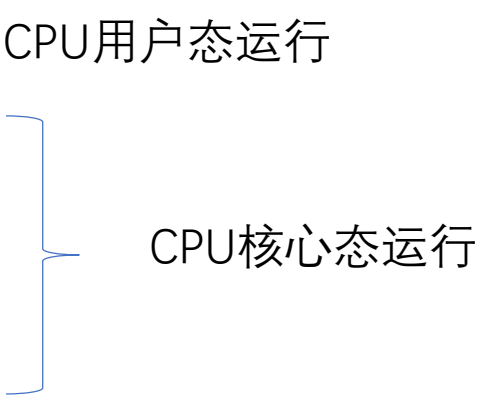
内核同步

内核不是串行运行的。不同内核控制路径可能并发，会交错运行

需要同步

5.1 内核怎样服务请求

- 中断服务
  - 执行应用程序时，有中断请求，服务中断请求
  - 执行系统调用时，有中断请求，服务中断请求
  - 执行中断时，有新中断，服务新中断
  - 全部中断处理完毕，执行系统调用。有可能不是被中断的那一个



- 系统调用服务
  - 用户态进程请求内核服务，它就执行 int 80h指令，发出系统调用。CPU切换至核心态运行，执行一系列内核子程序，为应用程序服务

进程切换 1

- 现运行进程主动放弃CPU

  入睡，终止。
- 现运行进程被剥夺

  没睡，把CPU让给了优先级更高的就绪进程，自己变就绪。

  考虑放弃CPU的时刻
  - 返回用户态　不可剥夺的内核。
    - 系统调用跑一半，现运行进程是不会放弃CPU的。
  - 非也　　可剥夺的内核。
    - 系统调用跑一半，可以把CPU让给优先级更高的其它系统调用。这需要现运行进程把CPU让给另一个进程。

进程切换 2

- schedule( ) → switch_to （ ）

- 可剥夺内核 和 不可剥夺内核内核的性能差异
  - 可剥夺内核 对外界反应快。
    - 外部硬件控制器，环境检测器，视频播放器
  - 不可剥夺内核开销小

内核可以被剥夺的情况

- 执行系统调用
- & 没有显式禁内核抢占
- & 开中断

处理preempt_count的宏

- preempt_disable　　preempt_count++

- preempt_enable

```
preempt_count--;
if(TIF_NEED_RESCHED==1)
    preempt_schedule( );
```

```
preempt_schedule( )执行的代码

if ( preempt_count是0 && 中断是开的 )
{
    schedule( );
    preempt_count = 0;
}
```

必需同步
考虑一个共享变量和它的相关临界区

- UP系统（单处理器系统），现运行进程
  - 执行系统调用期间，放弃CPU
  - 响应中断

- SMP系统
  - 在n个核心上会有n个内核控制路径同时在跑。它们彼此独立，保证共享变量的一致性会更难。

- 可以简化内核设计的5个约束条件
  - 中断上半段执行时，禁irq线。所以同类中断控制路径，上半段串行执行，不必同步。
  - 中断上半段和下半段，不会剥夺，不会入睡。它们的执行只会延迟，不会长时间推后。不必考虑因被剥夺引发的同步问题。
  - 上半段不会被下半段或系统调用打断。
  - 在一颗CPU上，softirq串行执行，不同类型的softirq也不会交叉执行。但，其它CPU可能并发软中断。
  - 一颗CPU上，tasklet串行执行。SMP系统中，同种类型的tasklet不会同时执行，但不同的CPU有可能会执行不同类型的tasklet。
- 不必同步的情形
  - 中断处理程序 和 tasklet可以不是可重入函数
  - 软中断 和 tasklet 访问的 Per CPU变量 不用同步
  - 只有一类tasklet会访问到的全局变量不用同步
  - 尽量用局部变量

5.2 同步技术

Technique	Description	Scope
Per-CPU variables	Duplicate a data structure among the CPUs	All CPUs
Atomic operation	Atomic read-modify-write instruction to a counter	All CPUs
Memory barrier	Avoid instruction reordering	Local CPU or All CPUs
Spin lock	Lock with busy wait	All CPUs
Semaphore	Lock with blocking wait (sleep)	All CPUs
Seqlocks	Lock based on an access counter	All CPUs
Local interrupt disabling	Forbid interrupt handling on a single CPU	Local CPU
Local softirq disabling	Forbid deferrable function handling on a single CPU	Local CPU
Read-copy-update (RCU)	Lock-free access to shared data structures through pointers	All CPUs

1、Per-CPU变量

- Per-CPU变量是一个数据结构的数组。每个CPU一个元素
  - 每颗CPU用自己的ID访问自己的专属元素。不能碰其它CPU的元素。所以不用去防其它CPU。
  - 但是：
    - 防中断上半段和下半段 并发访问同一个变量
    - 访问Per-CPU变量时，进程不能放弃CPU

使用Per-CPU变量的函数和宏

DEFINE_PER_CPU(type, name)
Statically allocates a per-CPU array called name of type data structures
alloc_percpu(type)
Dynamically allocates a per-CPU array of type data structures and returns its address

per_cpu(name, cpu)
Selects the element for CPU cpu of the per-CPU array name
_ _get_cpu_var(name)
Selects the local CPU's element of the per-CPU array name
get_cpu_var(name)
Disables kernel preemption, then selects the local CPU's element of the per-CPU array name
put_cpu_var(name)
Enables kernel preemption (name is not used)
free_percpu(pointer)
Releases a dynamically allocated per-CPU array at address pointer
per_cpu_ptr(pointer, cpu)
Returns the address of the element for CPU cpu of the per-CPU array at address pointer

Per-CPU 变量的使用

- 使用Per-CPU变量能够充分挖掘多核系统的并行运算能力，构造高性能子系统。
- 示例：Linux的调度系统中，runqueue是Per-CPU变量。
  - 4核系统，有4个runqueue（4个active队列，4个expired队列，每个CPU有1对），形成相对独立、自治的4个子调度系统。

数据结构 runqueue
（每个处理器有一个）

登记这颗CPU调度状态

```
struct runqueue {
spinlock_t              lock;                /* 保护运行队列的自旋锁*/
unsigned long           nr_running;          /* 可运行任务数目 */
unsigned long           nr_switches;         /* 上下文切换数目*/
unsigned long           expired_timestamp;   /* 队列最后被换出时间 */
unsigned long           nr_uninterruptible;  /* 处于不可中断睡眠状态的任务数目*/
unsigned long long      timestamp_last_tick; /* 最后一个调度程序的节拍 */
struct task_struct      *curr;               /* 当前运行任务 */
struct task_struct      *idle;               /* 该处理器的空任务*/
struct mm_struct        *prev_mm;            /* 最后运行任务的mm_struct结构体*/
struct prio_array       *active;             /* 活动优先级队列 */
struct prio_array       *expired;            /* 超时优先级队列 */
struct prio_array       arrays[2];           /* 实际优先级数组*/
struct task_struct      *migration_thread;   /* 移出线程 */
struct list_head        *migration_queue;    /* 移出队列*/
atomic_t                nr_iowait;           /* 等待I/O操作的任务数目 */
};
```

Per-CPU 变量的使用

- 示例：Linux的调度系统中，runqueue是Per-CPU变量。
  - 平时，每颗CPU操作自己的runqueue，独立调度。除以下情形外，其它CPU一般不会访问自己的runqueue，操作就绪队列不要上锁。
  - 一颗CPU需要访问另一颗CPU的就绪队列时，锁对方的就绪队列，用自旋锁 lock。
    - 唤醒睡眠进程。　　有可能会把被唤醒的进程放在另一个CPU的就绪队列
    - 实施均衡负载操作。　把本地就绪队列中的一些就绪进程挂到目标就绪队列（目的在于：使两条就绪队列的进程数量大致相等）

数据结构 runqueue
（每个处理器有一个）

登记这颗CPU调度状态

```
struct runqueue {
spinlock_t              lock;                /* 保护运行队列的自旋锁*/
unsigned long           nr_running;          /* 可运行任务数目 */
unsigned long           nr_switches;         /* 上下文切换数目*/
unsigned long           expired_timestamp;   /* 队列最后被换出时间 */
unsigned long           nr_uninterruptible;  /* 处于不可中断睡眠状态的任务数目*/
unsigned long long      timestamp_last_tick; /* 最后一个调度程序的节拍 */
struct task_struct      *curr;               /* 当前运行任务 */
struct task_struct      *idle;               /* 该处理器的空任务*/
struct mm_struct        *prev_mm;            /* 最后运行任务的mm_struct结构体*
struct prio_array       *active;             /* 活动优先级队列 */
struct prio_array       *expired;            /* 超时优先级队列 */
struct prio_array       arrays[2];           /* 实际优先级数组*/
struct task_struct      *migration_thread;   /* 移出线程 */
struct list_head        *migration_queue;    /* 移出队列*/
atomic_t                nr_iowait;           /* 等待I/O操作的任务数目 */
};
```

2、原子操作

- UP　　单条指令是原子的。　inc，dec ……全是原子的
- SMP
  - 不访存　　指令是原子的
  - 只访存一次 或 访存一次且变量是字对齐的　　指令是原子的
  - inc，dec 不是原子的
  - 加前缀 lock；******　　指令执行时锁总线，是原子操作

原子变量 和 针对原子变量的原子操作

- 原子变量

  C语言声明:　　atomic_t　v;

告诉编译器，使用单条原子指令，如果是SMP系统，还会在指令前加前缀 lock;

- 2类原子指令:　　atomic算术运算 和 atomic位运算
  - 例:　　atomic_add(i,v)　　i + *v

优化栅栏 和 内存栅栏

- 编译器会重新调整指令在程序中的位置，以优化寄存器的使用。这种重新调整操作（reorder）会极大地提高程序的运行速度。
- 同步会阻止编译器reorder部分指令片段。程序会变慢。

指令
……
指令

同步原语

指令
……
指令

所有同步原语都是 优化 和 内存栅栏
后面的指令不能和前面的指令放在一起优化

优化栅栏

- barrier( )宏

  asm volatile(“”:::”memory”)

- 可以保证 栅栏前面的指令不能和栅栏后面的指令一起优化

内存栅栏

- 保证前面的指令全部执行完毕后，后面的指令才可以执行

- 有memory barrier作用的80X86串行指令：
  - in，out
  - cli，sti……
  - iret
  - 所有 带lock前缀的指令
  - 所有原子操作

Linux使用的内存栅栏

- mb( ) Memory barrier for MP and UP
- rmb( ) Read memory barrier for MP and UP
- wmb( ) Write memory barrier for MP and UP
- smp_mb( ) Memory barrier for MP only
- smp_rmb( ) Read memory barrier for MP only
- smp_wmb( ) Write memory barrier for MP only

1、自旋锁　Spin Lock

- 自旋锁是工作在多处理器环境下的锁。通常自旋锁保护的临界区都很短，只有几条指令。

- 自旋锁的工作过程
  - 内核控制路径A发现锁是开的，它就把锁关上，继续运行。
  - 否则，持有锁的一定是另一个CPU上的其它内核控制路径B。A会忙等直至锁开。

- 注意事项:
  - 持锁进程不能放弃CPU
  - 但忙等锁开时，进程可以把CPU让给高优先级进程

数据结构 和 使用自旋锁的宏

- spinlock_t
  - slock　　1，锁开 ；　0，锁关
  - break_lock　　有其它进程等待开锁

Macro	Description
spin_lock_init( )	Set the spin lock to 1 (unlocked)
spin_lock( )	Cycle until spin lock becomes 1 (unlocked), then set it to 0 (locked)
spin_unlock( )	Set the spin lock to 1 (unlocked)
spin_unlock_wait( )	Wait until the spin lock becomes 1 (unlocked)
spin_is_locked( )	Return 0 if the spin lock is set to 1 (unlocked); 1 otherwise
spin_trylock( )	Set the spin lock to 0 (locked), and return 1 if the previous value of the lock was 1; 0 otherwise

不可剥夺内核

- spin_lock( )

```
1: lock; decb slp->slock
    jns 3f
2: pause
    cmpb $0,slp->slock
    jle 2b
    jmp 1b
3:
```

- spin_unlock( )

```
movb $1, slp->slock
```

可剥夺内核

- spin_lock( )

1、禁内核抢占

2、_raw_spin_trylock( )执行原子检测和设置

```
        movb $0, %al
lock; xchgb %al, slp->slock
```

3、若al是正的，返回，A持锁进临界区（不会放弃CPU)

4、允许内核抢占

5、while(spin_is_locked(slp))

```
    cpu_relax( );
```

6、goto 1

- spin_unlock( )

```
movb $1, slp->slock
preempt_enable( )
```

自旋锁使用时的注意事项

- 自旋锁不可以嵌套调用，会锁死
- 持自旋锁的进程，临界区不可以睡
- 上自旋锁必需 辅以 禁内核剥夺

2、读写自旋锁　Read/Write Spin Lock

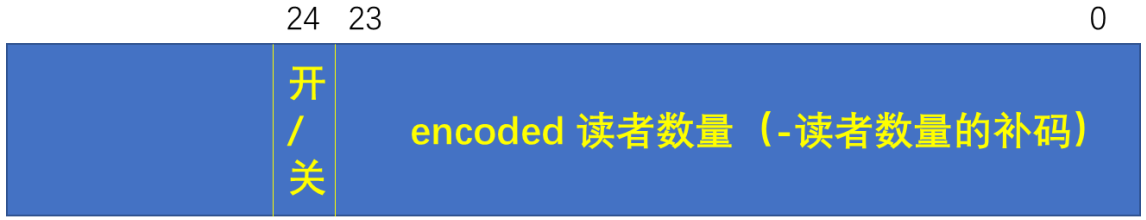
- 允许读者并发，提高系统并发度。

- 数据结构：rwLock_t 结构
  - lock　锁
  - break_lock

- 锁是开的　0x01000000
- 写着持锁　0x00000000

- 1个读者持锁　0x0fffff
- 2个读者持锁　0x0ffffe

读者



- 锁是开的　0x01000000
- 写着持锁　0x00000000
- 1个读者持锁　0x0fffffff
- 2个读者持锁　0x0ffffffe

- 上读锁 read_lock (rwlp)，代码框架与spin_lock宏相同。区别在第2步。

```
int _raw_read_trylock(rwlock_t *lock)
{
    atomic_t *count = (atomic_t *)lock->lock;
    atomic_dec(count);
    if (atomic_read(count) >= 0)
        return 1;
    atomic_inc(count);
    return 0;
}
```

- 解读锁 read_unlock (rwlp)

    lock;　incl　rwlp->lock

    preempt_enable( )

- spin_lock( )

1、禁内核抢占

2、_raw_spin_trylock( )执行原子检测和设置

    movb $0, %al

    lock; xchgb %al, slp->slock

3、若al是正的，返回，A持锁进临界区（不会放弃CPU）

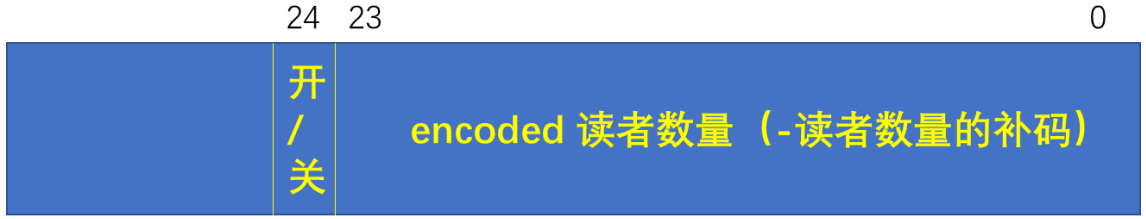
4、允许内核抢占

5、while(spin_is_locked(slp))

    cpu_relax( );　　忙等锁变为 0x01000000

6、goto 1

写者



- 锁是开的　0x01000000
- 写着持锁　0x00000000
- 1个读者持锁　0x0ffffff
- 2个读者持锁　0x0ffffe

- 上写锁 write_lock (rwlp)，代码框架与spin_lock宏相同。区别在第2步。

```
int _raw_write_trylock(rwlock_t *lock)
{
    atomic_t *count = (atomic_t *)lock->lock;
    if (atomic_sub_and_test(0x01000000, count))
        return 1;
    atomic_add(0x01000000, count);
    return 0;
}
```

- spin_lock( )

1、禁内核抢占

2、_raw_spin_trylock( )执行原子检测和设置

movb $0, %al

lock; xchgb %al, slp->slock

3、若al是正的，返回，A持锁进临界区（不会放弃CPU）

4、允许内核抢占

5、while(spin_is_locked(slp))

cpu_relax( );　　忙等锁变为 0x01000000

6、goto 1

- 解写锁 write_unlock (rwlp)

```
lock; addl $0x01000000,rwlp
preempt_enable( )
```

3、顺序锁 Seqlock

- 可以进一步提高系统的并发度。
  - 读操作执行时，写操作可以实施
  - 写操作执行时，不能执行另一个写操作 && 读操作获得的数据不可用
- 数据结构 seqlock_t
  - spin_lock_t lock;　　// 初始值是1，普通计数器
  - int sequence;　　// 初始值是0

写操作（不碰锁）

- spin_lock (lock) ;
- atomic_inc(sequence);
- 临界区，修改被保护的数据结构
- atomic_inc(sequence);
- spin_unlock (lock) ;

读操作

```
unsigned int seq;
do {
    seq = read_seqbegin(&seqlock);        // 得sequence的旧值
    /* ... CRITICAL REGION ...读被保护的数据结构 */
} while (read_seqretry(&seqlock, seq));
   // 读sequence值，偶数且与旧值相等，先前读出的数据结构可用。否则，再读一回
```

4、RCU（Read Copy Update）

- 多个读者和多个写者可以无锁前进。它没用锁也不用计数器。

- RCU锁可以使用的场合
  - RCU只能保护用指针引用的动态分配的数据结构
  - RCU保护的临界区，内核控制路径不能睡

- 读操作：
  - rcu_read_lock( )　　//preempt_disable()
  - pointer → localPointer
  - 用localPointer读数据结构的旧值
  - rcu_read_unlock( )　　//preempt_enable()

- 写操作
  1. newPointer = kmalloc ( ) //为数据结构分配一块新的内存空间
  2. rcu_read_lock( )
  3. pointer → localPointer
  4. 用localPointer读数据结构的旧值 → newPointer指向的内存空间
  5. rcu_read_unlock( )
  6. 随 意 修 改 localPointer指向的数据结构。多久都可以，还能睡
  7. localPointer → pointer

  // 3和7是原子操作，这就保证了读操作读到的要么是旧值，要么是新值.都是可用的值

挑战：回收数据结构旧的copy

- 注意事项：写者执行 localPointer → pointer 的时候，旧的copy不能释放。因为，有可能还有读者在读。

- 挑战1　设计数据结构用来登记所有未被回收的旧的copy

- 挑战2　回收的时机，没有读者持有RCU锁