrupt\_thread(thread);  
      /\* 等待线程退出 \*/  
      wait\_for\_completion(&event);  
      /\* 释放线程所占用的内存 \*/  
      kfree(thread);  
}  
如果考虑completion，md\_thread的逻辑是：  
int md\_thread(void \* arg)  
{  
      线程初始化;  
      complete(thread->event);   
      while (运行) {  
          处理逻辑;  
          接收信号;  
      }  
      complete(thread->event);   
      return 0;  
}  
需要说明的是，由于等待事件是在驱动程序和恢复线程中的一个共享资源，它必须是一个全局变量，或者如实现代码中，定义为一个局部变量，而将其指针放在恢复线程结构中。  
typedef struct mdk\_thread\_s {  
      ……  
      struct completion \*event;  
      ……  
} mdk\_thread\_t;