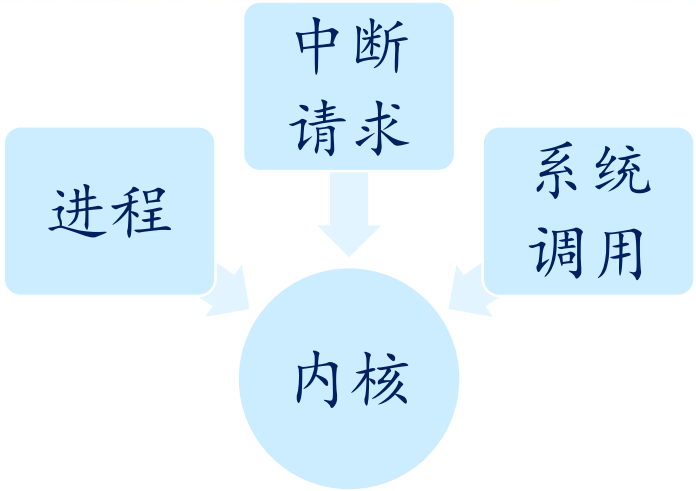
7.1 内核同步机制

西安邮电大学

内核同步引入

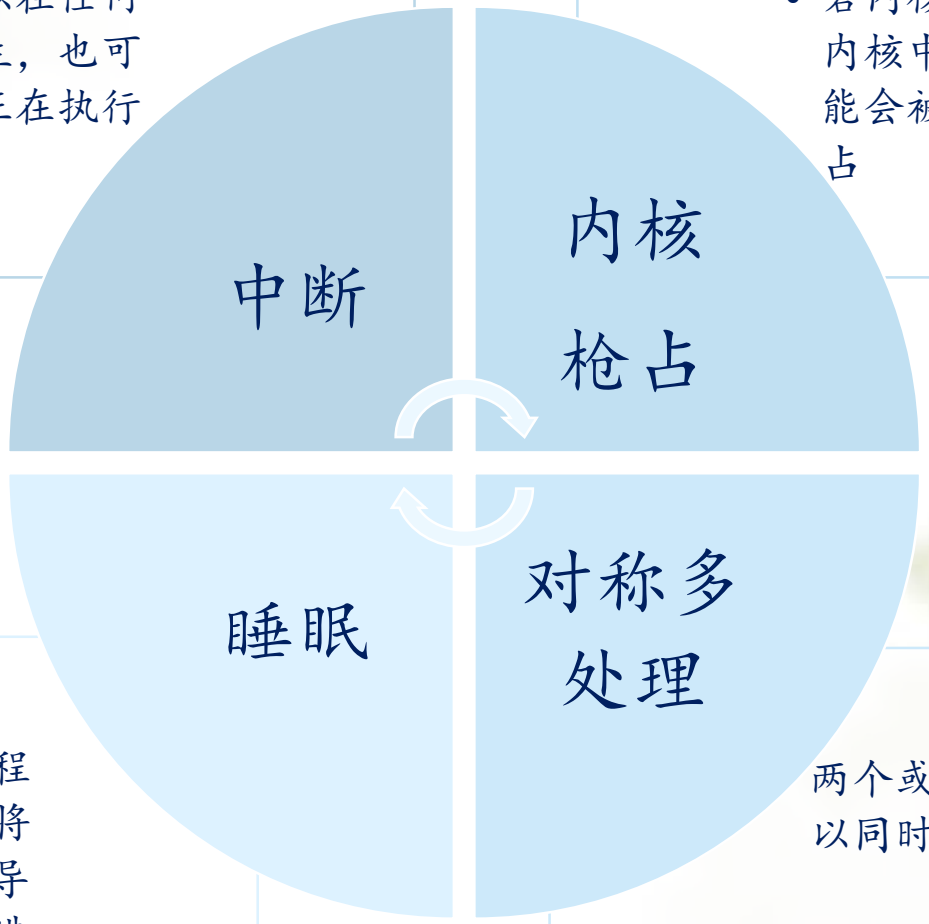


- 如果我们把内核看作不断对各种请求进行响应的服务器，那么，正在CPU上执行的进程、发出中断请求的外部设备等就相当于客户。正如服务器要随时响应客户的请求一样，内核也会随时响应进程、中断、系统调用等的请求。我们之所以这样比喻是为了强调内核中的各个任务，并不是严格按着顺序依次执行的，而是相互交错执行的。
- 对所有内核任务而言，内核中的很多数据都是共享资源，这就像高速公路供很多车辆行驶一样。对这些共享资源的访问必须遵循一定的访问规则，否则就可能造成对共享资源的破坏，就如同不遵守交通规则会造成撞车一样。

并发执行的原因

- 中断几乎可以在任何时刻异步发生，也可能随时打断正在执行的代码。

- 若内核具有抢占性，内核中的任务就可能会被另一任务抢占



- 在内核执行的进程可能会睡眠，这将唤醒调度程序，导致调度一个新的进程执行

两个或多个处理器可以同时执行代码

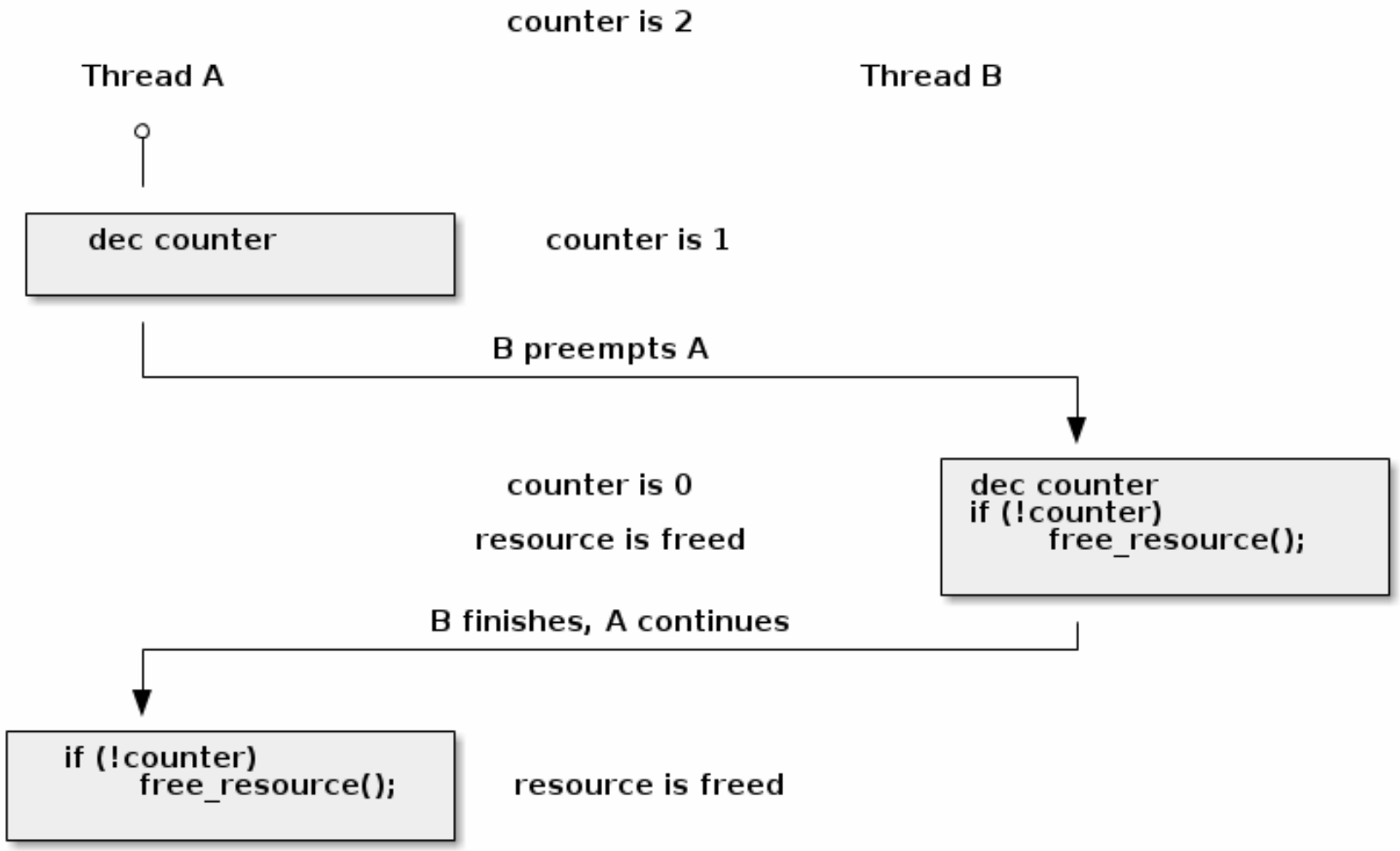
问题：系统中还有哪些并发源？

竞争条件（Race conditions）

- 当以下两个条件同时发生时，竞争发生：
  - （1）至少有两个可执行上下文“并行执行”
    - a. 真正的并行（比如，两个系统调用在不同的处理器上执行）
    - b. 其中一个上下文能随意枪占另一个（比如，一个中断枪占系统调用）
  - （2）可执行上下文对共享内存变量执行“读写”访问

- 问题：为什么这些竞争条件会导致各种难以调试的错误？

竞争条件导致的错误


counter is 2
Thread A
Thread B
dec counter
counter is 1
B preempts A
counter is 0
resource is freed
dec counter
if (!counter)
free_resource();
B finishes, A continues
if (!counter)
free_resource();
resource is freed

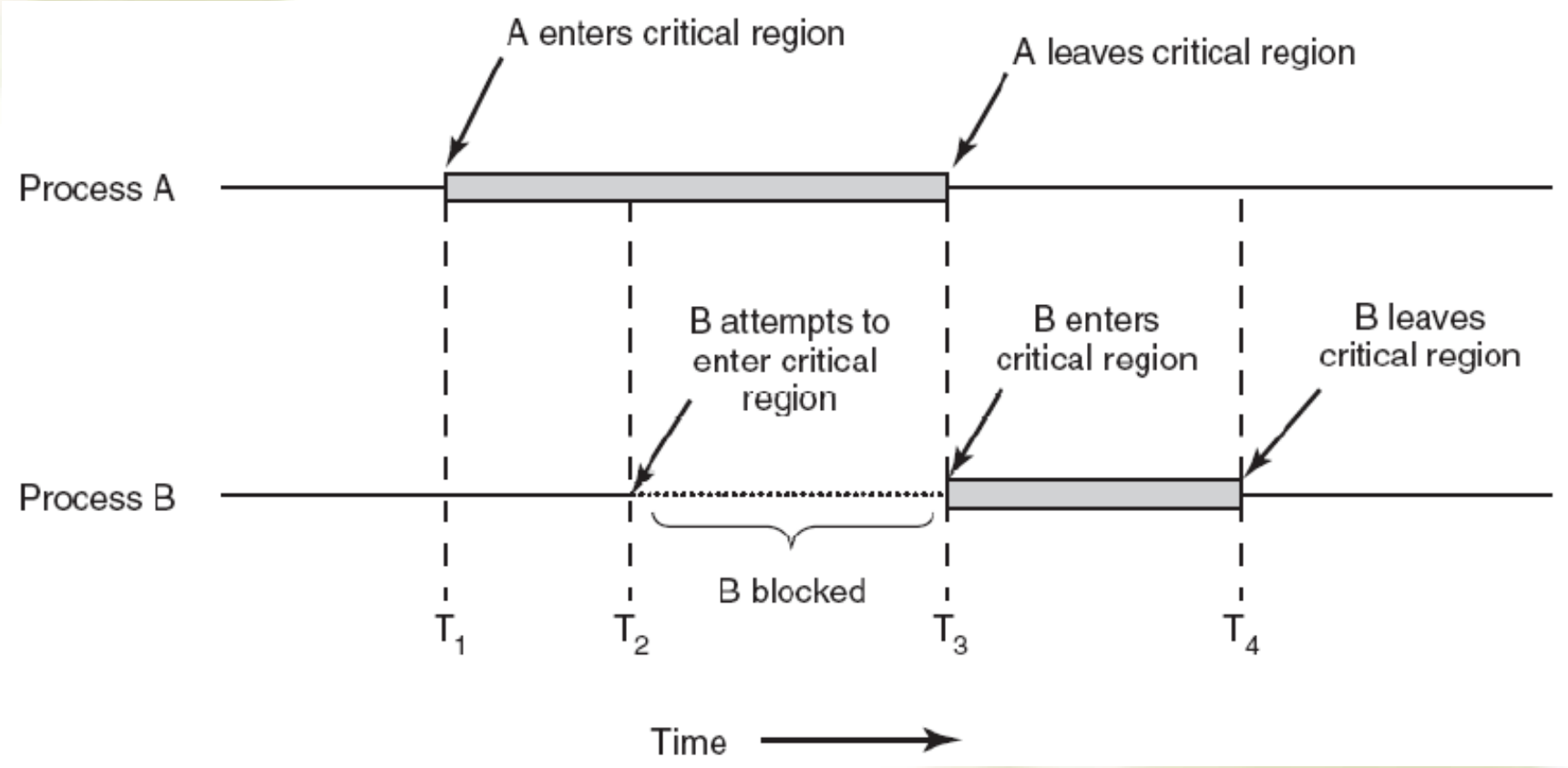
竞争条件导致的错误

- 举一个释放资源的例子。
- 在大多数情况下， release_resource()函数将只释放一次资源。然而，在图中，如果counter的初值为2，当线程A将该变量的值刚被减为1，B线程枪占A， counter的初值被减为0，则线程B调用释放函数释放资源，然后，线程A恢复执行，因为counter的值为0，线程A也释放资源，出现一个资源被释放两次的错误。
- 问题：如何解决？

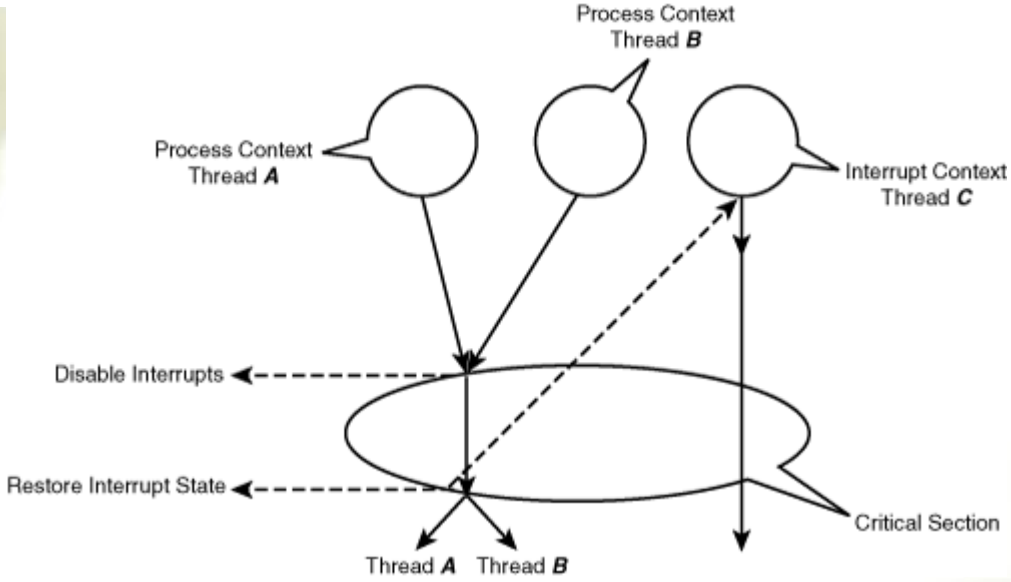
临界区

- 这里引出同步中的重要概念-临界区。什么是临界区？所谓临界区（critical regions）就是访问和操作共享数据的代码段。多个内核任务并发访问同一个资源通常是不安全的。为了避免对临界区进行并发访问，编程者必须保证临界区代码被原子地执行。也就是说，代码在执行期间不可被打断，就如同整个临界区是一个不可分割的指令一样。如图，A进程进入临界区后，B试图进入的时候被阻塞，只有当A离开临界区后，B才能进入。
- 问题：前面释放资源的例子中，临界区是什么？

临界区

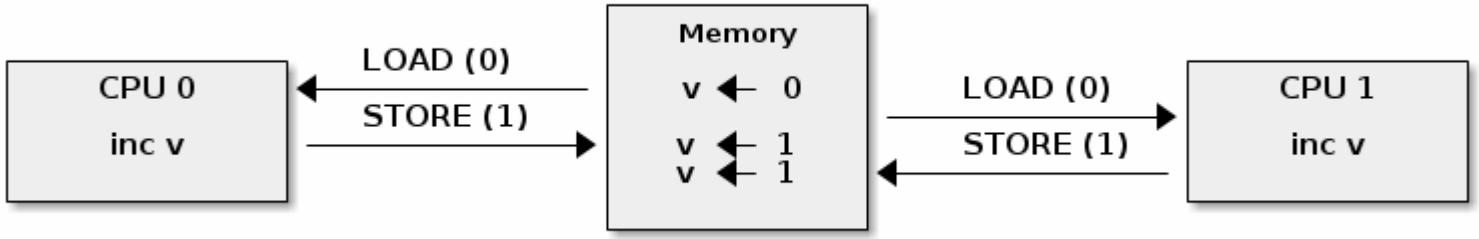

A enters critical region
A leaves critical region
Process A
B attempts to enter critical region
B enters critical region
B leaves critical region
Process B
B blocked
T1
T2
T3
T4
Time

保护临界区的措施



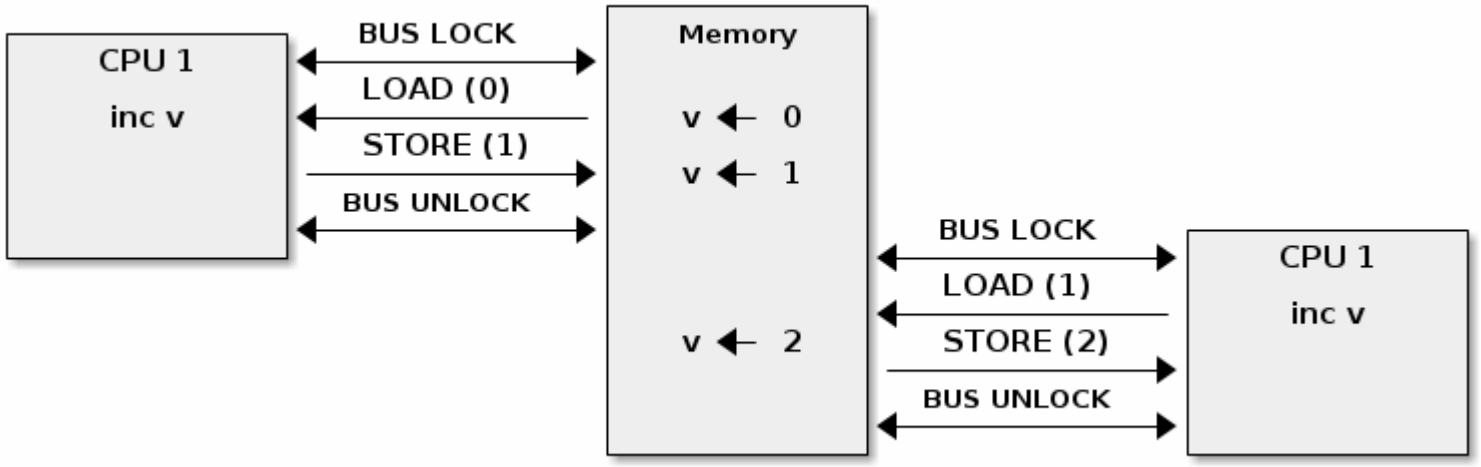
- （1）使临界区的操作原子地进行（例如，使用原子指令）
- （2）进入临界区后禁止枪占(例如，通过禁止中断、禁止下半部处理程序或者线程枪占等）
- （3）串行的访问临界区（例如使用自选锁、互斥锁只允许一个内核任务访问临界区）

并发执行中共享变量v加1操作



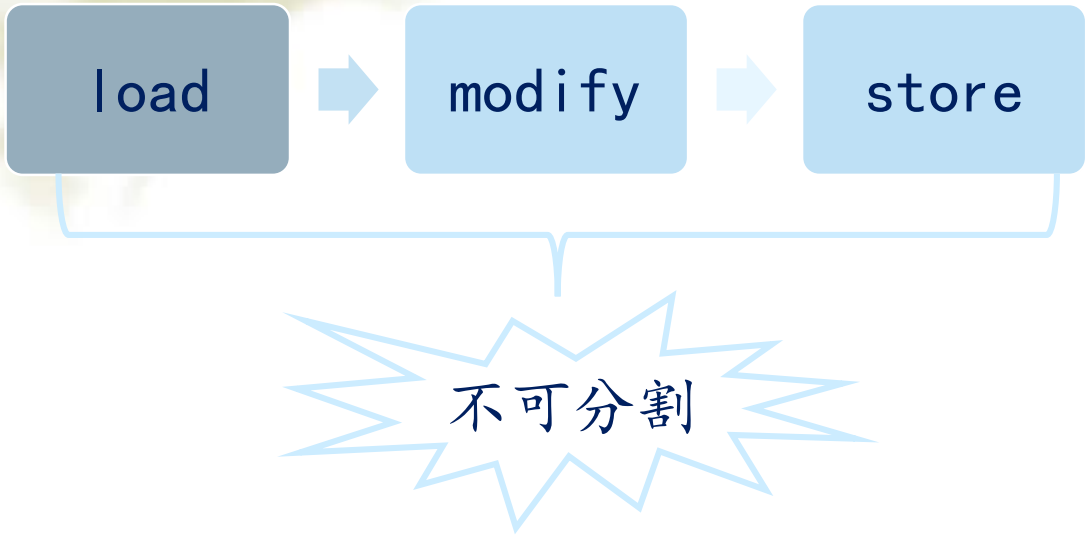
- 多个CPU和内存是通过总线互联的，在任意时刻，只能有一个总线主设备（例如CPU、DMA 控制器）访问该从设备（在这个场景中，从设备是RAM芯片）。因此，来自两个CPU上的读内存操作被串行化执行，分别获得了同样的旧值（0）。完成修改后，两个CPU都想进行写操作，把修改的值写回到内存。但是，硬件仲裁的限制使得CPU的写回必须是串行化的，因此CPU1首先获得了访问权，进行写回动作，随后，CPU2完成写回动作。在这种情况下，CPU1的对内存的修改被CPU2的操作覆盖了，因此执行结果是错误的（本来v两次加1后为2，结果成了1了）。
- 问题：在单CPU上，假设一个系统调用和一个中断服务程序并发执行，则对V的加1操作会出现什么情况？

并发执行中共享变量v加1操作



- 在SMP系统中，为了提供原子操作，不同CPU体系结构提供了不同的技术，例如，在X86下，当执行加有LOCK前缀的指令时，LOCK前缀用于锁定系统总线，使得前面的错误不会发生。
- 问题： ARM平台上采用什么指令？

原子操作



- 对于那些由多个内核任务进行共享的变量，其load-modify-store必须原子地进行，也就是不能分割（如图）。

原子类型

```
typedef struct {
    int counter;
} atomic_t;
```

- 于是内核提供了一个特殊的类型atomic_t，具体定义为：
- 从上面的定义来看，atomic_t实际上就是一个int类型的counter。
- 问题：为什么原子类型的定义要把一个整型放在结构体中？

原子操作的API

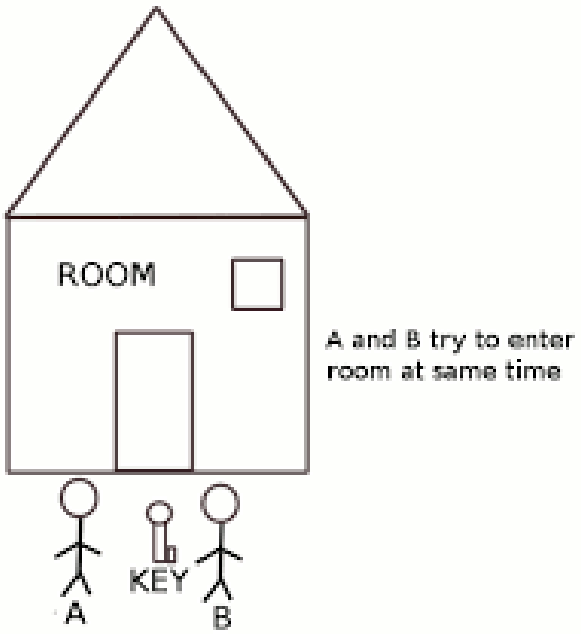
接口函数	描述
static inline void atomic_add(int i, atomic_t *v)	给一个原子变量v增加i
static inline void atomic_sub(int i, atomic_t *v)	给一个原子变量v减去i
atomic_read	获取原子变量的值
atomic_set	设定原子变量的值
atomic_inc(v)	原子变量的值加1
atomic_inc_return(v)	同上，只不过将变量v的最新值返回
atomic_sub_and_test(i, v)	给一个原子变量v减去i，并判断变量v的最新值是否等于0
atomic_dec_and_test(atomic_t *v);	对原子变量执行自减，减操作后 ，测试其是否为0，为 0 则返回 true，否则返回 false ：

原子操作举例

```
void release_resource()
{
    if
(atomic_dec_and_test(&counter))
        free_resource();
}
```

- 我们使用atomic_dec_and_test()实现资源计数器counter的减1并检查，这两个操作原子地进行。
- 问题：在多核系统中遇到原子操作，在系统层面上原子操作还是是原子的吗？在核级还是原子的吗？

共享队列和加锁



- 当共享资源是一个复杂的数据结构时，竞争状态往往会使该数据结构遭到破坏。
- 对于这种情况，锁机制可以避免竞争状态正如门锁和门一样，门后的房间可想象成一个临界区。
- 在一个指定时间内，房间里只能有个一个内核任务存在，当一个任务进入房间后，它会锁住身后的房门；当它结束对共享数据的操作后，就会走出房间，打开门锁。如图，A和B试图同时进入房间，当一个任务进去后就必须加锁，出来后打开锁。

共享队列和加锁

任务 1

试图锁定队列

成功：获得锁

访问队列…

为队列解除锁

…

任务2

试图锁定队列

失败：等待…

等待…

等待…

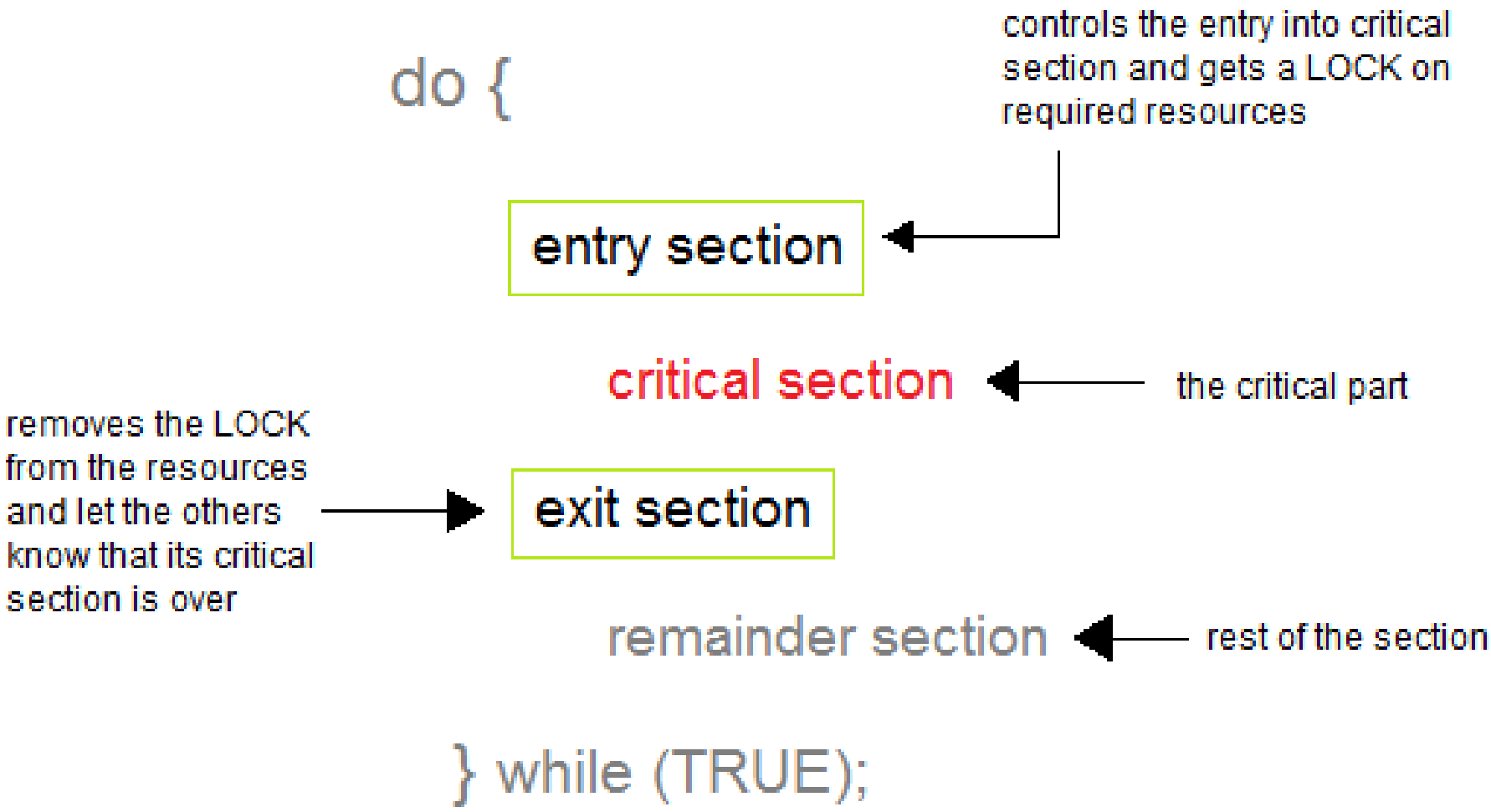
成功：获得锁

访问队列…

为队列解除锁

- 任何要访问队列的代码首先都需要占住相应的锁，这样该锁就能阻止来自其它内核任务的并发访问：

确定保护对象


do {
controls the entry into critical section and gets a LOCK on required resources
entry section
critical section
the critical part
removes the LOCK from the resources and let the others know that its critical section is over
exit section
remainder section
rest of the section
} while (TRUE);

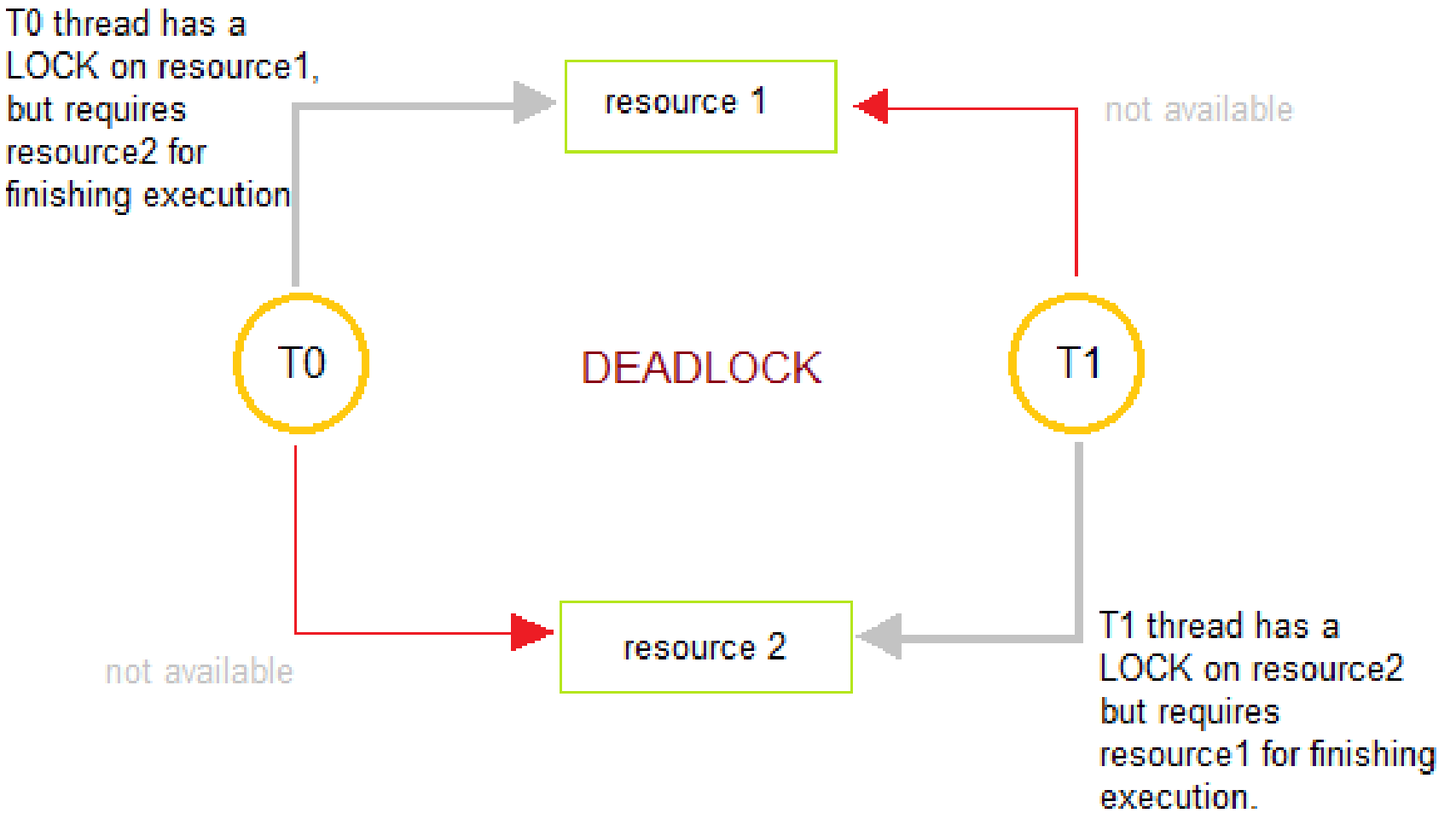
确定保护对象

- 找出哪些数据需要保护是关键所在，也就是要找出谁是临界区。
- 内核任务的局部数据仅仅被它本身访问，显然不需要保护。
- 如果数据只会被特定的进程访问，也不需加锁。
- 大多数内核数据结构都需要加锁，也就是说它们是临界区。

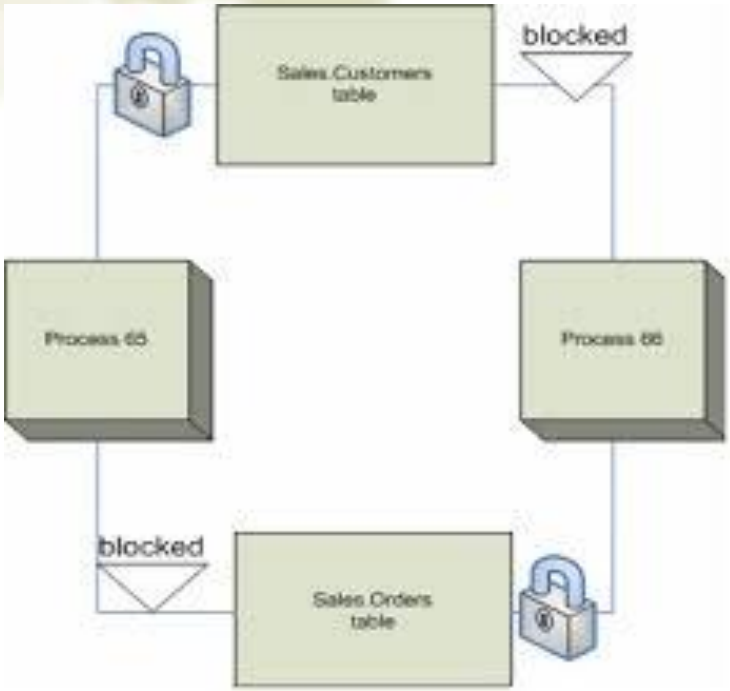
死　锁

- 所有任务都在相互等待，但它们永远不会释放已经占有的资源，于是任何任务都无法继续，这种情况就是死锁。
- 典型的死锁：
  - 四路交通堵塞
  - 自死锁：一个执行任务试图去获得一个自己已经持有的锁

死　锁

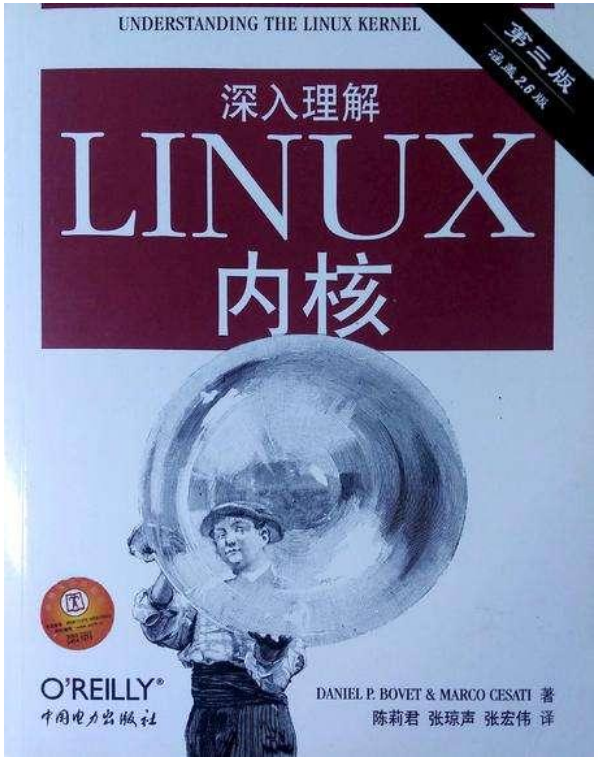

T0 thread has a LOCK on resource1, but requires resource2 for finishing execution
resource 1
not available
T0
DEADLOCK
T1
not available
resource 2
T1 thread has a LOCK on resource2 but requires resource1 for finishing execution.

死锁的避免



- 加锁的顺序是关键。使用嵌套的锁时必须保证以相同的顺序获取锁，这样可以阻止致命拥抱类型的死锁。
- 防止发生饥饿
- 不要重复请求同一个锁。
- 越复杂的加锁方案越有可能造成死锁，因此设计应力求简单

参考资料



深入理解Linux内核 第五章

带着思考离开

死锁是一种小概率事件还是大概率事件，如果内核出现死锁，该如何应对？

谢谢大家！

THANK YOU