

第7章

Linux设备驱动中的并发控制

本章导读

在 Linux 设备驱动中必须解决的一个问题是多个进程对共享资源的并发访问，并发的访问会导致竞态，即使是经验丰富的驱动工程师也会常常设计出包含并发问题 bug 的驱动程序。

Linux 提供了多种解决竞态问题的方式，这些方式适合不同的应用场景。

7.1 节讲解了并发和竞态的概念及发生场合。

7.2 节则讲解了编译乱序、执行乱序的问题，以及内存屏障。

7.3 ~ 7.8 节分别讲解了中断屏蔽、原子操作、自旋锁、信号量和互斥体等并发控制机制。

7.9 节讲解增加并发控制后的 globalmem 的设备驱动。

7.1 并发与竞态

并发（Concurrency）指的是多个执行单元同时、并行被执行，而并发的执行单元对共享资源（硬件资源和软件上的全局变量、静态变量等）的访问则很容易导致竞态（Race Conditions）。例如，对于 globalmem 设备，假设一个执行单元 A 对其写入 3000 个字符 “a”，而另一个执行单元 B 对其写入 4000 个 “b”，第三个执行单元 C 读取 globalmem 的所有字符。如果执行单元 A、B 的写操作按图 7.1 那样顺序发生，执行单元 C 的读操作当然不会有什么问题。但是，如果执行单元 A、B 按图 7.2 那样被执行，而执行单元 C 又“不合时宜”地读，则会读出 3000 个 “b”。



图 7.1 并发执行单元的顺序执行



图 7.2 并发执行单元的交错执行

比图 7.2 更复杂、更混乱的并发大量存在于设备驱动中，只要并发的多个执行单元存在对共享资源的访问，竞态就可能发生。在 Linux 内核中，主要的竞态发生于如下几种情况。

1. 对称多处理器 (SMP) 的多个 CPU

SMP 是一种紧耦合、共享存储的系统模型，其体系结构如图 7.3 所示，它的特点是多个 CPU 使用共同的系统总线，因此可访问共同的外设和存储器。

在 SMP 的情况下，两个核 (CPU0 和 CPU1) 的竞态可能发生于 CPU0 的进程与 CPU1 的进程之间、CPU0 的进程与 CPU1 的中断之间以及 CPU0 的中断与 CPU1 的中断之间，图 7.4 中任何一条线连接的两个实体都有核间并发可能性。

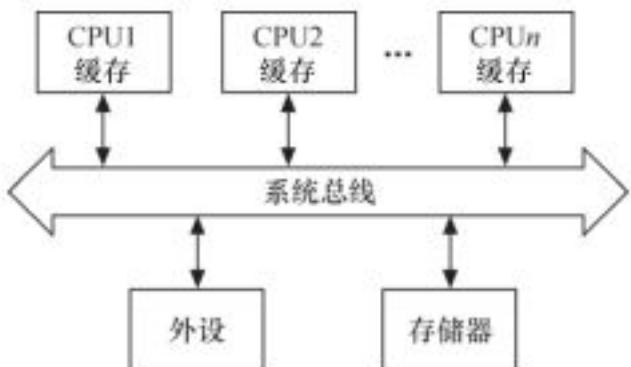


图 7.3 SMP 体系结构

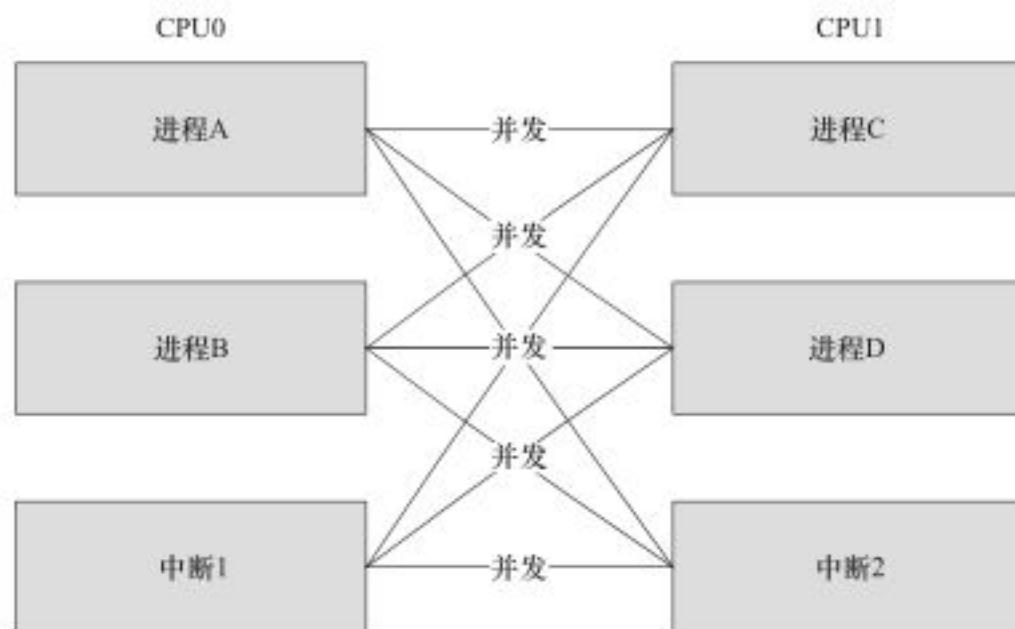


图 7.4 SMP 下多核之间的竞态

2. 单 CPU 内进程与抢占它的进程

Linux 2.6 以后的内核支持内核抢占调度，一个进程在内核执行的时候可能耗完了自己的时间片 (timeslice)，也可能被另一个高优先级进程打断，进程与抢占它的进程访问共享资源

的情况类似于 SMP 的多个 CPU。

3. 中断（硬中断、软中断、Tasklet、底半部）与进程之间

中断可以打断正在执行的进程，如果中断服务程序访问进程正在访问的资源，则竞态也会发生。

此外，中断也有可能被新的更高优先级的中断打断，因此，多个中断之间本身也可能引起并发而导致竞态。但是 Linux 2.6.35 之后，就取消了中断的嵌套。老版本的内核可以在申请中断时，设置标记 IRQF_DISABLED 以避免中断嵌套，由于新内核直接就默认不嵌套中断，这个标记反而变得无用了。详情见 <https://lwn.net/Articles/380931/> 文档《Disabling IRQF_DISABLED》。

上述并发的发生除了 SMP 是真正的并行以外，其他的都是单核上的“宏观并行，微观串行”，但其引发的实质问题和 SMP 相似。图 7.5 再现了 SMP 情况下总的的竞争状态可能性，既包含某一个核内的，也包括两个核间的竞态。

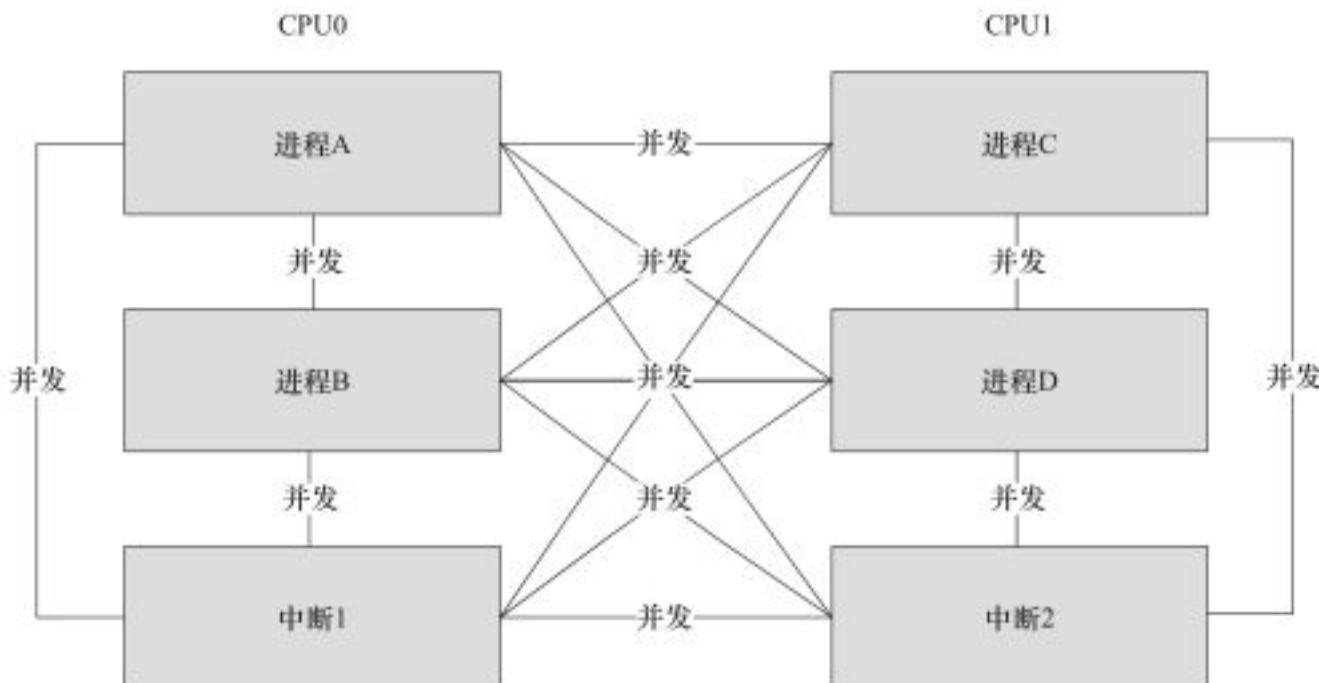


图 7.5 SMP 下核间与核内竞态

解决竞态问题的途径是保证对共享资源的互斥访问，所谓互斥访问是指一个执行单元在访问共享资源的时候，其他的执行单元被禁止访问。

访问共享资源的代码区域称为临界区（Critical Sections），临界区需要被以某种互斥机制加以保护。中断屏蔽、原子操作、自旋锁、信号量、互斥体等是 Linux 设备驱动中可采用的互斥途径。

7.2 编译乱序和执行乱序

理解 Linux 内核的锁机制，还需要理解编译器和处理器的特点。比如下面一段代码，写

端申请一个新的 struct foo 结构体并初始化其中的 a、b、c，之后把结构体地址赋值给全局 gp 指针：

```
struct foo {
    int a;
    int b;
    int c;
};

struct foo *gp = NULL;

/* . . . */

p = kmalloc(sizeof(*p), GFP_KERNEL);
p->a = 1;
p->b = 2;
p->c = 3;
gp = p;
```

而读端如果简单做如下处理，则程序的运行可能是不符合预期的：

```
p = gp;
if (p != NULL) {
    do_something_with(p->a, p->b, p->c);
}
```

有两种可能的原因会造成程序出错，一种可能性是编译乱序，另外一种可能性是执行乱序。

关于编译方面，C 语言顺序的“`p->a = 1; p->b = 2; p->c = 3; gp = p;`”的编译结果的指令顺序可能是 gp 的赋值指令发生在 a、b、c 的赋值之前。现代的高性能编译器在目标码优化上都具备对指令进行乱序优化的能力。编译器可以对访存的指令进行乱序，减少逻辑上不必要的访存，以及尽量提高 Cache 命中率和 CPU 的 Load/Store 单元的工作效率。因此在打开编译器优化以后，看到生成的汇编码并没有严格按照代码的逻辑顺序，这是正常的。

解决编译乱序问题，需要通过 barrier() 编译屏障进行。我们可以在代码中设置 barrier() 屏障，这个屏障可以阻挡编译器的优化。对于编译器来说，设置编译屏障可以保证屏障前的语句和屏障后的语句不乱“串门”。

比如，下面的一段代码在 `e = d[4095]` 与 `b = a, c = a` 之间没有编译屏障：

```
int main(int argc, char *argv[])
{
    int a = 0, b, c, d[4096], e;

    e = d[4095];
    b = a;
    c = a;

    printf("a:%d b:%d c:%d e:%d\n", a, b, c, e);

    return 0;
}
```

用“arm-linux-gnueabihf-gcc -O2”优化编译，反汇编结果是：

```

int main(int argc, char *argv[])
{
    831c: b530      push {r4, r5, lr}
    831e: f5ad 4d80  sub.w sp, sp, #16384 ; 0x4000
    8322: b083      sub sp, #12
    8324: 2100      movs r1, #0
    8326: f50d 4580  add.w r5, sp, #16384 ; 0x4000
    832a: f248 4018  movw r0, #33816 ; 0x8418
    832e: 3504      adds r5, #4
    8330: 460a      mov r2, r1      -> b= a;
    8332: 460b      mov r3, r1      -> c= a;
    8334: f2c0 0000  movt r0, #0
    8338: 682c      ldr r4, [r5, #0]
    833a: 9400      str r4, [sp, #0] -> e = d[4095];
    833c: f7ff efd4  blx 82e8 <_init+0x20>
}

```

显然，尽管源代码级别 $b = a$ 、 $c = a$ 发生在 $e = d[4095]$ 之后，但是目标代码的 $b = a$ 、 $c = a$ 指令发生在 $e = d[4095]$ 之前。

假设我们重新编写代码，在 $e = d[4095]$ 与 $b = a$ 、 $c = a$ 之间加上编译屏障：

```

#define barrier() __asm__ __volatile__("") : : "memory"

int main(int argc, char *argv[])
{
    int a = 0, b, c, d[4096], e;

    e = d[4095];
    barrier();
    b = a;
    c = a;

    printf("a:%d b:%d c:%d e:%d\n", a, b, c, e);

    return 0;
}

```

再次用“arm-linux-gnueabihf-gcc -O2”优化编译，反汇编结果是：

```

int main(int argc, char *argv[])
{
    831c: b510      push {r4, lr}
    831e: f5ad 4d80  sub.w sp, sp, #16384 ; 0x4000
    8322: b082      sub sp, #8
    8324: f50d 4380  add.w r3, sp, #16384 ; 0x4000
    8328: 3304      adds r3, #4
    832a: 681c      ldr r4, [r3, #0]
    832c: 2100      movs r1, #0

```

```

832e: f248 4018    movw r0, #33816   ; 0x8418
8332: f2c0 0000    movt r0, #0
8336: 9400         str  r4, [sp, #0]      -> e = d[4095];
8338: 460a         mov  r2, r1          -> b= a;
833a: 460b         mov  r3, r1          -> c= a;
833c: f7ff efd4    blx  82e8 <_init+0x20>
|

```

因为“`__asm__ __volatile__(""):::"memory")`”这个编译屏障的存在，原来的3条指令的顺序“拨乱反正”了。

关于解决编译乱序的问题，C语言`volatile`关键字的作用较弱，它更多的只是避免内存访问行为的合并，对C编译器而言，`volatile`是暗示除了当前的执行线索以外，其他的执行线索也可能改变某内存，所以它的含义是“易变的”。换句话说，就是如果线程A读取var这个内存中的变量两次而没有修改var，编译器可能觉得读一次就行了，第2次直接取第1次的结果。但是如果加了`volatile`关键字来形容var，则就是告诉编译器线程B、线程C或者其他执行实体可能把var改掉了，因此编译器就不会再把线程A代码的第2次内存读取优化掉了。另外，`volatile`也不具备保护临界资源的作用。总之，Linux内核明显不太喜欢`volatile`，这可参考内核源代码下的文档`Documentation/volatile-considered-harmful.txt`。

编译乱序是编译器的行为，而执行乱序则是处理器运行时的行为。执行乱序是指即便编译的二进制指令的顺序按照“`p->a = 1; p->b = 2; p->c = 3; gp = p;`”排放，在处理器上执行时，后发射的指令还是可能先执行完，这是处理器的“乱序执行(Out-of-Order Execution)”策略。高级的CPU可以根据自己缓存的组织特性，将访存指令重新排序执行。连续地址的访问可能会先执行，因为这样缓存命中率高。有的还允许访存的非阻塞，即如果前面一条访存指令因为缓存不命中，造成长延时的存储访问时，后面的访存指令可以先执行，以便从缓存中取数。因此，即使是从汇编上看顺序正确的指令，其执行的顺序也是不可预知的。

举个例子，ARM v6/v7的处理器会对以下指令顺序进行优化。

```

LDR r0, [r1] ;
STR r2, [r3] ;

```

假设第一条LDR指令导致缓存未命中，这样缓存就会填充行，并需要较多的时钟周期才能完成。老的ARM处理器，比如ARM926EJ-S会等待这个动作完成，再执行下一条STR指令。而ARM v6/v7处理器会识别出下一条指令(STR)且不需要等待第一条指令(LDR)完成(并不依赖于r0的值)，即会先执行STR指令，而不是等待LDR指令完成。

对于大多数体系结构而言，尽管每个CPU都是乱序执行，但是这一乱序对于单核的程序执行是不可见的，因为单个CPU在碰到依赖点(后面的指令依赖于前面指令的执行结果)的时候会等待，所以程序员可能感觉不到这个乱序过程。但是这个依赖点等待的过程，在SMP处理器里面对于其他核是不可见的。比如若在CPU0上执行：

```

while (f == 0);
print x;

```

CPU1 上执行：

```
x = 42;
f = 1;
```

我们不能武断地认为 CPU0 上打印的 x 一定等于 42，因为 CPU1 上即便 “`f = 1`” 编译在 “`x = 42`” 后面，执行时仍然可能先于 “`x = 42`” 完成，所以这个时候 CPU0 上打印的 x 不一定就是 42。

处理器为了解决多核间一个核的内存行为对另外一个核可见的问题，引入了一些内存屏障的指令。譬如，ARM 处理器的屏障指令包括：

DMB（数据内存屏障）：在 DMB 之后的显式内存访问执行前，保证所有在 DMB 指令之前的内存访问完成；

DSB（数据同步屏障）：等待所有在 DSB 指令之前的指令完成（位于此指令前的所有显式内存访问均完成，位于此指令前的所有缓存、跳转预测和 TLB 维护操作全部完成）；

ISB（指令同步屏障）：Flush 流水线，使得所有 ISB 之后执行的指令都是从缓存或内存中获得的。

Linux 内核的自旋锁、互斥体等互斥逻辑，需要用到上述指令：在请求获得锁时，调用屏障指令；在解锁时，也需要调用屏障指令。代码清单 7.1 的汇编代码描绘了一个简单的互斥逻辑，留意其中的第 14 行和 22 行。关于 `ldrex` 和 `strex` 指令的作用，会在 7.3 节详述。

代码清单 7.1 基于内存屏障指令的互斥逻辑

```

1 LOCKED    EQU 1
2 UNLOCKED  EQU 0
3 lock_mutex
4      ; 互斥量是否锁定 ?
5      LDREX r1, [r0]          ; 检查是否锁定
6      CMP r1, #LOCKED       ; 和 "locked" 比较
7      WFEEQ                ; 互斥量已经锁定，进入休眠
8      BEQ lock_mutex       ; 被唤醒，重新检查互斥量是否锁定
9      ; 尝试锁定互斥量
10     MOV r1, #LOCKED
11     STREX r2, r1, [r0]      ; 尝试锁定
12     CMP r2, #0x0           ; 检查 STR 指令是否完成
13     BNE lock_mutex       ; 如果失败，重试
14     DMB                    ; 进入被保护的资源前需要隔离，保证互斥量已经被更新
15     BX lr
16
17 unlock_mutex
18     DMB                    ; 保证资源的访问已经结束
19     MOV r1, #UNLOCKED      ; 向锁定域写 "unlocked"
20     STR r1, [r0]
21
22     DSB                    ; 保证在 CPU 唤醒前完成互斥量状态更新
23     SEV                    ; 像其他 CPU 发送事件，唤醒任何等待事件的 CPU
24
25     BX lr

```

前面提到每个 CPU 都是乱序执行，但是单个 CPU 在碰到依赖点的时候会等待，所以执行乱序对单核不一定可见。但是，当程序在访问外设的寄存器时，这些寄存器的访问顺序在 CPU 的逻辑上构不成依赖关系，但是从外设的逻辑角度来讲，可能需要固定的寄存器读写顺序，这个时候，也需要使用 CPU 的内存屏障指令。内核文档 Documentation/memory-barriers.txt 和 Documentation/io_ordering.txt 对此进行了描述。

在 Linux 内核中，定义了读写屏障 mb()、读屏障 rmb()、写屏障 wmb()、以及作用于寄存器读写的 __iormb()、__iowmb() 这样的屏障 API。读写寄存器的 readl_relaxed() 和 readl()、writel_relaxed() 和 writel() API 的区别就体现在有无屏障方面。

```
#define readb(c)          (( u8 __v = readb_relaxed(c); __iormb(); __v; ))
#define readw(c)          (( u16 __v = readw_relaxed(c); __iormb(); __v; ))
#define readl(c)          (( u32 __v = readl_relaxed(c); __iormb(); __v; ))

#define writeb(v,c)        (( __iowmb(); writeb_relaxed(v,c); ))
#define writew(v,c)        (( __iowmb(); writew_relaxed(v,c); ))
#define writel(v,c)        (( __iowmb(); writel_relaxed(v,c); ))
```

比如我们通过 writel_relaxed() 写完 DMA 的开始地址、结束地址、大小之后，我们一定要调用 writel() 来启动 DMA。

```
writel_relaxed(DMA_SRC_REG, src_addr);
writel_relaxed(DMA_DST_REG, dst_addr);
writel_relaxed(DMA_SIZE_REG, size);
writel(DMA_ENABLE, 1);
```

7.3 中断屏蔽

在单 CPU 范围内避免竞态的一种简单而有效的方法是在进入临界区之前屏蔽系统的中断，但是在驱动编程中不值得推荐，驱动通常需要考虑跨平台特点而不假定自己在单核上运行。CPU 一般都具备屏蔽中断和打开中断的功能，这项功能可以保证正在执行的内核执行路径不被中断处理程序所抢占，防止某些竞态条件的发生。具体而言，中断屏蔽将使得中断与进程之间的并发不再发生，而且，由于 Linux 内核的进程调度等操作都依赖中断来实现，内核抢占进程之间的并发也得以避免了。

中断屏蔽的使用方法为：

```
local_irq_disable()      /* 屏蔽中断 */
...
critical section        /* 临界区 */
...
local_irq_enable()      /* 开中断 */
```

其底层的实现原理是让 CPU 本身不响应中断，比如，对于 ARM 处理器而言，其底层的实现是屏蔽 ARM CPSR 的 I 位：

```

static inline void arch_local_irq_disable(void)
{
    __asm volatile(
        "        cpsid i          @ arch_local_irq_disable"
        :
        :
        : "memory", "cc");
}

```

由于Linux的异步I/O、进程调度等很多重要操作都依赖于中断，中断对于内核的运行非常重要，在屏蔽中断期间所有的中断都无法得到处理，因此长时间屏蔽中断是很危险的，这有可能造成数据丢失乃至系统崩溃等后果。这就要求在屏蔽了中断之后，当前的内核执行路径应当尽快地执行完临界区的代码。

`local_irq_disable()`和`local_irq_enable()`都只能禁止和使能本CPU内的中断，因此，并不能解决SMP多CPU引发的竞争。因此，单独使用中断屏蔽通常不是一种值得推荐的避免竞争的方法（换句话说，驱动中使用`local_irq_disable/enable()`通常意味着一个bug），它适合与下文将要介绍的自旋锁联合使用。

与`local_irq_disable()`不同的是，`local_irq_save(flags)`除了进行禁止中断的操作以外，还保存目前CPU的中断位信息，`local_irq_restore(flags)`进行的是与`local_irq_save(flags)`相反的操作。对于ARM处理器而言，其实就是保存和恢复CPSR。

如果只是想禁止中断的底半部，应使用`local_bh_disable()`，使能被`local_bh_disable()`禁止的底半部应该调用`local_bh_enable()`。

7.4 原子操作

原子操作可以保证对一个整型数据的修改是排他性的。Linux内核提供了一系列函数来实现内核中的原子操作，这些函数又分为两类，分别针对位和整型变量进行原子操作。位和整型变量的原子操作都依赖于底层CPU的原子操作，因此所有这些函数都与CPU架构密切相关。对于ARM处理器而言，底层使用LDREX和STREX指令，比如`atomic_inc()`底层的实现会调用到`atomic_add()`，其代码如下：

```

static inline void atomic_add(int i, atomic_t *v)
{
    unsigned long tmp;
    int result;

    prefetchw(&v->counter);
    __asm__ __volatile__ ("@ atomic_add\n"
"1: ldrex   %0, [%3]\n"
"    add     %0, %0, %4\n"
"    strex   %1, %0, [%3]\n"
"    teq     %1, #0\n"
    );
}

```

```

"      bne    1b"
: "&r" (result), "&r" (tmp), "+Qo" (v->counter)
: "r" (&v->counter), "Ir" (i)
: "cc");
}

```

ldrex 指令跟 strex 配对使用，可以让总线监控 ldrex 到 strex 之间有无其他的实体存取该地址，如果有并发的访问，执行 strex 指令时，第一个寄存器的值被设置为 1 (Non-Exclusive Access) 并且存储的行为也不成功；如果没有并发的存取，strex 在第一个寄存器里设置 0 (Exclusive Access) 并且存储的行为也是成功的。本例中，如果两个并发实体同时调用 ldrex + strex，如图 7.6 所示，在 T3 时间点上，CPU0 的 strex 会执行失败，在 T4 时间点上 CPU1 的 strex 会执行成功。所以 CPU0 和 CPU1 之间只有 CPU1 执行成功了，执行 strex 失败的 CPU0 的“teq %1, #0”判断语句不会成立，于是失败的 CPU0 通过“bne 1b”再次进入 ldrex。ldrex 和 strex 的这一过程不仅适用于多核之间的并发，也适用于同一个核内部并发的情况。

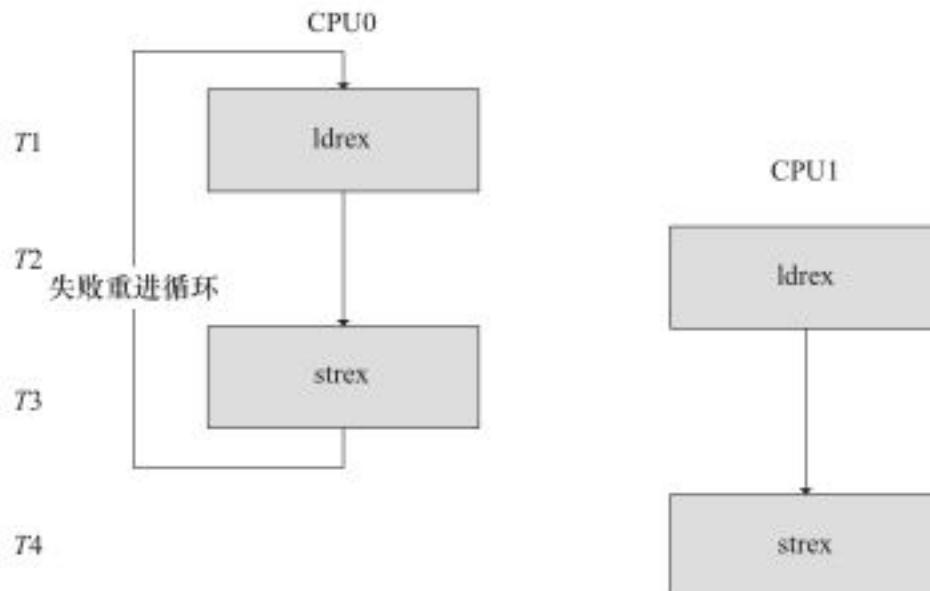


图 7.6 ldrex 和 strex 指令

7.4.1 整型原子操作

1. 设置原子变量的值

```

void atomic_set	atomic_t *v, int i); /* 设置原子变量的值为 i */
atomic_t v = ATOMIC_INIT(0);      /* 定义原子变量 v 并初始化为 0 */

```

2. 获取原子变量的值

```

atomic_read	atomic_t *v); /* 返回原子变量的值 */

```

3. 原子变量加 / 减

```

void atomic_add(int i, atomic_t *v); /* 原子变量增加 i */

```

```
void atomic_sub(int i, atomic_t *v); /* 原子变量减少 i */
```

4. 原子变量自增 / 自减

```
void atomic_inc(atomic_t *v); /* 原子变量增加 1 */  
void atomic_dec(atomic_t *v); /* 原子变量减少 1 */
```

5. 操作并测试

```
int atomic_inc_and_test(atomic_t *v);  
int atomic_dec_and_test(atomic_t *v);  
int atomic_sub_and_test(int i, atomic_t *v);
```

上述操作对原子变量执行自增、自减和减操作后（注意没有加），测试其是否为 0，为 0 返回 true，否则返回 false。

6. 操作并返回

```
int atomic_add_return(int i, atomic_t *v);  
int atomic_sub_return(int i, atomic_t *v);  
int atomic_inc_return(atomic_t *v);  
int atomic_dec_return(atomic_t *v);
```

上述操作对原子变量进行加 / 减和自增 / 自减操作，并返回新的值。

7.4.2 位原子操作

1. 设置位

```
void set_bit(nr, void *addr);
```

上述操作设置 addr 地址的第 nr 位，所谓设置位即是将位写为 1。

2. 清除位

```
void clear_bit(nr, void *addr);
```

上述操作清除 addr 地址的第 nr 位，所谓清除位即是将位写为 0。

3. 改变位

```
void change_bit(nr, void *addr);
```

上述操作对 addr 地址的第 nr 位进行反置。

4. 测试位

```
test_bit(nr, void *addr);
```

上述操作返回 addr 地址的第 nr 位。

5. 测试并操作位

```
int test_and_set_bit(nr, void *addr);
```

```
int test_and_clear_bit(nr, void *addr);
int test_and_change_bit(nr, void *addr);
```

上述 test_and_xxx_bit(nr, void *addr) 操作等同于执行 test_bit(nr, void *addr) 后再执行 xxx_bit(nr, void *addr)。

代码清单 7.2 给出了原子变量的使用例子，它使得设备最多只能被一个进程打开。

代码清单 7.2 使用原子变量使设备只能被一个进程打开

```
1 static atomic_t xxx_available = ATOMIC_INIT(1); /* 定义原子变量 */
2
3 static int xxx_open(struct inode *inode, struct file *filp)
4 {
5     ...
6     if (!atomic_dec_and_test(&xxx_available)) {
7         atomic_inc(&xxx_available);
8         return -EBUSY;           /* 已经打开 */
9     }
10    ...
11    return 0;                 /* 成功 */
12 }
13
14 static int xxx_release(struct inode *inode, struct file *filp)
15 {
16     atomic_inc(&xxx_available); /* 释放设备 */
17     return 0;
18 }
```

7.5 自旋锁

7.5.1 自旋锁的使用

自旋锁（Spin Lock）是一种典型的对临界资源进行互斥访问的手段，其名称来源于它的工作方式。为了获得一个自旋锁，在某 CPU 上运行的代码需先执行一个原子操作，该操作测试并设置（Test-And-Set）某个内存变量。由于它是原子操作，所以在该操作完成之前其他执行单元不可能访问这个内存变量。如果测试结果表明锁已经空闲，则程序获得这个自旋锁并继续执行；如果测试结果表明锁仍被占用，程序将在一个循环内重复这个“测试并设置”操作，即进行所谓的“自旋”，通俗地说就是“在原地打转”，如图 7.7 所示。当自旋锁的持有者通过重置该变量释放这个自旋锁后，某个等待的“测试并设置”操作向其调用者报告锁已释放。

理解自旋锁最简单的方法是把它作为一个变量看待，该变量把一个临界区标记为“我当前在运行，请稍等一会”或者标记为“我当前不在运行，可以被使用”。如果 A 执行单元首先进入例程，它将持有自旋锁；当 B 执行单元试图进入同一个例程时，将获知自旋锁已被持有，需等到 A 执行单元释放后才能进入。



图 7.7 自旋

在 ARM 体系结构下，自旋锁的实现借用了 ldrex 指令、strex 指令、ARM 处理器内存屏障指令 dmb 和 dsb、wfe 指令和 sev 指令，这类似于代码清单 7.1 的逻辑。可以说既要保证排他性，也要处理好内存屏障。

Linux 中与自旋锁相关的操作主要有以下 4 种。

1. 定义自旋锁

```
spinlock_t lock;
```

2. 初始化自旋锁

```
spin_lock_init(lock)
```

该宏用于动态初始化自旋锁 lock。

3. 获得自旋锁

```
spin_lock(lock)
```

该宏用于获得自旋锁 lock，如果能够立即获得锁，它就马上返回，否则，它将在那里自旋，直到该自旋锁的保持者释放。

```
spin_trylock(lock)
```

该宏尝试获得自旋锁 lock，如果能立即获得锁，它获得锁并返回 true，否则立即返回 false，实际上不再“在原地打转”。

4. 释放自旋锁

```
spin_unlock(lock)
```

该宏释放自旋锁 lock，它与 spin_trylock 或 spin_lock 配对使用。

自旋锁一般这样被使用：

```
/* 定义一个自旋锁 */
spinlock_t lock;
```

```

spin_lock_init(&lock);

spin_lock(&lock);      /* 获取自旋锁，保护临界区 */
... /* 临界区 */
spin_unlock(&lock);   /* 解锁 */

```

自旋锁主要针对 SMP 或单 CPU 但内核可抢占的情况，对于单 CPU 和内核不支持抢占的系统，自旋锁退化为空操作。在单 CPU 和内核可抢占的系统中，自旋锁持有期间内核的抢占将被禁止。由于内核可抢占的单 CPU 系统的行为实际上很类似于 SMP 系统，因此，在这样的单 CPU 系统中使用自旋锁仍十分必要。另外，在多核 SMP 的情况下，任何一个核拿到了自旋锁，该核上的抢占调度也暂时禁止了，但是没有禁止另外一个核的抢占调度。

尽管用了自旋锁可以保证临界区不受别的 CPU 和本 CPU 内的抢占进程打扰，但是得到锁的代码路径在执行临界区的时候，还可能受到中断和底半部（BH，稍后的章节会介绍）的影响。为了防止这种影响，就需要用到自旋锁的衍生。spin_lock()/spin_unlock() 是自旋锁机制的基础，它们和关中断 local_irq_disable() / 开中断 local_irq_enable()、关底半部 local_bh_disable() / 开底半部 local_bh_enable()、关中断并保存状态字 local_irq_save() / 开中断并恢复状态字 local_irq_restore() 结合就形成了整套自旋锁机制，关系如下：

```

spin_lock_irq() = spin_lock() + local_irq_disable()
spin_unlock_irq() = spin_unlock() + local_irq_enable()
spin_lock_irqsave() = spin_lock() + local_irq_save()
spin_unlock_irqrestore() = spin_unlock() + local_irq_restore()
spin_lock_bh() = spin_lock() + local_bh_disable()
spin_unlock_bh() = spin_unlock() + local_bh_enable()

```

spin_lock_irq()、spin_lock_irqsave()、spin_lock_bh() 类似函数会为自旋锁的使用系好“安全带”以避免突如其来的中断驶入对系统造成的伤害。

在多核编程的时候，如果进程和中断可能访问同一片临界资源，我们一般需要在进程上下文中调用 spin_lock_irqsave()/spin_unlock_irqrestore()，在中断上下文中调用 spin_lock() / spin_unlock()，如图 7.8 所示。这样，在 CPU0 上，无论是进程上下文，还是中断上下文获得了自旋锁，此后，如果 CPU1 无论是进程上下文，还是中断上下文，想获得同一自旋锁，都必须忙等待，这避免一切核间并发的可能性。同时，由于每个核的进程上下文持有锁的时候用的是 spin_lock_irqsave()，所以该核上的中断是不可能进入的，这避免了核内并发的可能性。

驱动工程师应谨慎使用自旋锁，而且在使用中还要特别注意如下几个问题。

- 1) 自旋锁实际上是忙等锁，当锁不可用时，CPU 一直循环执行“测试并设置”该锁直到可用而取得该锁，CPU 在等待自旋锁时不做任何有用的工作，仅仅是等待。因此，只有在占用锁的时间极短的情况下，使用自旋锁才是合理的。当临界区很大，或有共享设备的时候，需要较长时间占用锁，使用自旋锁会降低系统的性能。

- 2) 自旋锁可能导致系统死锁。引发这个问题最常见的原因是递归使用一个自旋锁，即如果一个已经拥有某个自旋锁的 CPU 想第二次获得这个自旋锁，则该 CPU 将死锁。

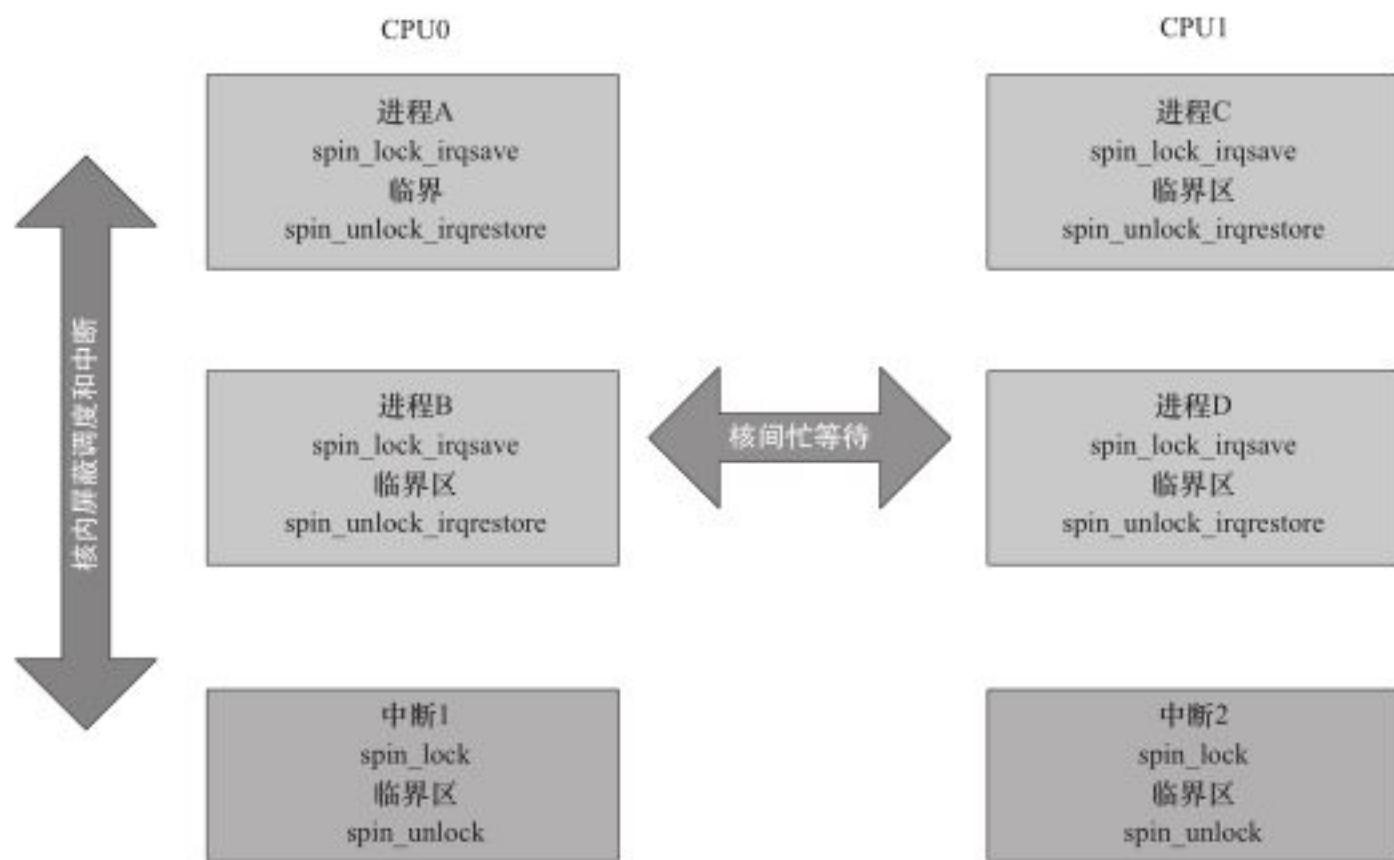


图 7.8 自旋锁的使用实例

3) 在自旋锁锁定期间不能调用可能引起进程调度的函数。如果进程获得自旋锁之后再阻塞，如调用 `copy_from_user()`、`copy_to_user()`、`kmalloc()` 和 `msleep()` 等函数，则可能导致内核的崩溃。

4) 在单核情况下编程的时候，也应该认为自己的CPU是多核的，驱动特别强调跨平台的概念。比如，在单CPU的情况下，若中断和进程可能访问同一临界区，进程里调用 `spin_lock_irqsave()` 是安全的，在中断里其实不调用 `spin_lock()` 也没有问题，因为 `spin_lock_irqsave()` 可以保证这个CPU的中断服务程序不可能执行。但是，若CPU变成多核，`spin_lock_irqsave()` 不能屏蔽另外一个核的中断，所以另外一个核就可能造成并发问题。因此，无论如何，我们在中断服务程序里也应该调用 `spin_lock()`。

代码清单 7.3 给出了自旋锁的使用例子，它被用于实现使得设备只能被最多 1 个进程打开，功能和代码清单与 7.2 类似。

代码清单 7.3 使用自旋锁使设备只能被一个进程打开

```

1 int xxx_count = 0; /* 定义文件打开次数计数 */
2
3 static int xxx_open(struct inode *inode, struct file *filp)
4 {
5     ...
6     spinlock(&xxx_lock);
7     if (xxx_count) { /* 已经打开 */
8         spin_unlock(&xxx_lock);
9     }
10    xxx_count++;
11    spin_unlock(&xxx_lock);
12    return 0;
13 }

```

```

9         return -EBUSY;
10    }
11    xxx_count++; /* 增加使用计数 */
12    spin_unlock(&xxx_lock);
13    ...
14    return 0; /* 成功 */
15 }
16
17 static int xxx_release(struct inode *inode, struct file *filp)
18 {
19    ...
20    spinlock(&xxx_lock);
21    xxx_count--; /* 减少使用计数 */
22    spin_unlock(&xxx_lock);
23
24    return 0;
25 }

```

7.5.2 读写自旋锁

自旋锁不关心锁定的临界区究竟在进行什么操作，不管是读还是写，它都一视同仁。即便多个执行单元同时读取临界资源也会被锁住。实际上，对共享资源并发访问时，多个执行单元同时读取它是不会有问题是的，自旋锁的衍生锁读写自旋锁（rwlock）可允许读的并发。读写自旋锁是一种比自旋锁粒度更小的锁机制，它保留了“自旋”的概念，但是在写操作方面，只能最多有1个写进程，在读操作方面，同时可以有多个读执行单元。当然，读和写也不能同时进行。

读写自旋锁涉及的操作如下。

1. 定义和初始化读写自旋锁

```

rwlock_t my_rwlock;
rwlock_init(&my_rwlock); /* 动态初始化 */

```

2. 读锁定

```

void read_lock(rwlock_t *lock);
void read_lock_irqsave(rwlock_t *lock, unsigned long flags);
void read_lock_irq(rwlock_t *lock);
void read_lock_bh(rwlock_t *lock);

```

3. 读解锁

```

void read_unlock(rwlock_t *lock);
void read_unlock_irqrestore(rwlock_t *lock, unsigned long flags);
void read_unlock_irq(rwlock_t *lock);
void read_unlock_bh(rwlock_t *lock);

```

在对共享资源进行读取之前，应该先调用读锁定函数，完成之后应调用读解锁函数。

`read_lock_irqsave()`、`read_lock_irq()` 和 `read_lock_bh()` 也分别是 `read_lock()` 分别与 `local_irq_save()`、`local_irq_disable()` 和 `local_bh_disable()` 的组合，读解锁函数 `read_unlock_irqrestore()`、`read_unlock_irq()`、`read_unlock_bh()` 的情况与此类似。

4. 写锁定

```
void write_lock(rwlock_t *lock);
void write_lock_irqsave(rwlock_t *lock, unsigned long flags);
void write_lock_irq(rwlock_t *lock);
void write_lock_bh(rwlock_t *lock);
int write_trylock(rwlock_t *lock);
```

5. 写解锁

```
void write_unlock(rwlock_t *lock);
void write_unlock_irqrestore(rwlock_t *lock, unsigned long flags);
void write_unlock_irq(rwlock_t *lock);
void write_unlock_bh(rwlock_t *lock);
```

`write_lock_irqsave()`、`write_lock_irq()`、`write_lock_bh()` 分别是 `write_lock()` 与 `local_irq_save()`、`local_irq_disable()` 和 `local_bh_disable()` 的组合，写解锁函数 `write_unlock_irqrestore()`、`write_unlock_irq()`、`write_unlock_bh()` 的情况与此类似。

在对共享资源进行写之前，应该先调用写锁定函数，完成之后应调用写解锁函数。和 `spin_trylock()` 一样，`write_trylock()` 也只是尝试获取读写自旋锁，不管成功失败，都会立即返回。

读写自旋锁一般这样被使用：

```
rwlock_t lock;           /* 定义 rwlock */
rwlock_init(&lock);     /* 初始化 rwlock */

/* 读时获取锁 */
read_lock(&lock);
...
/* 临界资源 */
read_unlock(&lock);

/* 写时获取锁 */
write_lock_irqsave(&lock, flags);
...
/* 临界资源 */
write_unlock_irqrestore(&lock, flags);
```

7.5.3 顺序锁

顺序锁（seqlock）是对读写锁的一种优化，若使用顺序锁，读执行单元不会被写执行单元阻塞，也就是说，读执行单元在写执行单元对被顺序锁保护的共享资源进行写操作时仍然可以继续读，而不必等待写执行单元完成写操作，写执行单元也不需要等待所有读执行单元完成读操作才去进行写操作。但是，写执行单元与写执行单元之间仍然是互斥的，即如果有写执行单元在进行写操作，其他写执行单元必须自旋在那里，直到写执行单元释放了顺序锁。

对于顺序锁而言，尽管读写之间不互相排斥，但是如果读执行单元在读操作期间，写执行单元已经发生了写操作，那么，读执行单元必须重新读取数据，以便确保得到的数据是完整的。所以，在这种情况下，读端可能反复读多次同样的区域才能读到有效的数据。

在Linux内核中，写执行单元涉及的顺序锁操作如下。

1. 获得顺序锁

```
void write_seqlock(seqlock_t *sl);
int write_tryseqlock(seqlock_t *sl);
write_seqlock_irqsave(lock, flags)
write_seqlock_irq(lock)
write_seqlock_bh(lock)
```

其中，

```
write_seqlock_irqsave() = local_irq_save() + write_seqlock()
write_seqlock_irq() = local_irq_disable() + write_seqlock()
write_seqlock_bh() = local_bh_disable() + write_seqlock()
```

2. 释放顺序锁

```
void write_sequnlock(seqlock_t *sl);
write_sequnlock_irqrestore(lock, flags)
write_sequnlock_irq(lock)
write_sequnlock_bh(lock)
```

其中，

```
write_sequnlock_irqrestore() = write_sequnlock() + local_irq_restore()
write_sequnlock_irq() = write_sequnlock() + local_irq_enable()
write_sequnlock_bh() = write_sequnlock() + local_bh_enable()
```

写执行单元使用顺序锁的模式如下：

```
write_seqlock(&seqlock_a);
.../* 写操作代码块 */
write_sequnlock(&seqlock_a);
```

因此，对写执行单元而言，它的使用与自旋锁相同。

读执行单元涉及的顺序锁操作如下。

1. 读开始

```
unsigned read_seqbegin(const seqlock_t *sl);
read_seqbegin_irqsave(lock, flags)
```

读执行单元在对被顺序锁 sl 保护的共享资源进行访问前需要调用该函数，该函数返回顺序锁 sl 的当前顺序号。其中，

```
read_seqbegin_irqsave() = local_irq_save() + read_seqbegin()
```

2. 重读

```
int read_seqretry(const seqlock_t *sl, unsigned iv);
read_seqretry_irqrestore(lock, iv, flags)
```

读执行单元在访问完被顺序锁 sl 保护的共享资源后需要调用该函数来检查，在读访问期间是否有写操作。如果有写操作，读执行单元就需要重新进行读操作。其中，

```
read_seqretry_irqrestore() = read_seqretry() + local_irq_restore()
```

读执行单元使用顺序锁的模式如下：

```
do {
    seqnum = read_seqbegin(&seqlock_a);
    /* 读操作代码块 */
    ...
} while (read_seqretry(&seqlock_a, seqnum));
```

7.5.4 读-复制-更新

RCU (Read-Copy-Update, 读 - 复制 - 更新)，它是基于其原理命名的。RCU 并不是新的锁机制，早在 20 世纪 80 年代就有了这种机制，而在 Linux 中是在开发内核 2.5.43 时引入该技术的，并正式包含在 2.6 内核中。

Linux 社区关于 RCU 的经典文档位于 <https://www.kernel.org/doc/ols/2001/read-copy.pdf>，Linux 内核源代码 Documentation/RCU/ 也包含了 RCU 的一些讲解。

不同于自旋锁，使用 RCU 的读端没有锁、内存屏障、原子指令类的开销，几乎可以认为是直接读（只是简单地标明读开始和读结束），而 RCU 的写执行单元在访问它的共享资源前首先复制一个副本，然后对副本进行修改，最后使用一个回调机制在适当的时机把指向原来数据的指针重新指向新的被修改的数据，这个时机就是所有引用该数据的 CPU 都退出对共享数据读操作的时候。等待适当时机的这一时期称为宽限期（Grace Period）。

比如，有下面的一个由 struct foo 结构体组成的链表：

```
struct foo {
    struct list_head list;
    int a;
    int b;
    int c;
};
```

假设进程 A 要修改链表中某个节点 N 的成员 a、b。自旋锁的思路是排他性地访问这个链表，等所有其他持有自旋锁的进程或者中断把自旋锁释放后，进程 A 再拿到自旋锁访问链表并找到 N 节点，之后修改它的 a、b 两个成员，完成后解锁。而 RCU 的思路则不同，它直接制造一个新的节点 M，把 N 的内容复制给 M，之后在 M 上修改 a、b，并用 M 来代替 N 原本在链表的位置。之后进程 A 等待在链表前期已经存在的所有读端结束后（即宽限期，通

过下文说的 synchronize_rcu() API 完成), 再释放原来的 N。用代码来描述这个逻辑就是:

```

struct foo {
    struct list_head list;
    int a;
    int b;
    int c;
};

LIST_HEAD(head);

/* . . . */

p = search(head, key);
if (p == NULL) {
    /* Take appropriate action, unlock, and return. */
}
q = kmalloc(sizeof(*p), GFP_KERNEL);
*q = *p;
q->b = 2;
q->c = 3;
list_replace_rcu(&p->list, &q->list);
synchronize_rcu();
kfree(p);

```

RCU 可以看作读写锁的高性能版本, 相比读写锁, RCU 的优点在于既允许多个读执行单元同时访问被保护的数据, 又允许多个读执行单元和多个写执行单元同时访问被保护的数据。但是, RCU 不能替代读写锁, 因为如果写比较多时, 对读执行单元的性能提高不能弥补写执行单元同步导致的损失。因为使用 RCU 时, 写执行单元之间的同步开销会比较大, 它需要延迟数据结构的释放, 复制被修改的数据结构, 它也必须使用某种锁机制来同步并发的其他写执行单元的修改操作。

Linux 中提供的 RCU 操作包括如下 4 种。

1. 读锁定

```

rcu_read_lock()
rcu_read_lock_bh()

```

2. 读解锁

```

rcu_read_unlock()
rcu_read_unlock_bh()

```

使用 RCU 进行读的模式如下:

```

rcu_read_lock()
.../* 读临界区 */
rcu_read_unlock()

```

3. 同步 RCU

```
synchronize_rcu()
```

该函数由 RCU 写执行单元调用，它将阻塞写执行单元，直到当前 CPU 上所有的已经存在 (Ongoing) 的读执行单元完成读临界区，写执行单元才可以继续下一步操作。synchronize_rcu() 并不需要等待后续 (Subsequent) 读临界区的完成，如图 7.9 所示。

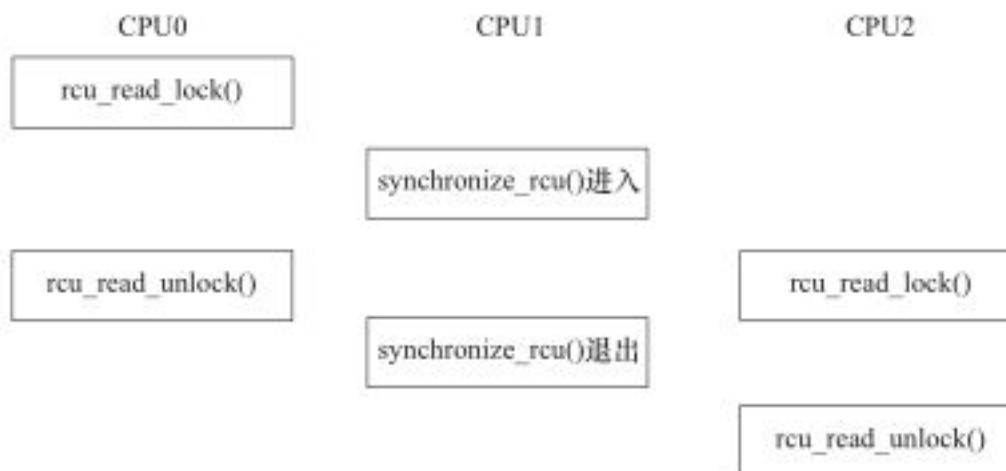


图 7.9 synchronize_rcu

探测所有的 rcu_read_lock() 被 rcu_read_unlock() 结束的过程很类似 Java 语言垃圾回收的工作。

4. 挂接回调

```
void call_rcu(struct rcu_head *head,
              void (*func)(struct rcu_head *rcu));
```

函数 call_rcu() 也由 RCU 写执行单元调用，与 synchronize_rcu() 不同的是，它不会使写执行单元阻塞，因而可以在中断上下文或软中断中使用。该函数把函数 func 挂接到 RCU 回调函数链上，然后立即返回。挂接的回调函数会在一个宽限期结束（即所有已经存在的 RCU 读临界区完成）后被执行。

```
rcu_assign_pointer(p, v)
```

给 RCU 保护的指针赋一个新的值。

```
rcu_dereference(p)
```

读端使用 rcu_dereference() 获取一个 RCU 保护的指针，之后既可以安全地引用它（访问它指向的区域）。一般需要在 rcu_read_lock()/rcu_read_unlock() 保护的区间引用这个指针，例如：

```
rcu_read_lock();
irq_rt = rcu_dereference(kvm->irq_routing);
if (irq < irq_rt->nr_rt_entries)
    hlist_for_each_entry(e, &irq_rt->map[irq], link) {
        if (likely(e->type == KVM_IRQ_ROUTING_MSI))
            ret = kvm_set_msi_inatomic(e, kvm);
        else
            ret = -EWOULDBLOCK;
    }
}
```

```

        break;
    }
rcu_read_unlock();

```

上述代码取自 `virt/kvm/irq_comm.c` 的 `kvm_set_irq_inatomic()` 函数。

```
rcu_access_pointer(p)
```

读端使用 `rcu_access_pointer()` 获取一个 RCU 保护的指针，之后并不引用它。这种情况下，我们只关心指针本身的价值，而不关心指针指向的内容。比如我们可以使用该 API 来判断指针是否为 NULL。

把 `rcu_assign_pointer()` 和 `rcu_dereference()` 结合起来使用，写端分配一个新的 `struct foo` 内存，并初始化其中的成员，之后把该结构体的地址赋值给全局的 `gp` 指针：

```

struct foo {
    int a;
    int b;
    int c;
};

struct foo *gp = NULL;

/* . . . */

p = kmalloc(sizeof(*p), GFP_KERNEL);
p->a = 1;
p->b = 2;
p->c = 3;
rcu_assign_pointer(gp, p);

```

读端访问该片区域：

```

rcu_read_lock();
p = rcu_dereference(gp);
if (p != NULL) {
    do_something_with(p->a, p->b, p->c);
}
rcu_read_unlock();

```

在上述代码中，我们可以把写端 `rcu_assign_pointer()` 看成发布（Publish）了 `gp`，而读端 `rcu_dereference()` 看成订阅（Subscribe）了 `gp`。它保证读端可以看到 `rcu_assign_pointer()` 之前所有内存被设置的情况（即 `gp->a`, `gp->b`, `gp->c` 等于 1、2、3 对于读端可见）。由此可见，与 RCU 相关的原语已经内嵌了相关的编译屏障或内存屏障。

对于链表数据结构而言，Linux 内核增加了专门的 RCU 保护的链表操作 API：

```
static inline void list_add_rcu(struct list_head *new, struct list_head *head);
```

该函数把链表元素 `new` 插入 RCU 保护的链表 `head` 的开头。

```
static inline void list_add_tail_rcu(struct list_head *new,
```

```
    struct list_head *head);
```

该函数类似于 list_add_rcu(), 它将把新的链表元素 new 添加到被 RCU 保护的链表的末尾。

```
static inline void list_del_rcu(struct list_head *entry);
```

该函数从 RCU 保护的链表中删除指定的链表元素 entry。

```
static inline void list_replace_rcu(struct list_head *old, struct list_head *new);
```

它使用新的链表元素 new 取代旧的链表元素 old。

```
list_for_each_entry_rcu(pos, head)
```

该宏用于遍历由 RCU 保护的链表 head, 只要在读执行单元临界区使用该函数, 它就可以安全地和其他 RCU 保护的链表操作函数(如 list_add_rcu())并发运行。

链表的写端代码模型如下：

```
struct foo {
    struct list_head list;
    int a;
    int b;
    int c;
};

LIST_HEAD(head);

/* . . . */

p = kmalloc(sizeof(*p), GFP_KERNEL);
p->a = 1;
p->b = 2;
p->c = 3;
list_add_rcu(&p->list, &head);
```

链表的读端代码则形如：

```
rcu_read_lock();
list_for_each_entry_rcu(p, head, list) {
    do_something_with(p->a, p->b, p->c);
}
rcu_read_unlock();
```

前面已经看到了对 RCU 保护链表中节点进行修改以及添加新节点的动作, 下面我们看一下 RCU 保护的链表删除节点 N 的工作。写端分为两个步骤, 第 1 步是从链表中删除 N, 之后等一个宽限期结束, 再释放 N 的内存。下面的代码分别用读写锁和 RCU 两种不同的方法来描述这一过程:

```
1 struct el {
2     struct list_head lp;
3     long key;
```

```
1 struct el {
2     struct list_head lp;
3     long key;
```

```

4  spinlock_t mutex;
5  int data;
6  /* Other data fields */
7 };
8 DEFINE_RWLOCK(listmutex);
9 LIST_HEAD(head);

1 int search(long key, int *result)
2 {
3     struct el *p;
4
5     read_lock(&listmutex);
6     list_for_each_entry(p, &head, lp) {
7         if (p->key == key) {
8             *result = p->data;
9             read_unlock(&listmutex);
10        return 1;
11    }
12 }
13 read_unlock(&listmutex);
14 return 0;
15 }

1 int delete(long key)
2 {
3     struct el *p;
4
5     write_lock(&listmutex);
6     list_for_each_entry(p, &head, lp) {
7         if (p->key == key) {
8             list_del(&p->lp);
9             write_unlock(&listmutex);
10            kfree(p);
11            return 1;
12        }
13 }
14 write_unlock(&listmutex);
15 return 0;
16 }

4  spinlock_t mutex;
5  int data;
6  /* Other data fields */
7 };
8 DEFINE_SPINLOCK(listmutex);
9 LIST_HEAD(head);

1 int search(long key, int *result)
2 {
3     struct el *p;
4
5     rCU_read_lock();
6     list_for_each_entry_rcu(p, &head, lp) {
7         if (p->key == key) {
8             *result = p->data;
9             rCU_read_unlock();
10            return 1;
11        }
12 }
13 rCU_read_unlock();
14 return 0;
15 }

1 int delete(long key)
2 {
3     struct el *p;
4
5     spin_lock(&listmutex);
6     list_for_each_entry(p, &head, lp) {
7         if (p->key == key) {
8             list_del_rcu(&p->lp);
9             spin_unlock(&listmutex);
10            synchronize_rcu();
11            kfree(p);
12            return 1;
13        }
14 }
15 spin_unlock(&listmutex);
16 return 0;
17 }

```

7.6 信号量

信号量 (Semaphore) 是操作系统中最典型的用于同步和互斥的手段，信号量的值可以是 0、1 或者 n 。信号量与操作系统中的经典概念 PV 操作对应。

P (S): ①将信号量 S 的值减 1，即 $S = S - 1$ ；

②如果 $S \geq 0$ ，则该进程继续执行；否则该进程置为等待状态，排入等待队列。

V (S): ①将信号量 S 的值加 1，即 $S = S + 1$ ；

②如果 $S > 0$ ，唤醒队列中等待信号量的进程。

Linux 中与信号量相关的操作主要有下面几种。

1. 定义信号量

下列代码定义名称为 sem 的信号量：

```
struct semaphore sem;
```

2. 初始化信号量

```
void sema_init(struct semaphore *sem, int val);
```

该函数初始化信号量，并设置信号量 sem 的值为 val。

3. 获得信号量

```
void down(struct semaphore * sem);
```

该函数用于获得信号量 sem，它会导致睡眠，因此不能在中断上下文中使用。

```
int down_interruptible(struct semaphore * sem);
```

该函数功能与 down 类似，不同之处为，因为 down() 进入睡眠状态的进程不能被信号打断，但因为 down_interruptible() 进入睡眠状态的进程能被信号打断，信号也会导致该函数返回，这时候函数的返回值非 0。

```
int down_trylock(struct semaphore * sem);
```

该函数尝试获得信号量 sem，如果能够立刻获得，它就获得该信号量并返回 0，否则，返回非 0 值。它不会导致调用者睡眠，可以在中断上下文中使用。

在使用 down_interruptible() 获取信号量时，对返回值一般会进行检查，如果非 0，通常立即返回 -ERESTARTSYS，如：

```
if (down_interruptible(&sem))
    return -ERESTARTSYS;
```

4. 释放信号量

```
void up(struct semaphore * sem);
```

该函数释放信号量 sem，唤醒等待者。

作为一种可能的互斥手段，信号量可以保护临界区，它的使用方式和自旋锁类似。与自旋锁相同，只有得到信号量的进程才能执行临界区代码。但是，与自旋锁不同的是，当获取不到信号量时，进程不会原地打转而是进入休眠等待状态。用作互斥时，信号量一般这样被使用：

进程 P ₁	进程 P ₂	进程 P _n
.....
P(S);	P(S);		P(S);

临界区；	临界区；	临界区；
<code>V(S);</code>	<code>V(S);</code>	<code>V(S);</code>
.....

由于新的Linux内核倾向于直接使用mutex作为互斥手段，信号量用作互斥不再被推荐使用。

信号量也可以用于同步，一个进程A执行`down()`等待信号量，另外一个进程B执行`up()`释放信号量，这样进程A就同步地等待了进程B。其过程类似：

进程 P1	进程 P2
代码区 C1;	<code>P(S);</code>
<code>V(S);</code>	代码区 C2;

此外，对于关心具体数值的生产者/消费者问题，使用信号量则较为合适。因为生产者/消费者问题也是一种同步问题。

7.7 互斥体

尽管信号量已经可以实现互斥的功能，但是“正宗”的mutex在Linux内核中还是真实地存在着。

下面代码定义了名为`my_mutex`的互斥体并初始化它：

```
struct mutex my_mutex;
mutex_init(&my_mutex);
```

下面的两个函数用于获取互斥体：

```
void mutex_lock(struct mutex *lock);
int mutex_lock_interruptible(struct mutex *lock);
int mutex_trylock(struct mutex *lock);
```

`mutex_lock()`与`mutex_lock_interruptible()`的区别和`down()`与`down_trylock()`的区别完全一致，前者引起的睡眠不能被信号打断，而后者可以。`mutex_trylock()`用于尝试获得mutex，获取不到mutex时不会引起进程睡眠。

下列函数用于释放互斥体：

```
void mutex_unlock(struct mutex *lock);
```

mutex的使用方法和信号量用于互斥的场合完全一样：

```
struct mutex my_mutex; /* 定义 mutex */
mutex_init(&my_mutex); /* 初始化 mutex */

mutex_lock(&my_mutex); /* 获取 mutex */
```

```
...           /* 临界资源 */
mutex_unlock(&my_mutex); /* 释放 mutex */
```

自旋锁和互斥体都是解决互斥问题的基本手段，面对特定的情况，应该如何取舍这两种手段呢？选择的依据是临界区的性质和系统的特点。

从严格意义上说，互斥体和自旋锁属于不同层次的互斥手段，前者的实现依赖于后者。在互斥体本身的实现上，为了保证互斥体结构存取的原子性，需要自旋锁来互斥。所以自旋锁属于更底层的手段。

互斥体是进程级的，用于多个进程之间对资源的互斥，虽然也是在内核中，但是该内核执行路径是以进程的身份，代表进程来争夺资源的。如果竞争失败，会发生进程上下文切换，当前进程进入睡眠状态，CPU 将运行其他进程。鉴于进程上下文切换的开销也很大，因此，只有当进程占用资源时间较长时，用互斥体才是较好的选择。

当所要保护的临界区访问时间比较短时，用自旋锁是非常方便的，因为它可节省上下文切换的时间。但是 CPU 得不到自旋锁会在那里空转直到其他执行单元解锁为止，所以要求锁不能在临界区里长时间停留，否则会降低系统的效率。

由此，可以总结出自旋锁和互斥体选用的 3 项原则。

1) 当锁不能被获取到时，使用互斥体的开销是进程上下文切换时间，使用自旋锁的开销是等待获取自旋锁（由临界区执行时间决定）。若临界区比较小，宜使用自旋锁，若临界区很大，应使用互斥体。

2) 互斥体所保护的临界区可包含可能引起阻塞的代码，而自旋锁则绝对要避免用来保护包含这样代码的临界区。因为阻塞意味着要进行进程的切换，如果进程被切换出去后，另一个进程企图获取本自旋锁，死锁就会发生。

3) 互斥体存在于进程上下文，因此，如果被保护的共享资源需要在中断或软中断情况下使用，则在互斥体和自旋锁之间只能选择自旋锁。当然，如果一定要使用互斥体，则只能通过 mutex_trylock() 方式进行，不能获取就立即返回以避免阻塞。

7.8 完成量

Linux 提供了完成量（Completion，关于这个名词，至今没有好的翻译，笔者将其译为“完成量”），它用于一个执行单元等待另一个执行单元执行完某事。

Linux 中与完成量相关的操作主要有以下 4 种。

1. 定义完成量

下列代码定义名为 my_completion 的完成量：

```
struct completion my_completion;
```

2. 初始化完成量

下列代码初始化或者重新初始化 my_completion 这个完成量的值为 0（即没有完成的状态）：

```
init_completion(&my_completion);
reinit_completion(&my_completion)
```

3. 等待完成量

下列函数用于等待一个完成量被唤醒：

```
void wait_for_completion(struct completion *c);
```

4. 唤醒完成量

下面两个函数用于唤醒完成量：

```
void complete(struct completion *c);
void complete_all(struct completion *c);
```

前者只唤醒一个等待的执行单元，后者释放所有等待同一完成量的执行单元。

完成量用于同步的流程一般如下：

进程 P1	进程 P2
代码区 C1;	wait_for_completion(&done);
complete(&done);	
	代码区 C2;

7.9 增加并发控制后的 globalmem 的设备驱动

在 globalmem() 的读写函数中，由于要调用 copy_from_user()、copy_to_user() 这些可能导致阻塞的函数，因此不能使用自旋锁，宜使用互斥体。

驱动工程师习惯将某设备所使用的自旋锁、互斥体等辅助手段也放在设备结构中，因此，可如代码清单 7.4 那样修改 globalmem_dev 结构体的定义，并在模块初始化函数中初始化这个信号量，如代码清单 7.5 所示。

代码清单 7.4 增加并发控制后的 globalmem 设备结构体

```
1 struct globalmem_dev {
2     struct cdev cdev;
3     unsigned char mem[GLOBALMEM_SIZE];
4     struct mutex mutex;
5 };
```

代码清单 7.5 增加并发控制后的 globalmem 设备驱动模块加载函数

```
1 static int __init globalmem_init(void)
2 {
3     int ret;
4     dev_t devno = MKDEV(globalmem_major, 0);
5
6     if (globalmem_major)
```

```

7         ret = register_chrdev_region(devno, 1, "globalmem");
8     else {
9         ret = alloc_chrdev_region(&devno, 0, 1, "globalmem");
10        globalmem_major = MAJOR(devno);
11    }
12    if (ret < 0)
13        return ret;
14
15    globalmem_devp = kzalloc(sizeof(struct globalmem_dev), GFP_KERNEL);
16    if (!globalmem_devp) {
17        ret = -ENOMEM;
18        goto fail_malloc;
19    }
20
21    mutex_init(&globalmem_devp->mutex);
22    globalmem_setup_cdev(globalmem_devp, 0);
23    return 0;
24
25 fail_malloc:
26    unregister_chrdev_region(devno, 1);
27    return ret;
28 }
29 module_init(globalmem_init);

```

在访问 `globalmem_dev` 中的共享资源时，需先获取这个互斥体，访问完成后，随即释放这个互斥体。驱动中新的 `globalmem` 读、写操作如代码清单 7.6 所示。

代码清单 7.6 增加并发控制后的 `globalmem` 读、写操作

```

1 static ssize_t globalmem_read(struct file *filp, char __user * buf, size_t size,
2                               loff_t * ppos)
3 {
4     unsigned long p = *ppos;
5     unsigned int count = size;
6     int ret = 0;
7     struct globalmem_dev *dev = filp->private_data;
8
9     if (p >= GLOBALMEM_SIZE)
10         return 0;
11     if (count > GLOBALMEM_SIZE - p)
12         count = GLOBALMEM_SIZE - p;
13
14     mutex_lock(&dev->mutex);
15
16     if (copy_to_user(buf, dev->mem + p, count)) {
17         ret = -EFAULT;
18     } else {
19         *ppos += count;
20         ret = count;
21     }

```

```

22     printk(KERN_INFO "read %u bytes(s) from %lu\n", count, p);
23 }
24
25 mutex_unlock(&dev->mutex);
26
27 return ret;
28 }
29
30 static ssize_t globalmem_write(struct file *filp, const char __user * buf,
31                               size_t size, loff_t * ppos)
32 {
33     unsigned long p = *ppos;
34     unsigned int count = size;
35     int ret = 0;
36     struct globalmem_dev *dev = filp->private_data;
37
38     if (p >= GLOBALMEM_SIZE)
39         return 0;
40     if (count > GLOBALMEM_SIZE - p)
41         count = GLOBALMEM_SIZE - p;
42
43     mutex_lock(&dev->mutex);
44
45     if (copy_from_user(dev->mem + p, buf, count))
46         ret = -EFAULT;
47     else {
48         *ppos += count;
49         ret = count;
50
51         printk(KERN_INFO "written %u bytes(s) from %lu\n", count, p);
52     }
53
54     mutex_unlock(&dev->mutex);
55
56     return ret;
57 }

```

代码第 14 行和第 43 行用于获取互斥体，代码第 25 和 54 行用于在对临界资源访问结束后释放信号量。

除了 globalmem 的读、写操作之外，如果在读、写的同时，另一个执行单元执行 MEM_CLEAR IO 控制命令，也会导致全局内存的混乱，因此，globalmem_ioctl() 函数也需被重写，如代码清单 7.7 所示。

代码清单 7.7 增加并发控制后的 globalmem 设备驱动 ioctl() 函数

```

1 static long globalmem_ioctl(struct file *filp, unsigned int cmd,
2                             unsigned long arg)
3 {
4     struct globalmem_dev *dev = filp->private_data; /* ??µ?éè±??álli????? */

```

```
5
6     switch (cmd) {
7         case MEM_CLEAR:
8             mutex_lock(&dev->mutex);
9             memset(dev->mem, 0, GLOBALMEM_SIZE);
10            mutex_unlock(&dev->mutex);
11
12            printk(KERN_INFO "globalmem is set to zero\n");
13            break;
14
15        default:
16            return -EINVAL;
17    }
18
19    return 0;
20 }
```

增加并发控制后 globalmem 的完整驱动位于本书虚拟机的例子 /kernel/drivers/globalmem/ch7 目录下，其使用方法与第 6 章 globalmem 驱动在用户空间的验证一致。

7.10 总结

并发和竞态广泛存在，中断屏蔽、原子操作、自旋锁和互斥体都是解决并发问题的机制。中断屏蔽很少单独被使用，原子操作只能针对整数进行，因此自旋锁和互斥体应用最为广泛。

自旋锁会导致死循环，锁定期间不允许阻塞，因此要求锁定的临界区小。互斥体允许临界区阻塞，可以适用于临界区大的情况。