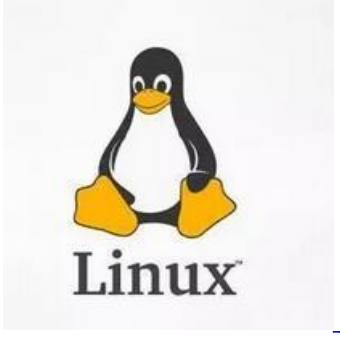
操作系统——数据同步问题

原创

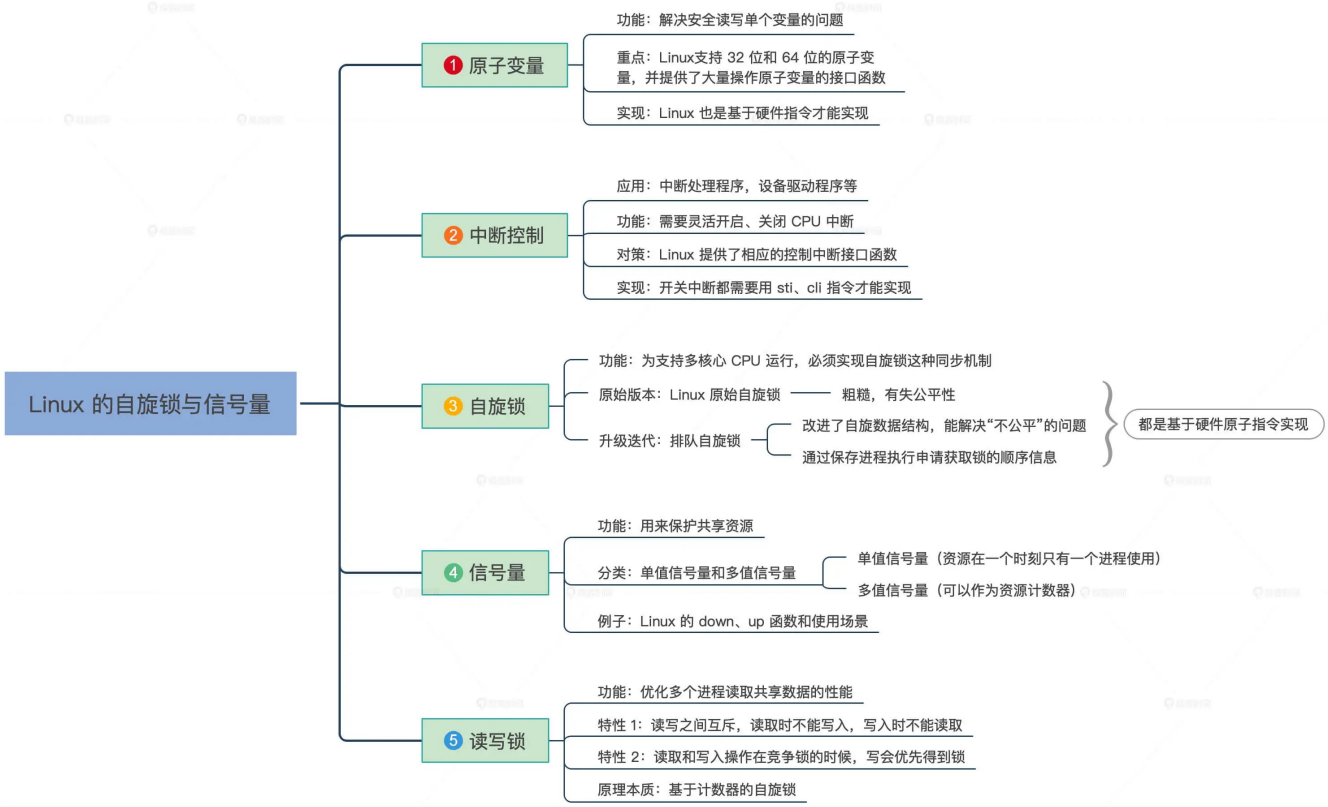
庄小焱 于 2021-08-26 10:19:17 发布 254 收藏 1
分类专栏： 操作系统 文章标签： 操作系统

本文链接：https://blog.csdn.net/weixin_41605937/article/details/119924486



摘要

操作系统对数据同步的方法有：对最重要的几种锁（原子变量，关中断，信号量，自旋锁）。



博客来源：极客时间操作系统45讲实战。

非预期结果的全局变量

```
int a = 0;
void interrupt_handle()
{
    a++;
}
void thread_func()
{
    a++;
}
```

首先我们梳理一下编译器的翻译过程，通常编译器会把 a++ 语句翻译成这 3 条指令。

- 1. 把 a 加载某个寄存器中。
- 2. 这个寄存器加 1。
- 3. 把这个寄存器写回内存。

那么不难推断，可能导致结果不确定的情况是这样的：thread_func 函数还没运行完第 2 条指令时，中断就来了。因此，CPU 转而处理中断，也就是开始运行 interrupt_handle 函数，这个函数运行完 a=1，CPU 还会回去继续运行第 3 条指令，此时 a 依然是 1，这显然是错的。显然在 t2 时刻发生了中断，导致了 t2 到 t4 运行了 interrupt_handle 函数，t5 时刻 thread_func 又恢复运行，导致 interrupt_handle 函数中 a 的操作丢失，因此出错。

原子操作

有这样两种思路。一种是把 a++ 变成原子操作，这里的原子是不可分隔的，也就是说要 a++ 这个操作不可分隔，即 a++ 要么不执行，要么一口气执行完；另一种就是控制中断，比如在执行 a++ 之前关掉中断，执行完了之后打开中断。

显然靠编译器自动生成原子操作不太可能。第一，编译器没有这么智能，能检测哪个变量需要原子操作；第二，编译器必须要考虑代码的移植性，例如有些硬件平台支持原子操作的机器指令，有的硬件平台不支持原子操作。既然实现原子操作无法依赖于具体编译器，那就需要我们自己动手，x86 平台支持很多原子指令，我们只需要直接应用这些指令，比如原子加、原子减，原子读写等，用汇编代码写出对应的原子操作函数就行了。

```
//定义一个原子类型
typedef struct s_ATOMIC{
    volatile s32_t a_count; //在变量前加上volatile，是为了禁止编译器优化，使其每次都从内存中加载变量
}atomic_t;
//原子读
static inline s32_t atomic_read(const atomic_t *v)
{
        //x86平台取地址处是原子
        return (*(volatile u32_t*)&(v)->a_count);
}
//原子写
static inline void atomic_write(atomic_t *v, int i)
{
        //x86平台把一个值写入一个地址处也是原子的
        v->a_count = i;
}
//原子加上一个整数
static inline void atomic_add(int i, atomic_t *v)
{
        __asm__ __volatile__("lock;" "addl %1,%0"
                        : "+m" (v->a_count)
                        : "ir" (i));
}
//原子减去一个整数
static inline void atomic_sub(int i, atomic_t *v)
{
        __asm__ __volatile__("lock;" "subl %1,%0"
                        : "+m" (v->a_count)
                        : "ir" (i));
}
//原子加1
static inline void atomic_inc(atomic_t *v)
{
        __asm__ __volatile__("lock;" "incl %0"
                         : "+m" (v->a_count));
}
//原子减1
static inline void atomic_dec(atomic_t *v)
{
       __asm__ __volatile__("lock;" "decl %0"
                        : "+m" (v->a_count));
}
```

以上代码中，加上 lock 前缀的 addl、subl、incl、decl 指令都是原子操作，lock 前缀表示锁定总线。

```
__asm__ __volatile__(代码部分:输出部分列表: 输入部分列表:损坏部分列表);
```

可以看到代码模板从 __asm__ 开始（当然也可以是 asm），紧跟着 __volatile__，然后是跟着一对括号，最后以分号结束。括号里大致分为 4 个部分：

1. 汇编代码部分，这里是实际嵌入的汇编代码。

2. 输出列表部分，让 GCC 能够处理 C 语言左值表达式与汇编代码的结合。

3. 输入列表部分，也是让 GCC 能够处理 C 语言表达式、变量、常量，让它们能够输入到汇编代码中去。

4. 损坏列表部分，告诉 GCC 汇编代码中用到了哪些寄存器，以便 GCC 在汇编代码运行前，生成保存它们的代码，并且在生成的汇编代码运行后，恢复它们（寄存器）的代码。

原子操作的例子

```
static inline void atomic_add(int i, atomic_t *v)
{
        __asm__ __volatile__("lock;" "addl %1,%0"
                     : "+m" (v->a_count)
                     : "ir" (i));
}
//"lock;" "addl %1,%0" 是汇编指令部分，%1,%0是占位符，它表示输出、输入列表中变量或表态式，占位符的数字从输出部分开始依
//: "+m" (v->a_count) 是输出列表部分，“+m”表示(v->a_count)和内存地址关联
//: "ir" (i) 是输入列表部分，“ir” 表示i是和立即数或者寄存器关联
```

有了这些原子操作函数之后 ，前面场景中的代码就变成下面这样了：无论有没有中断，或者什么时间来中断，都不会出错。

```
atomic_t a = {0};
void interrupt_handle()
{
    atomic_inc(&a);
}
void thread_func()
{
    atomic_inc(&a);
}
```

Linux的原子变量实现

类型变量 atomic_t

```
typedef struct {
    int counter;
} atomic_t;//常用的32位的原子变量类型
#ifdef CONFIG_64BIT
typedef struct {
    s64 counter;
} atomic64_t;//64位的原子变量类型
#endif
```

上述代码自然不能用普通的代码去读写加减，而是要用 Linux 专门提供的接口函数去操作，否则就不能保证原子性了，代码如下。

```
//原子读取变量中的值
static __always_inline int arch_atomic_read(const atomic_t *v)
{
    return __READ_ONCE((v)->counter);
}
//原子写入一个具体的值
static __always_inline void arch_atomic_set(atomic_t *v, int i)
{
    __WRITE_ONCE(v->counter, i);
}
//原子加上一个具体的值
static __always_inline void arch_atomic_add(int i, atomic_t *v)
{
    asm volatile(LOCK_PREFIX "addl %1,%0"
             : "+m" (v->counter)
             : "ir" (i) : "memory");
}
//原子减去一个具体的值
static __always_inline void arch_atomic_sub(int i, atomic_t *v)
{
    asm volatile(LOCK_PREFIX "subl %1,%0"
             : "+m" (v->counter)
             : "ir" (i) : "memory");
}
//原子加1
static __always_inline void arch_atomic_inc(atomic_t *v)
{
    asm volatile(LOCK_PREFIX "incl %0"
             : "+m" (v->counter) :: "memory");
}
//原子减1
static __always_inline void arch_atomic_dec(atomic_t *v)
{
    asm volatile(LOCK_PREFIX "decl %0"
             : "+m" (v->counter) :: "memory");
}
```

你会发现，Linux 的实现也同样采用了 x86 CPU 的原子指令，LOCK_PREFIX 是一个宏，根据需要展开成“lock;”或者空串。单核心 CPU 是不需要 lock 前缀的，只要在多核心 CPU 下才需要加上 lock 前缀。

剩下 __READ_ONCE，__WRITE_ONCE 两个宏，我们来看看它们分别做了什么，如下所示。

```
#define __READ_ONCE(x)  \
(*(const volatile __unqual_scalar_typeof(x) *)&(x))
#define __WRITE_ONCE(x, val) \
do {*(volatile typeof(x) *)&(x) = (val);} while (0)
//__unqual_scalar_typeof表示声明一个非限定的标量类型，非标量类型保持不变。说人话就是返回x变量的类型，这是GCC的功能，ty
//如果 x 是int类型则返回“int”
#define __READ_ONCE(x)  \
(*(const volatile int *)&(x))
#define __WRITE_ONCE(x, val) \
do {*(volatile int *)&(x) = (val);} while (0)
```

结合刚才的代码，我给你做个解读。Linux 定义了 __READ_ONCE，__WRITE_ONCE 这两个宏，是对代码封装并利用 GCC 的特性对代码进行检查，把让错误显现在编译阶段。其中的“volatile int *”是为了提醒编译器：这是对内存地址读写，不要有优化动作，每次都必须强制写入内存或从内存读取。

中断控制

中断是 CPU 响应外部事件的重要机制，时钟、键盘、硬盘等 IO 设备都是通过发出中断来请求 CPU 执行相关操作的（即执行相应的中断处理代码），比如下一个时钟到来、用户按下了键盘上的某个按键、硬盘已经准备好了数据。但是中断处理代码中如果操作了其它代码的数据，这就需要相应的控制机制了，这样才能保证在操作数据过程中不发生中断。你或许在想，可以用原子操作啊？不过，原子操作只适合于单体变量，如整数。操作系统的数据结构有的可能有几百字节大小，其中可能包含多种不同的基本数据类型。这显然用原子操作无法解决。

下面，我们就要写代码实现关闭开启、中断了，x86 CPU 上关闭、开启中断有专门的指令，即 cli、sti 指令，它们主要是对 CPU 的 eflags 寄存器的 IF 位（第 9 位）进行清除和设置，CPU 正是通过此位来决定是否响应中断信号。这两条指令只能 Ring0 权限才能执行，代码如下。

```
//关闭中断
void hal_cli()
{
    __asm__ __volatile__("cli": : :"memory");
}
//开启中断
void hal_sti()
{
    __asm__ __volatile__("sti": : :"memory");
}
//使用场景
void foo()
{
    hal_cli();
    //操作数据……
    hal_sti();
}
void bar()
{
    hal_cli();
    //操作数据……
    hal_sti();
}
```

它看似完美地解决了问题，其实有重大缺陷，hal_cli()，hal_sti()，无法嵌套使用，看一个例子你就明白了，代码如下。

```
void foo()
{
    hal_cli();
    //操作数据第一步……
    hal_sti();
}
void bar()
{
    hal_cli();
    foo();
    //操作数据第二步……
    hal_sti();
}
```

上面代码的关键问题在 bar 函数在关中断下调用了 foo 函数，foo 函数中先关掉中断，处理好数据然后开启中断，回到 bar 函数中，bar 函数还天真地以为中断是关闭的，接着处理数据，以为不会被中断抢占。

那么怎么解决上面的问题呢？我们只要修改一下开启、关闭中断的函数就行了。我们可以这样操作：在关闭中断函数中先保存 eflags 寄存器，然后执行 cli 指令，在开启中断函数中直接恢复之前保存的 eflags 寄存器就行了，具体代码如下。

```
typedef u32_t cpuflg_t;
static inline void hal_save_flags_cli(cpuflg_t* flags)
{
     __asm__ __volatile__(
            "pushfl \t\n" //把eflags寄存器压入当前栈顶
            "cli    \t\n" //关闭中断
            "popl %0 \t\n"//把当前栈顶弹出到eflags为地址的内存中
            : "=m"(*flags)
            :
            : "memory"
          );
}
static inline void hal_restore_flags_sti(cpuflg_t* flags)
{
    __asm__ __volatile__(
              "pushl %0 \t\n"//把flags为地址处的值寄存器压入当前栈顶
              "popfl \t\n"   //把当前栈顶弹出到eflags寄存器中
              :
              : "m"(*flags)
              : "memory"
              );
}
```

Linux中断实现

Linux 控制 CPU 响应中断的函数如下。

```
//实际保存eflags寄存器
extern __always_inline unsigned long native_save_fl(void){
    unsigned long flags;
    asm volatile("# __raw_save_flags\n\t"
                 "pushf ; pop %0":"=rm"(flags)::"memory");
```

```
    return flags;
}
//实际恢复eflags寄存器
extern inline void native_restore_fl(unsigned long flags){
    asm volatile("push %0 ; popf"::"g"(flags):"memory","cc");
}
//实际关中断
static __always_inline void native_irq_disable(void){
    asm volatile("cli":::"memory");
}
//实际开启中断
static __always_inline void native_irq_enable(void){
    asm volatile("sti":::"memory");
}
//arch层关中断
static __always_inline void arch_local_irq_disable(void){
    native_irq_disable();
}
//arch层开启中断
static __always_inline void arch_local_irq_enable(void){
    native_irq_enable();
}
//arch层保存eflags寄存器
static __always_inline unsigned long           arch_local_save_flags(void){
    return native_save_fl();
}
//arch层恢复eflags寄存器
static  __always_inline void arch_local_irq_restore(unsigned long flags){
    native_restore_fl(flags);
}
//实际保存eflags寄存器并关中断
static __always_inline unsigned long arch_local_irq_save(void){
    unsigned long flags = arch_local_save_flags();
    arch_local_irq_disable();
    return flags;
}
//raw层关闭开启中断宏
#define raw_local_irq_disable()     arch_local_irq_disable()
#define raw_local_irq_enable()      arch_local_irq_enable()
//raw层保存恢复eflags寄存器宏
#define raw_local_irq_save(flags)           \
    do {                        \
        typecheck(unsigned long, flags);    \
        flags = arch_local_irq_save();      \
    } while (0)

#define raw_local_irq_restore(flags)            \
    do {                        \
        typecheck(unsigned long, flags);    \
        arch_local_irq_restore(flags);      \
    } while (0)

#define raw_local_save_flags(flags)         \
    do {                        \
        typecheck(unsigned long, flags);    \
        flags = arch_local_save_flags();    \
    } while (0)
//通用层接口宏
#define local_irq_enable()              \
    do { \
```

```
        raw_local_irq_enable();         \
    } while (0)

#define local_irq_disable()             \
    do {                        \
        raw_local_irq_disable();        \
    } while (0)

#define local_irq_save(flags)               \
    do {                        \
        raw_local_irq_save(flags);      \
    } while (0)

#define local_irq_restore(flags)            \
    do {                        \
        raw_local_irq_restore(flags);       \
    } while (0)
```

编译 Linux 代码时，编译器自动对宏进行展开。其中，do{}while(0)是 Linux 代码中一种常用的技巧，do{}while(0) 表达式会保证{}中的代码片段执行一次，保证宏展开时这个代码片段是一个整体。带 native_ 前缀之类的函数则跟我们之前实现的 hal_ 前缀对应，而 Linux 为了支持不同的硬件平台，做了多层封装。

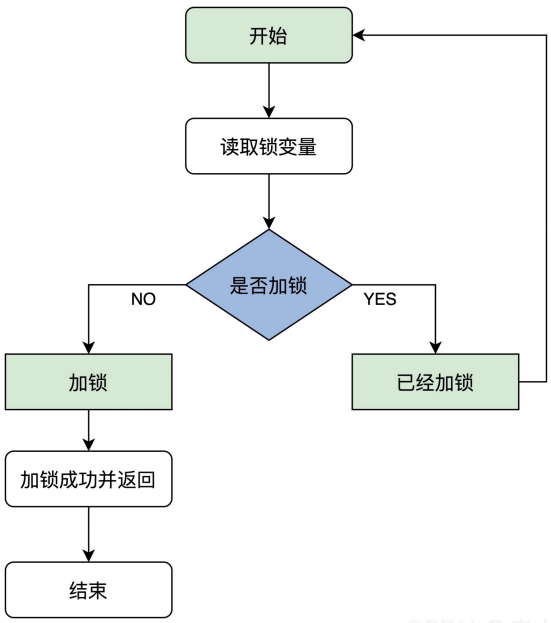
自旋锁的原理

那是因为以前是单 CPU，同一时刻只有一条代码执行流，除了中断会中止当前代码执行流，转而运行另一条代码执行流（中断处理程序），再无其它代码执行流。这种情况下只要控制了中断，就能安全地操作全局数据。

但是我们都知道，现在情况发生了改变，CPU 变成了多核心，或者主板上安装了多颗 CPU，同一时刻下系统中存在多条代码执行流，控制中断只能控制本地 CPU 的中断，无法控制其它 CPU 核心的中断。

所以，原先通过控制中断来维护全局数据安全的方案失效了，这就需要全新的机制来处理这样的情况，于是就轮到自旋锁登场了。

自旋锁的原理，它是这样的：首先读取锁变量，判断其值是否已经加锁，如果未加锁则执行加锁，然后返回，表示加锁成功；如果已经加锁了，就要返回第一步继续执行后续步骤，因而得名自旋锁。为了让你更好理解，下面来画一个图描述这个算法。



这个算法看似很好，但是想要正确执行它，就必须保证读取锁变量和判断并加锁的操作是原子执行的。否则，CPU0 在读取了锁变量之后，CPU1 读取锁变量判断未加锁执行加锁，然后 CPU0 也判断未加锁执行加锁，这时就会发现两个 CPU 都加锁成功，因此这个算法出错了。

怎么解决这个问题呢？这就要找硬件要解决方案了，x86 CPU 给我们提供了一个原子交换指令，xchg，它可以让寄存器里的一个值跟内存空间中的一个值做交换。例如，让 eax=memlock，memlock=eax 这个动作是原子的，不受其它 CPU 干扰。

```
//自旋锁结构
typedef struct
{
     volatile u32_t lock;//volatile可以防止编译器优化，保证其它代码始终从内存加载lock变量的值
} spinlock_t;
//锁初始化函数
static inline void x86_spin_lock_init(spinlock_t * lock)
{
     lock->lock = 0;//锁值初始化为0是未加锁状态
}
//加锁函数
static inline void x86_spin_lock(spinlock_t * lock)
{
    __asm__ __volatile__ (
    "1: \n"
    "lock; xchg  %0, %1 \n"//把值为1的寄存器和lock内存中的值进行交换
    "cmpl   $0, %0 \n" //用0和交换回来的值进行比较
    "jnz    2f \n"  //不等于0则跳转后面2标号处运行
    "jmp 3f \n"     //若等于0则跳转后面3标号处返回
    "2:         \n"
    "cmpl   $0, %1  \n"//用0和lock内存中的值进行比较
    "jne    2b      \n"//若不等于0则跳转到前面2标号处运行继续比较
    "jmp    1b      \n"//若等于0则跳转到前面1标号处运行，交换并加锁
    "3:  \n"     :
    : "r"(1), "m"(*lock));
}
//解锁函数
static inline void x86_spin_unlock(spinlock_t * lock)
{
    __asm__ __volatile__(
    "movl   $0, %0\n"//解锁把lock内存中的值设为0就行
    :
    : "m"(*lock));
}
```

自旋锁依然有中断嵌套的问题，也就是说，在使用自旋锁的时候我们仍然要注意中断。在中断处理程序访问某个自旋锁保护的某个资源时，依然有问题，所以我们要写的自旋锁函数必须适应这样的中断环境，也就是说，它需要在处理中断的过程中也能使用，如下所示

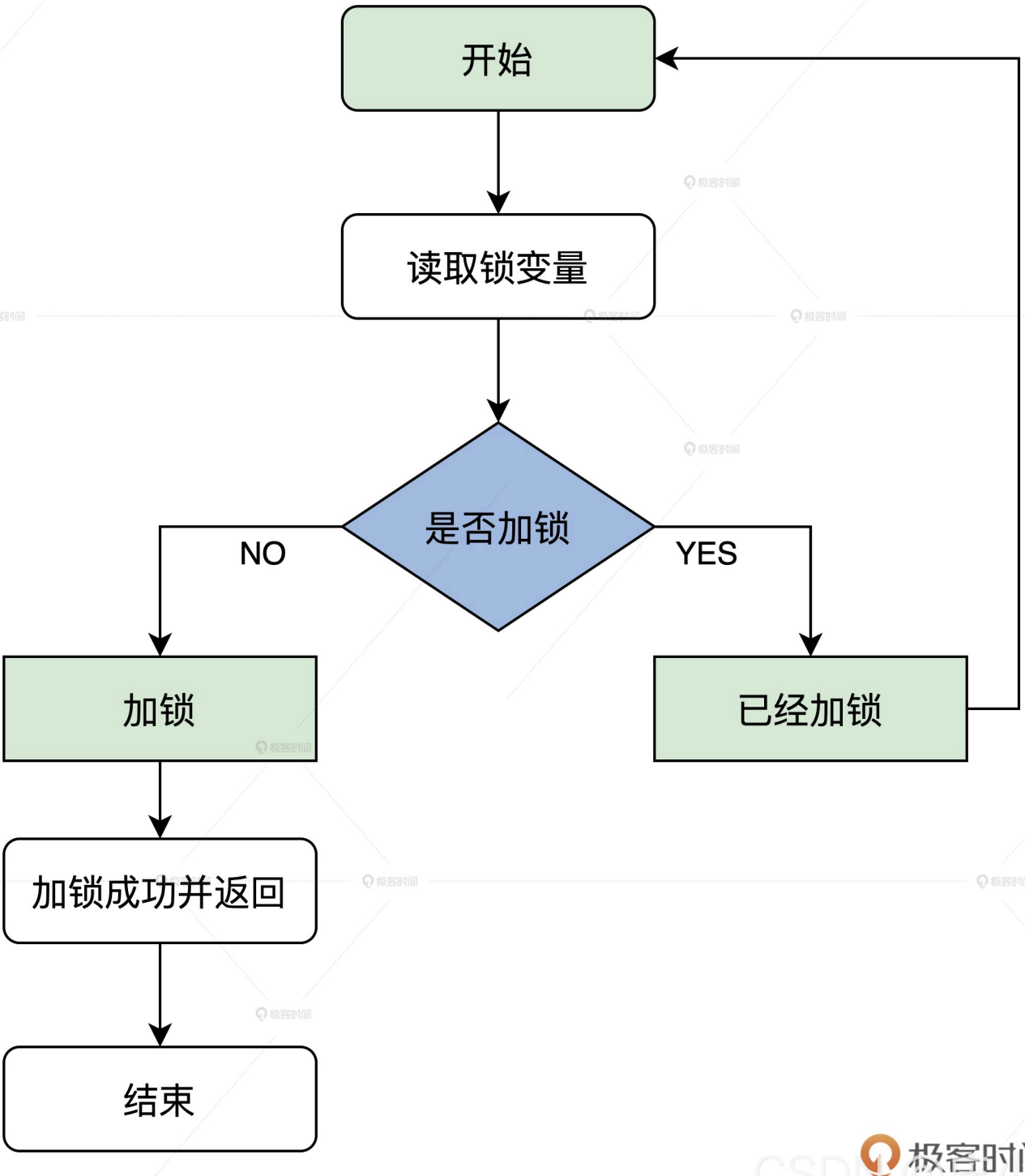
```
static inline void x86_spin_lock_disable_irq(spinlock_t * lock,cpuflg_t* flags)
{
    __asm__ __volatile__(
    "pushfq                 \n\t"
    "cli                    \n\t"
    "popq %0                \n\t"
    "1:         \n\t"
    "lock; xchg  %1, %2 \n\t"
    "cmpl   $0,%1       \n\t"
    "jnz    2f      \n\t"
    "jmp    3f      \n"
    "2:         \n\t"
    "cmpl   $0,%2       \n\t"
    "jne    2b      \n\t"
    "jmp    1b      \n\t"
    "3:     \n"
     :"=m"(*flags)
    : "r"(1), "m"(*lock));
}
static inline void x86_spin_unlock_enabled_irq(spinlock_t* lock,cpuflg_t* flags)
{
    __asm__ __volatile__(
    "movl   $0, %0\n\t"
    "pushq %1 \n\t"
    "popfq \n\t"
    :
    : "m"(*lock), "m"(*flags));
}
```

以上代码实现了关中断下获取自旋锁，以及恢复中断状态释放自旋锁。在中断环境下也完美地解决了问题。

linux自旋转的实现

Linux 也是支持多核心 CPU 的操作系统内核，因此 Linux 也需要自旋锁来对系统中的共享资源进行保护。同一时刻，只有获取了锁的进程才能使用共享资源。根据上节课对自旋锁算法的理解，自旋锁不会引起加锁进程睡眠，如果自旋锁已经被别的进程持有，加锁进程就需要一直循环在那里，查看是否该自旋锁的持有者已经释放了锁，"自旋"一词就是因此而得名。Linux 有多种自旋锁，我们这里只介绍两种，**原始自旋锁和排队自旋锁。**

我们先看看 Linux 原始的自旋锁，Linux 的原始自旋锁本质上用一个整数来表示，值为 1 代表锁未被占用，为 0 或者负数则表示被占用。你可以结合上节课的这张图，理解后面的内容。当某个 CPU 核心执行进程请求加锁时，如果锁是未加锁状态，则加锁，然后操作共享资源，最后释放锁；如果锁已被加锁，则进程并不会转入睡眠状态，而是循环等待该锁，一旦锁被释放，则第一个感知此信息的进程将获得锁。



我们先来看看 Linux 原始自旋锁的数据结构，为方便你阅读，我删除了用于调试的数据字段，代码如下。

```
//最底层的自旋锁数据结构
typedef struct{
volatile unsigned long lock;//真正的锁值变量，用volatile标识
}spinlock_t;
```

Linux 原始自旋锁数据结构封装了一个 unsigned long 类型的变量。有了数据结构，我们再来看看操作这个数据结构的函数，即自旋锁接口，代码如下。

```
#define spin_unlock_string \
    "movb $1,%0" \ //写入1表示解锁
    :"=m" (lock->lock) : : "memory"

#define spin_lock_string \
  "\n1:\t" \
    "lock ; decb %0\n\t" \ //原子减1
  "js 2f\n" \    //当结果小于0则跳转到标号2处，表示加锁失败
    ".section .text.lock,\"ax\"\n" \ //重新定义一个代码段，这是优化技术，避免后面的代码填充cache，因为大部分情况会加
  "2:\t" \
    "cmpb $0,%0\n\t" \  //和0比较
    "rep;nop\n\t" \  //空指令
    "jle 2b\n\t" \   //小于或等于0跳转到标号2
    "jmp 1b\n" \   //跳转到标号1
    ".previous"
//获取自旋锁
static inline void spin_lock(spinlock_t*lock){
    __asm__ __volatile__(
    spin_lock_string
    :"=m"(lock->lock)::"memory"
    );
}
//释放自旋锁
static inline void spin_unlock(spinlock_t*lock){
__asm__ __volatile__(
    spin_unlock_string
    );
}
```

上述代码中用 spin_lock_string、spin_unlock_string 两个宏，定义了获取、释放自旋锁的汇编指令。spin_unlock_string 只是简单将锁值变量设置成 1，表示释放自旋锁，spin_lock_string 中并没有像我们 Cosmos 一样使用 xchg 指令，而是使用了 decb 指令，这条指令也能原子地执行减 1 操作。开始锁值变量为 1 时，执行 decb 指令就变成了 0，0 就表示加锁成功。如果小于 0，则表示有其它进程已经加锁了，就会导致循环比较。

Linux 排队自旋锁的实现

现在我们再来看看 100 个进程获取同一个自旋锁的情况，开始 1 个进程获取了自旋锁 L，后面继续来了 99 个进程，它们都要获取自旋锁 L，但是它们必须等待，这时第 1 进程释放了自旋锁 L。请问，这 99 个进程中谁能先获取自旋锁 L 呢？答案是不确定，因为这个次序依赖于哪个 CPU 核心能最先访问内存，而哪个 CPU 核心可以访问内存是由总线仲裁协议决定的。很有可能最后来的进程最先获取自旋锁 L，这对其它等待的进程极其不公平，为了解决获取自旋锁的公平性，Linux 开发出了排队自旋锁。你可以这样理解，想要给进程排好队，就需要确定顺序，也就是进程申请获取锁的先后次序，Linux 的排队自旋锁通过保存这个信息，就能更公平地调度进程了。

```
//RAW层的自旋锁数据结构
typedef struct raw_spinlock{
    unsigned int slock;//真正的锁值变量
}raw_spinlock_t;
//最上层的自旋锁数据结构
typedef struct spinlock{
    struct raw_spinlock rlock;
}spinlock_t;
//Linux没有这样的结构，这只是为了描述方便
typedef struct raw_spinlock{
    union {
        unsigned int slock;//真正的锁值变量
        u16 owner;
        u16 next;
    }
}raw_spinlock_t;
```

slock 域被分成两部分，分别保存锁持有者和未来锁申请者的序号，如上述代码 10～16 行所示。只有 next 域与 owner 域相等时，才表示自旋锁处于未使用的状态（此时也没有进程申请该锁）。在排队自旋锁初始化时，slock 被置为 0，即 next 和 owner 被置为 0，Linux 进程执行申请自旋锁时，原子地将 next 域加 1，并将原值返回作为自己的序号。如果返回的序号等于申请时的 owner 值，说明自旋锁处于未使用的状态，则进程直接获得锁；否则，该进程循环检查 owner 域是否等于自己持有的序号，一旦相等，则表明锁轮到自己获取。

进程释放自旋锁时，原子地将 owner 域加 1 即可，下一个进程将会发现这一变化，从循环状态中退出。进程将严格地按照申请顺序依次获取排队自旋锁。这样一来，原先进程无序竞争的乱象就迎刃而解了。

```
static inline void __raw_spin_lock(raw_spinlock_t*lock){
int inc = 0x00010000;
int tmp;
__asm__ __volatile__(
"lock ; xaddl %0, %1\n" //将inc和slock交换，然后 inc=inc+slock
                        //相当于原子读取next和owner并对next+1
"movzwl %w0, %2\n\t"//将inc的低16位做0扩展后送tmp tmp=(u16)inc
"shrl $16, %0\n\t" //将inc右移16位 inc=inc>>16
"1:\t"
"cmpl %0, %2\n\t" //比较inc和tmp，即比较next和owner
"je 2f\n\t" //相等则跳转到标号2处返回
"rep ; nop\n\t" //空指令
"movzwl %1, %2\n\t" //将slock的低16位做0扩展后送tmp 即tmp=owner
"jmp 1b\n" //跳转到标号1处继续比较
"2:"
:"+Q"(inc),"+m"(lock->slock),"=r"(tmp)
::"memory","cc"
);
}
#define UNLOCK_LOCK_PREFIX LOCK_PREFIX
static inline void __raw_spin_unlock(raw_spinlock_t*lock){
__asm__ __volatile__(
UNLOCK_LOCK_PREFIX"incw %0"//将slock的低16位加1 即owner+1
:"+m"(lock->slock)
::"memory","cc");
}
```

假如当你去银行办事，又发现人很多时，你很可能会选择先去处理一些别的事情，等过一会人比较少了，再来办理我们自己的业务。其实，在使用自旋锁时也有同样的情况，当一个进程发现另一个进程已经拥有自己所请求的自旋锁时，就自愿放弃，转而做其它别的工作，并不想在这里循环等待，浪费自己的时间。对于这种情况，Linux 同样提供了相应的自旋锁接口，如下所示

```
static inline int __raw_spin_trylock(raw_spinlock_t*lock){
    int tmp;
    int new;
    asm volatile(
    "movl %2,%0\n\t"//tmp=slock
    "movl %0,%1\n\t"//new=tmp
    "roll $16, %0\n\t"//tmp循环左移16位，即next和owner交换了
    "cmpl %0,%1\n\t"//比较tmp和new即（owner、next）？=（next、owner）
    "jne 1f\n\t" //不等则跳转到标号1处
    "addl $0x00010000, %1\n\t"//相当于next+1
    "lock ; cmpxchgl %1,%2\n\t"//new和slock交换比较
    "1:"
    "sete %b1\n\t" //new = eflags.ZF位，ZF取决于前面的判断是否相等
    "movzbl %b1,%0\n\t" //tmp = new
    :"=&a"(tmp),"=Q"(new),"+m"(lock->slock)
    ::"memory","cc");
    return tmp;
}
int __lockfunc _spin_trylock(spinlock_t*lock){
    preempt_disable();
    if(_raw_spin_trylock(lock)){
        spin_acquire(&lock->dep_map,0,1,_RET_IP_);
        return 1;
    }
    preempt_enable();
    return 0;
}
#define spin_trylock(lock) __cond_lock(lock, _spin_trylock(lock))
```

信号量

如果长时间等待后才能获取数据，在这样的情况下，前面中断控制和自旋锁都不能很好地解决，于是我们开发了信号量。信号量由一套数据结构和函数组成，它能使获取数据的代码执行流进入睡眠，然后在相关条件满足时被唤醒，这样就能让 CPU 能有时间处理其它任务。所以信号量同时解决了三个问题：**等待、互斥、唤醒。**

假设这样一个情境：微信等待你从键盘上的输入信息，然后把这个信息发送出去。

下面我们就来说说实现它的一般方法，当然具体实现中可能不同，但是原理是相通的，具体如下。1. 一块内存，相当于缓冲区，用于保存键盘的按键码。

2. 需要一套控制机制，比如微信读取这个缓冲区，而该缓冲区为空时怎么处理；该缓冲区中有了按键码，却没有代码执行流来读取，又该怎么处理。

我们期望是这样的，一共有三点。

1. 当微信获取键盘输入信息时，发现键盘缓冲区中是空的，就进入等待状态。

2. 同一时刻，只能有一个代码执行流操作键盘缓冲区。

3. 当用户按下键盘时，我们有能力把按键码写入缓冲区中，并且能看一看微信或者其它程序是否在等待该缓冲区，如果是就重新激活微信和其它的程序，让它们重新竞争读取键盘缓冲区，如果竞争失败依然进入等待状态。

其实以上所述无非是三个问题：**等待、互斥、唤醒（即重新激活等待的代码执行流）**。这就需要一种全新的数据结构来解决这些问题。根据上面的问题，这个数据结构至少需要一个变量来表示互斥，比如大于 0 则代码执行流可以继续运行，等于 0 则让代码执行流进入等待状态。还需要一个等待链，用于保存等待的代码执行流。

```
#define SEM_FLG_MUTEX 0
#define SEM_FLG_MULTI 1
#define SEM_MUTEX_ONE_LOCK 1
#define SEM_MULTI_LOCK 0
//等待链数据结构，用于挂载等待代码执行流（线程）的结构，里面有用于挂载代码执行流的链表和计数器变量，这里我们先不深入研究这
typedef struct s_KWLST
{
    spinlock_t wl_lock;
    uint_t   wl_tdnr;
    list_h_t wl_list;
}kwlst_t;
//信号量数据结构
typedef struct s_SEM
{
    spinlock_t sem_lock;//维护sem_t自身数据的自旋锁
    uint_t sem_flg;//信号量相关的标志
    sint_t sem_count;//信号量计数值
    kwlst_t sem_waitlst;//用于挂载等待代码执行流（线程）结构
}sem_t;
```

搞懂了信号量的结构，我们再来看看信号量的一般用法，注意信号量在使用之前需要先进行初始化。这里假定信号量数据结构中的 sem_count 初始化为 1，sem_waitlst 等待链初始化为空。

第一步，获取信号量。

- 1. 首先对用于保护信号量自身的自旋锁 sem_lock 进行加锁。
- 2. 对信号值 sem_count 执行“减 1”操作，并检查其值是否小于 0。
- 3. 上步中检查 sem_count 如果小于 0，就让进程进入等待状态并且将其挂入 sem_waitlst 中，然后调度其它进程运行。否则表示获取信号量成功。当然最后别忘了对自旋锁 sem_lock 进行解锁。

第二步，代码执行流开始执行相关操作，例如读取键盘缓冲区。第三步，释放信号量。

- 1. 首先对用于保护信号量自身的自旋锁 sem_lock 进行加锁。
- 2. 对信号值 sem_count 执行“加 1”操作，并检查其值是否大于 0。
- 3. 上步中检查 sem_count 值如果大于 0，就执行唤醒 sem_waitlst 中进程的操作，并且需要调度进程时就执行进程调度操作，不管 sem_count 是否大于 0（通常会大于 0）都标记信号量释放成功。当然最后别忘了对自旋锁 sem_lock 进行解锁。

Linux 信号量的实现

Linux 中的信号量同样是用来保护共享资源，能保证资源在一个时刻只有一个进程使用，这是单值信号量。也可以作为资源计数器，比如一种资源有五份，同时最多可以有五个进程，这是多值信号量。单值信号量，类比于私人空间一次只进去一个人，其信号量的值初始值为 1，而多值信号量，相当于是客厅，可同时容纳多个人。其信号量的值初始值为 5，就可容纳 5 个人。信号量的值为正的时候。所申请的进程可以锁定使用它。若为 0，说明它被其它进程占用，申请的进程要进入睡眠队列中，等待被唤醒。所以信号量最大的优势是既可以使申请失败的进程睡眠，还可以作为资源计数器使用。

```
struct semaphore{
    raw_spinlock_t lock;//保护信号量自身的自旋锁
    unsigned int count;//信号量值
    struct list_head wait_list;//挂载睡眠等待进程的链表
};
```

下面我们就跟着 Linux 信号量接口函数，一步步探索 Linux 信号量工作原理，和它对进程状态的影响，先来看看 Linux 信号量的使用案例，如下所示。

```
#define down_console_sem() do { \
    down(&console_sem);\
} while (0)
static void __up_console_sem(unsigned long ip) {
    up(&console_sem);
}
#define up_console_sem() __up_console_sem(_RET_IP_)
//加锁console
void console_lock(void)
{
    might_sleep();
    down_console_sem();//获取信号量console_sem
    if (console_suspended)
        return;
    console_locked = 1;
    console_may_schedule = 1;
}
//解锁console
void console_unlock(void)
{
    static char ext_text[CONSOLE_EXT_LOG_MAX];
    static char text[LOG_LINE_MAX + PREFIX_MAX];
    //……删除了很多代码
    up_console_sem();//释放信号量console_sem
    raw_spin_lock(&logbuf_lock);
    //……删除了很多代码
}
```

在 Linux 源代码的 kernel/printk.c 中，使用宏 DEFINE_SEMAPHORE 声明了一个单值信号量 console_sem，也可以说是互斥锁，它用于保护 console 驱动列表 console_drivers 以及同步对整个 console 驱动的访问。

其中定义了宏 down_console_sem() 来获得信号量 console_sem，定义了宏 up_console_sem() 来释放信号量 console_sem，console_lock 和 console_unlock 函数是用于互斥访问 console 驱动的，核心操作就是调用前面定义两个宏。

上面的情景中，down_console_sem() 和 up_console_sem() 宏的核心主要是调用了信号量的接口函数 down、up 函数，完成获取、释放信号量的核心操作，代码如下。

```
static inline int __sched __down_common(struct semaphore *sem, long state,long timeout)
{
    struct semaphore_waiter waiter;
    //把waiter加入sem->wait_list的头部
    list_add_tail(&waiter.list, &sem->wait_list);
    waiter.task = current;//current表示当前进程，即调用该函数的进程
    waiter.up = false;
    for (;;) {
        if (signal_pending_state(state, current))
            goto interrupted;
        if (unlikely(timeout <= 0))
            goto timed_out;
        __set_current_state(state);//设置当前进程的状态，进程睡眠，即先前__down函数中传入的TASK_UNINTERRUPTIBLE：
        raw_spin_unlock_irq(&sem->lock);//释放在down函数中加的锁
        timeout = schedule_timeout(timeout);//真正进入睡眠
        raw_spin_lock_irq(&sem->lock);//进程下次运行会回到这里，所以要加锁
        if (waiter.up)
            return 0;
    }
 timed_out:
    list_del(&waiter.list);
    return -ETIME;
 interrupted:
    list_del(&waiter.list);
    return -EINTR;

    //为了简单起见处理进程信号（signal）和超时的逻辑代码我已经删除
}
//进入睡眠等待
static noinline void __sched __down(struct semaphore *sem)
{
    __down_common(sem, TASK_UNINTERRUPTIBLE, MAX_SCHEDULE_TIMEOUT);
}
//获取信号量
void down(struct semaphore *sem)
{
    unsigned long flags;
    //对信号量本身加锁并关中断，必须另一段代码也在操作该信号量
    raw_spin_lock_irqsave(&sem->lock, flags);
    if (likely(sem->count > 0))
        sem->count--;//如果信号量值大于0,则对其减1
    else
        __down(sem);//否则让当前进程进入睡眠
    raw_spin_unlock_irqrestore(&sem->lock, flags);
}
//实际唤醒进程
static noinline void __sched __up(struct semaphore *sem)
{
    struct semaphore_waiter *waiter = list_first_entry(&sem->wait_list, struct semaphore_waiter, list);
    //获取信号量等待链表中的第一个数据结构semaphore_waiter，它里面保存着睡眠进程的指针
    list_del(&waiter->list);
    waiter->up = true;
    wake_up_process(waiter->task);//唤醒进程重新加入调度队列
}
//释放信号量
```

```
void up(struct semaphore *sem)
{
    unsigned long flags;
    //对信号量本身加锁并关中断，必须另一段代码也在操作该信号量
    raw_spin_lock_irqsave(&sem->lock, flags);
    if (likely(list_empty(&sem->wait_list)))
        sem->count++;//如果信号量等待链表中为空，则对信号量值加1
    else
        __up(sem);//否则执行唤醒进程相关的操作
    raw_spin_unlock_irqrestore(&sem->lock, flags);
}
```

Linux 读写锁

在操作系统中，有很多共享数据，进程对这些共享数据要进行修改的情况很少，而读取的情况却是非常多的，这些共享数据的操作基本都是在读取。如果每次读取这些共享数据都加锁的话，那就太浪费时间了，会降低进程的运行效率。因为读操作不会导致修改数据，所以在读取数据的时候不用加锁了，而是可以共享的访问，只有涉及到对共享数据修改的时候，才需要加锁互斥访问。

想像一下 100 个进程同时读取一个共享数据，而每个进程都要加锁解锁，剩下的进程只能等待，这会大大降低整个系统性能，这时候就需要使用一种新的锁了——读写锁。

读写锁也称为共享 - 独占（shared-exclusive）锁，当读写锁用读取模式加锁时，它是以共享模式上锁的，当以写入修改模式加锁时，它是以独占模式上锁的（互斥）。读写锁非常适合读取数据的频率远大于修改数据的频率的场景中。这样可以在任何时刻，保证多个进程的读取操作并发地执行，给系统带来了更高的并发度。

那读写锁是怎么工作的呢？读写之间是互斥的，读取的时候不能写入，写入的时候不能读取，而且读取和写入操作在竞争锁的时候，写会优先得到锁，步骤如下。

- 1. 当共享数据没有锁的时候，读取的加锁操作和写入的加锁操作都可以满足。
- 2. 当共享数据有读锁的时候，所有的读取加锁操作都可以满足，写入的加锁操作不能满足，读写是互斥的。
- 3. 当共享数据有写锁的时候，所有的读取的加锁操作都不能满足，所有的写入的加锁操作也不能满足，读与写之间是互斥的，写与写之间也是互斥的。

锁的状态	读取的加锁操作	写入的加锁操作
无锁	允许	允许
持有读锁	允许	不允许
持有写锁	不允许	不允许

现在就去看看 Linux 中的读写锁的实现，Linux 中的读写锁本质上是自旋锁的变种

后面这段代码是 Linux 中读写锁的核心代码，请你注意，实际操作的时候，我们不是直接使用上面的函数和数据结构，而是应该使用 Linux 提供的标准接口，如 read_lock、write_lock 等。

```
//读写锁初始化锁值
#define RW_LOCK_BIAS     0x01000000
//读写锁的底层数据结构
typedef struct{
    unsigned int lock;
}arch_rwlock_t;
//释放读锁
static inline void arch_read_unlock(arch_rwlock_t*rw){
    asm volatile(
        LOCK_PREFIX"incl %0" //原子对lock加1
        :"+m"(rw->lock)::"memory");
}
//释放写锁
static inline void arch_write_unlock(arch_rwlock_t*rw){
    asm volatile(
        LOCK_PREFIX"addl %1, %0"//原子对lock加上RW_LOCK_BIAS
        :"+m"(rw->lock):"i"(RW_LOCK_BIAS):"memory");
}
//获取写锁失败时调用
ENTRY(__write_lock_failed)
    //(%eax)表示由eax指向的内存空间是调用者传进来的
    2:LOCK_PREFIX addl  $ RW_LOCK_BIAS,(%eax)
    1:rep;nop//空指令
    cmpl $RW_LOCK_BIAS,(%eax)
    //不等于初始值则循环比较，相等则表示有进程释放了写锁
    jne   1b
    //执行加写锁
    LOCK_PREFIX subl  $ RW_LOCK_BIAS,(%eax)
    jnz 2b //不为0则继续测试，为0则表示加写锁成功
    ret //返回
ENDPROC(__write_lock_failed)
//获取读锁失败时调用
ENTRY(__read_lock_failed)
    //(%eax)表示由eax指向的内存空间是调用者传进来的
    2:LOCK_PREFIX incl(%eax)//原子加1
    1:  rep; nop//空指令
    cmpl  $1,(%eax) //和1比较 小于0则
    js 1b //为负则继续循环比较
    LOCK_PREFIX decl(%eax) //加读锁
    js  2b  //为负则继续加1并比较，否则返回
    ret //返回
ENDPROC(__read_lock_failed)
//获取读锁
static inline void arch_read_lock(arch_rwlock_t*rw){
    asm volatile(
        LOCK_PREFIX" subl $1,(%0)\n\t"//原子对lock减1
        "jns 1f\n"//不为小于0则跳转标号1处，表示获取读锁成功
        "call __read_lock_failed\n\t"//调用__read_lock_failed
        "1:\n"
        ::LOCK_PTR_REG(rw):"memory");
}
//获取写锁
static inline void arch_write_lock(arch_rwlock_t*rw){
```

```
    asm volatile(
        LOCK_PREFIX"subl %1,(%0)\n\t"//原子对lock减去RW_LOCK_BIAS
        "jz 1f\n"//为0则跳转标号1处
        "call __write_lock_failed\n\t"//调用__write_lock_failed
        "1:\n"
        ::LOCK_PTR_REG(rw),"i"(RW_LOCK_BIAS):"memory");
}
```

Linux 读写锁的原理本质是基于计数器，初始值为 0x01000000，获取读锁时对其减 1，结果不小于 0 则表示获取读锁成功，获取写锁时直接减去 0x01000000。

说到这里你可能要问了，为何要减去初始值呢？这是因为只有当锁值为初始值时，减去初始值结果才可以是 0，这是唯一没有进程持有任何锁的情况，这样才能保证获取写锁时是互斥的。

__read_lock_failed、__write_lock_failed 是两个汇编函数，注释写得很详细了，和前面自旋锁的套路是一样的。我们可以看出，读写锁其实是带计数的特殊自旋锁，能同时被多个读取数据的进程占有或一个修改数据的进程占有，但不能同时被读取数据的进程和修改数据的进程占有。

释放读写锁的流程

- 1. 获取读锁时，锁值变量 lock 计数减去 1，判断结果的符号位是否为 1。若结果符号位为 0 时，获取读锁成功，即表示 lock 大于 0。
- 2. 获取读锁时，锁值变量 lock 计数减去 1，判断结果的符号位是否为 1。若结果符号位为 1 时，获取读锁失败，表示此时读写锁被修改数据的进程占有，此时调用 __read_lock_failed 失败处理函数，循环测试 lock+1 的值，直到结果的值大于等于 1。
- 3. 获取写锁时，锁值变量 lock 计数减去 RW_LOCK_BIAS_STR，即 lock-0x01000000，判断结果是否为 0。若结果为 0 时，表示获取写锁成功。
- 4. 获取写锁时，锁值变量 lock 计数减去 RW_LOCK_BIAS_STR，即 lock-0x01000000，判断结果是否为 0。若结果不为 0 时，获取写锁失败，表示此时有读取数据的进程占有读锁或有修改数据的进程占有写锁，此时调用 __write_lock_failed 失败处理函数，循环测试 lock+0x01000000，直到结果的值等于 0x01000000。