北京郵電大學
Beijing University of Posts and Telecommunications

操作系统内核
-基于Linux

第8讲
内核同步

主讲：杨文川

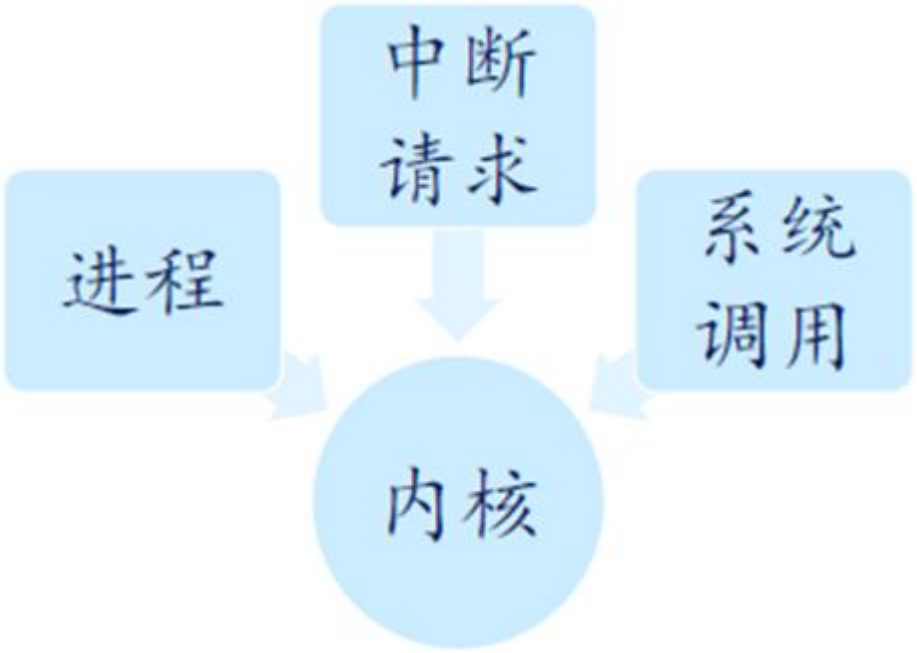
内容

- 1 内核同步概述
- 2 原子操作
- 3 内核同步机制
- 4 信号量
- 5 实践-内核多任务并发实例

内容导航：

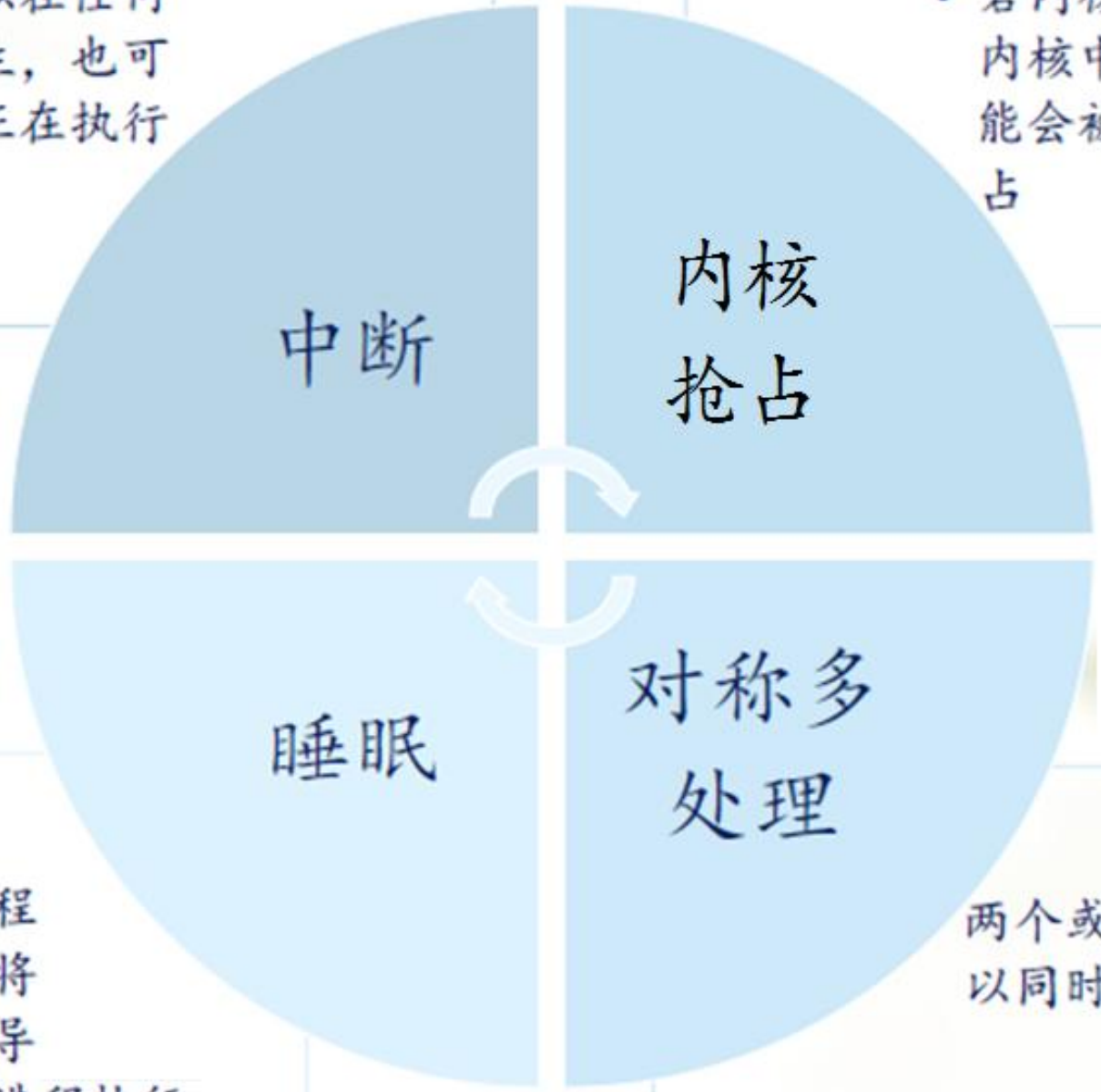
1 内核同步机制

内核同步引入



- 如果我们把内核看作服务器，不断对各种请求进行响应，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。
- 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

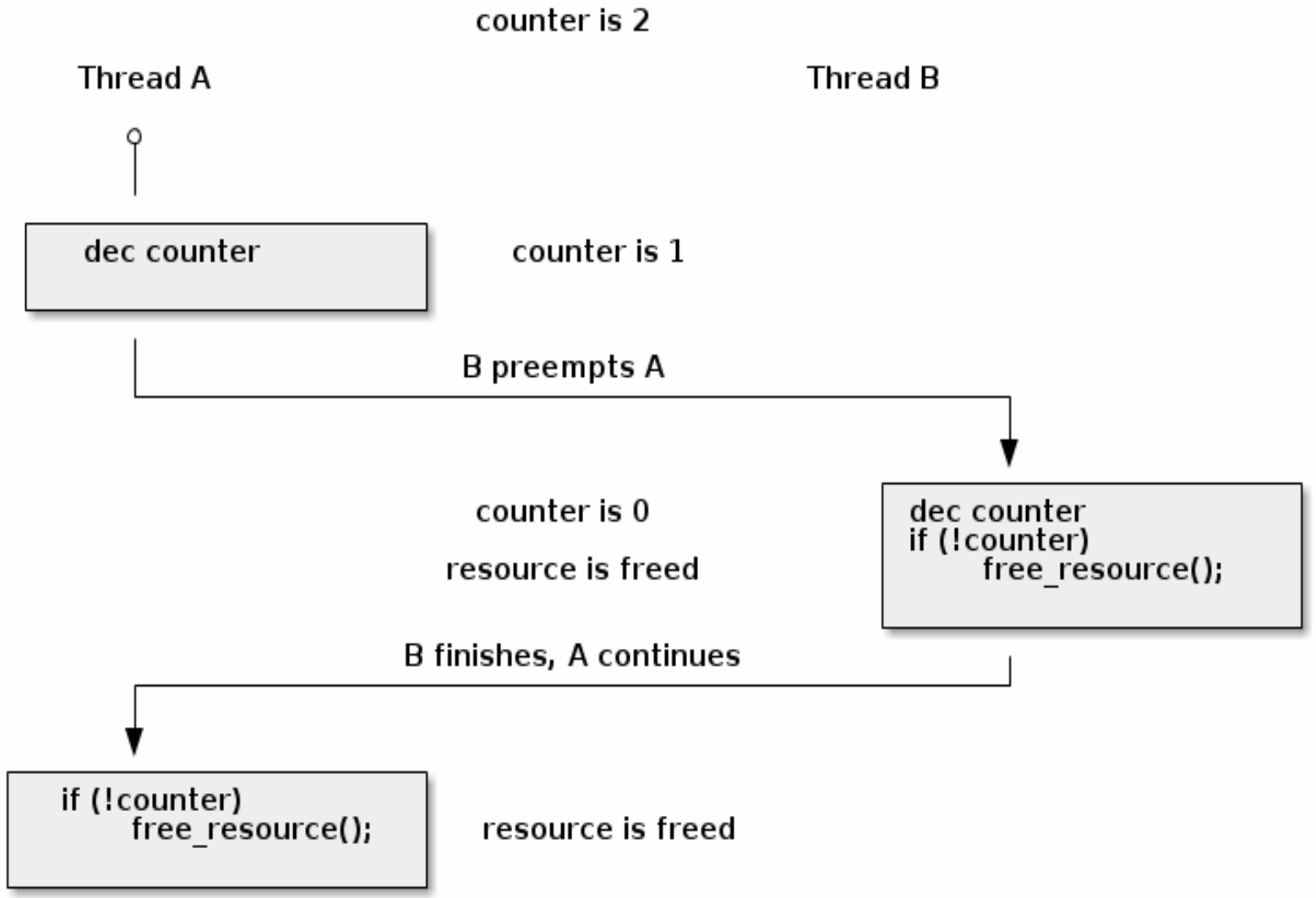
竞争条件(Race conditions)

- 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抢占另一个(比如,一个中断抢占系统调用)
- (2)可执行上下文对共享内存变量执行“读写”访问

- 下面举一个释放资源的例子。

竞争条件导致的错误

- 在大多数情况下，release_resource()函数将只释放一次资源。
- 然而，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线程抢占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。

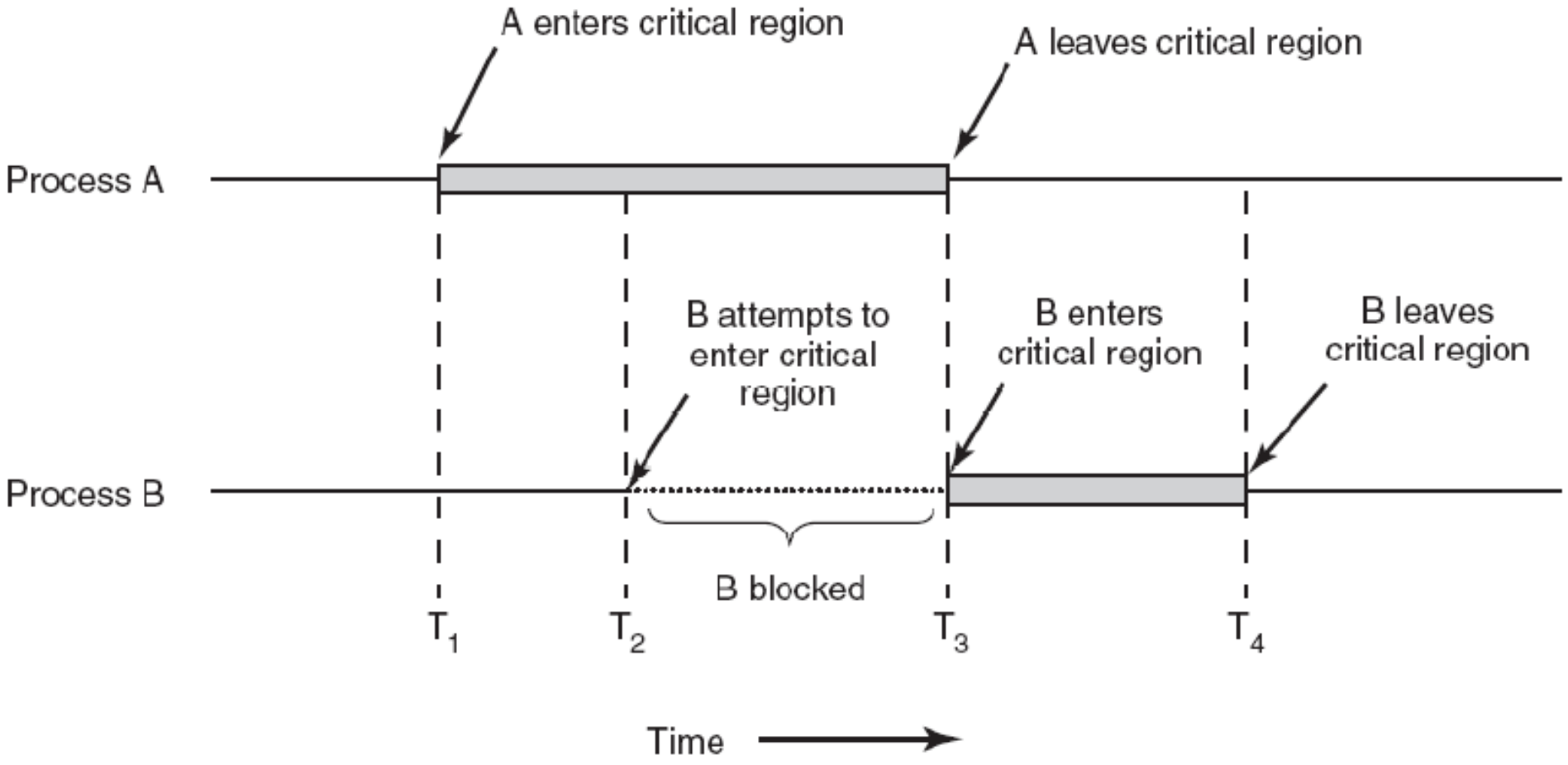


临界区

- 这里引出同步中的重要概念-临界区，那么什么是临界区？
- 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

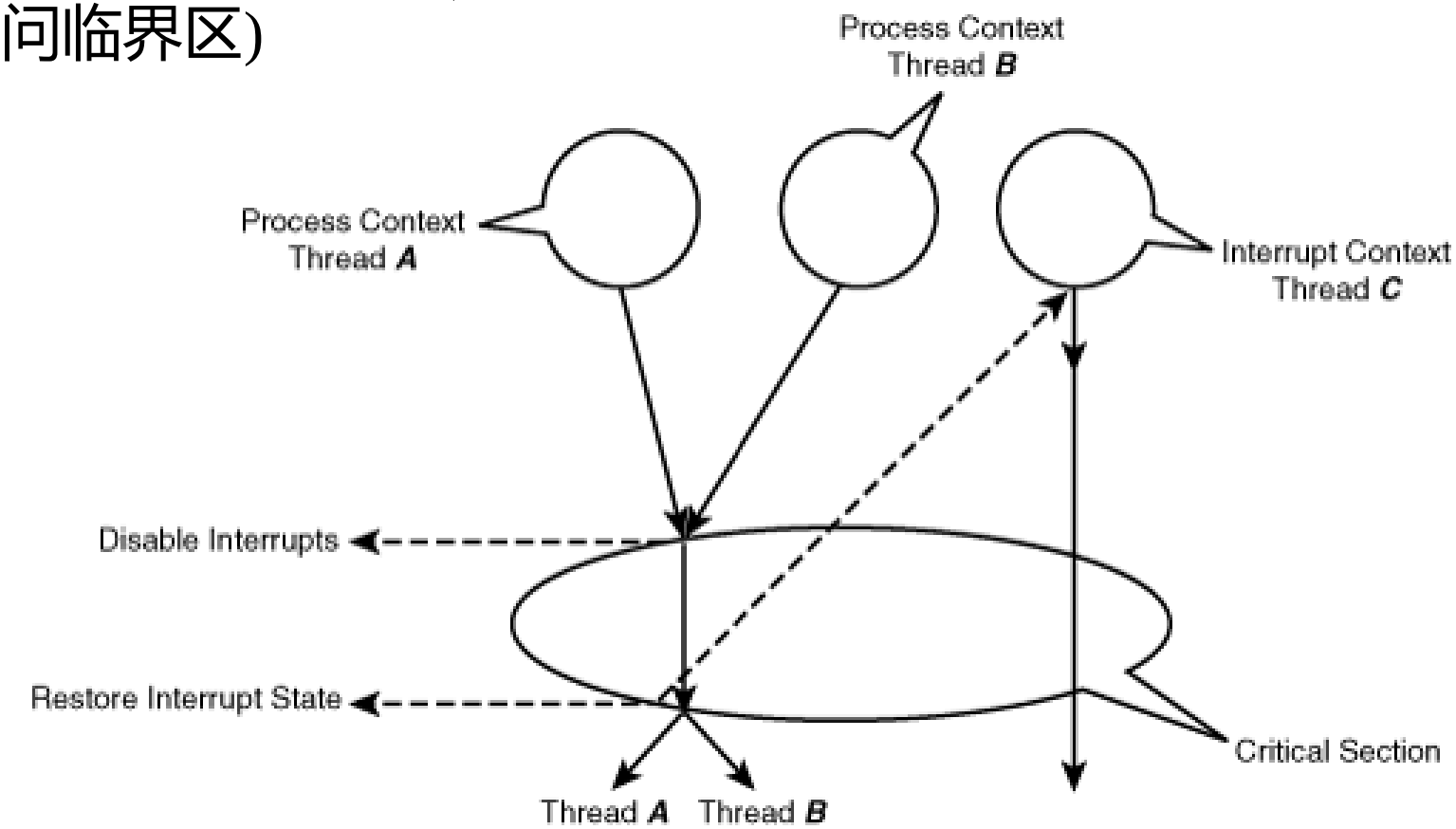
临界区

- 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。

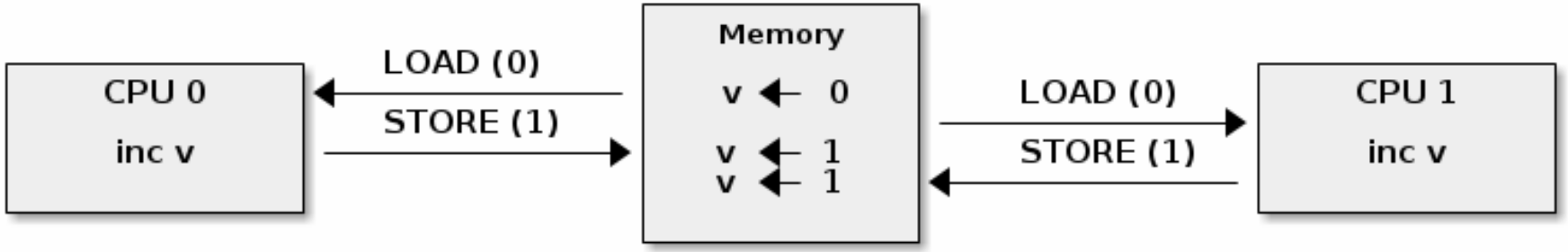


保护临界区的措施

- (1)使临界区的操作原子地进行(例如，使用原子指令)
- (2)进入临界区后禁止抢占(例如，通过禁止中断、禁止下半部处理程序或者线程抢占等)
- (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)



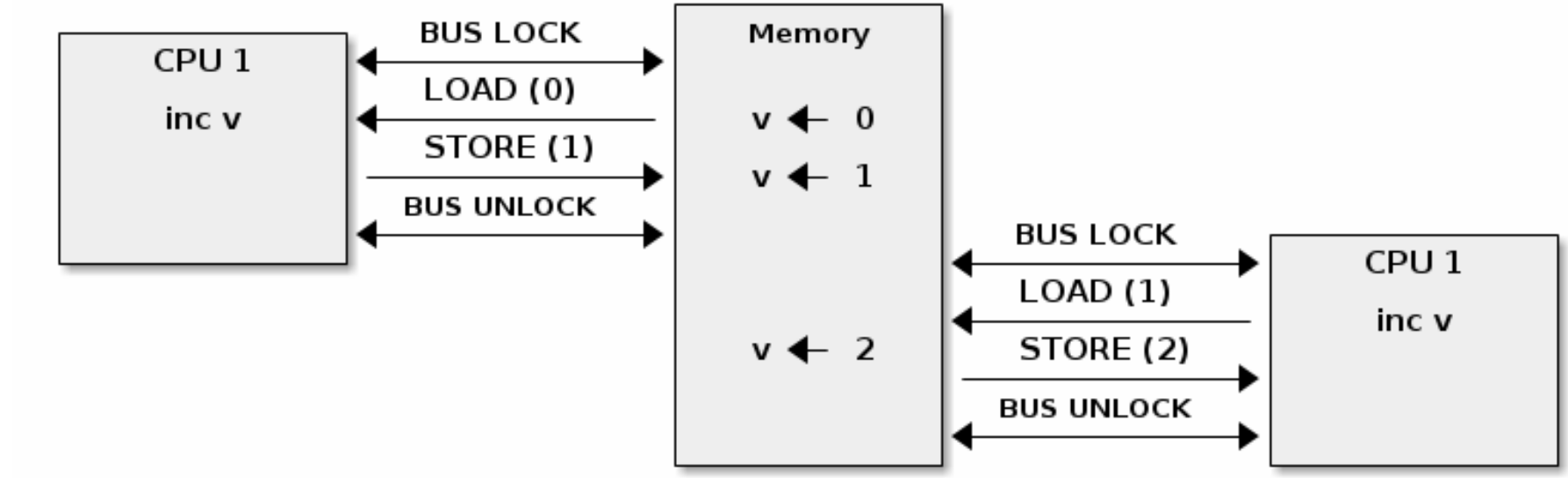
并发执行中 共享变量v加1操作



- 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)。

并发执行中 共享变量v加1操作1

- 在 SMP（ Symmetrical Multi-Processing）**对称多处理系统** 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。

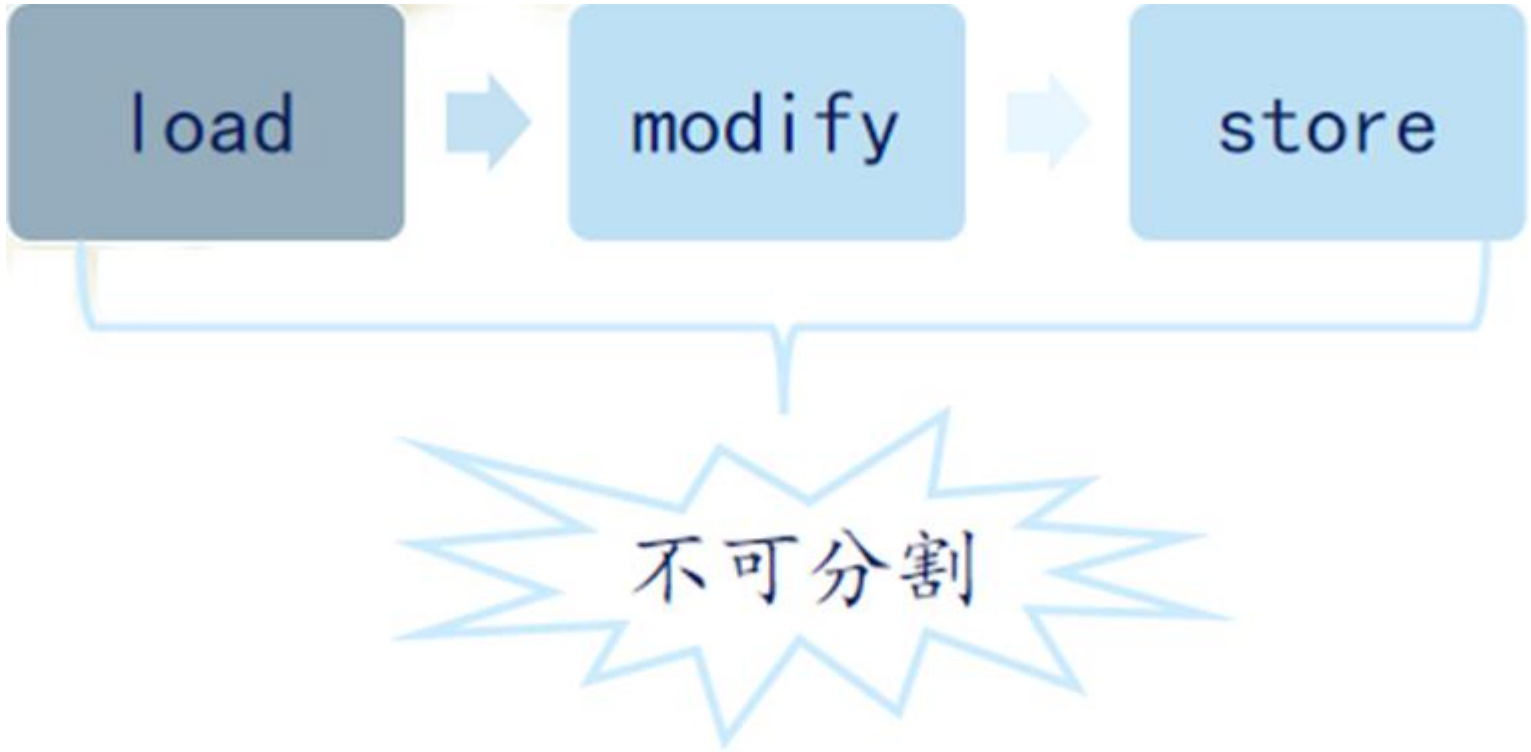


内容导航：

2 原子操作

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)。



原子类型及例子

- 内核提供了一个特殊的类型atomic_t，具体定义为：
-
```
typedef struct {
        int counter;
} atomic_t;
```

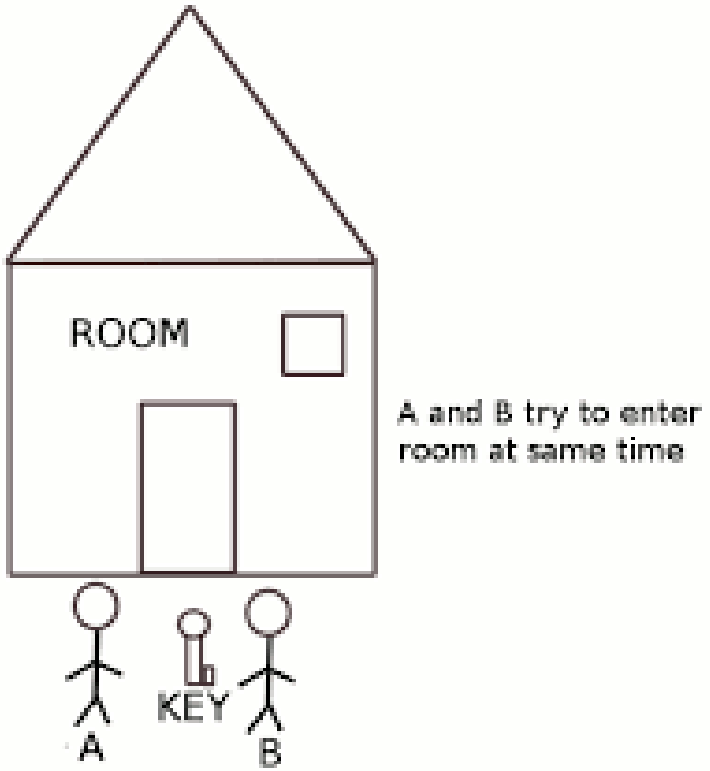
  - 在 include\linux\types.h
- 从上面的定义来看，atomic_t实际上就是一个int类型的counter。
- 下面是一个原子操作举例，我们使用atomic_dec_and_test() 实现资源计数器counter的减1并检查，这两个操作原子地进行。

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

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 ：

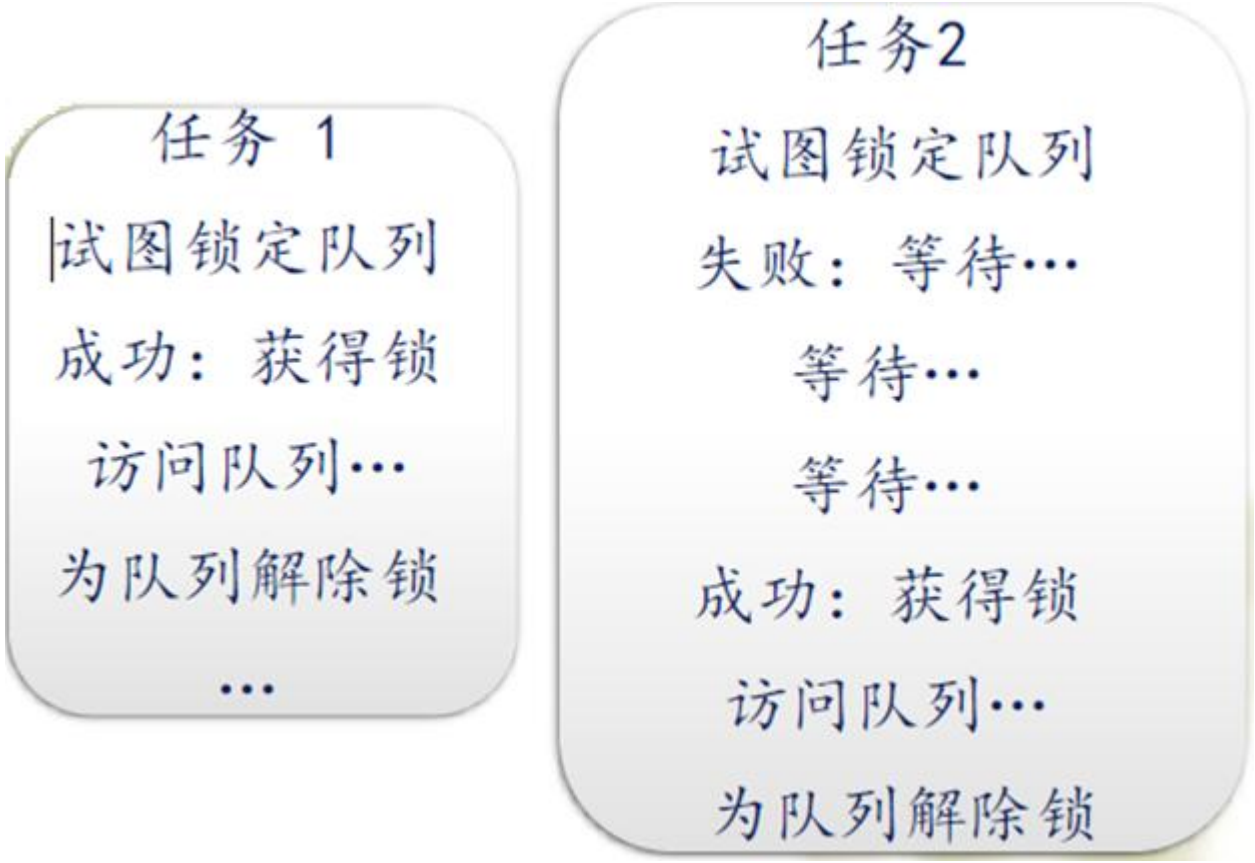
共享队列和加锁



- 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。

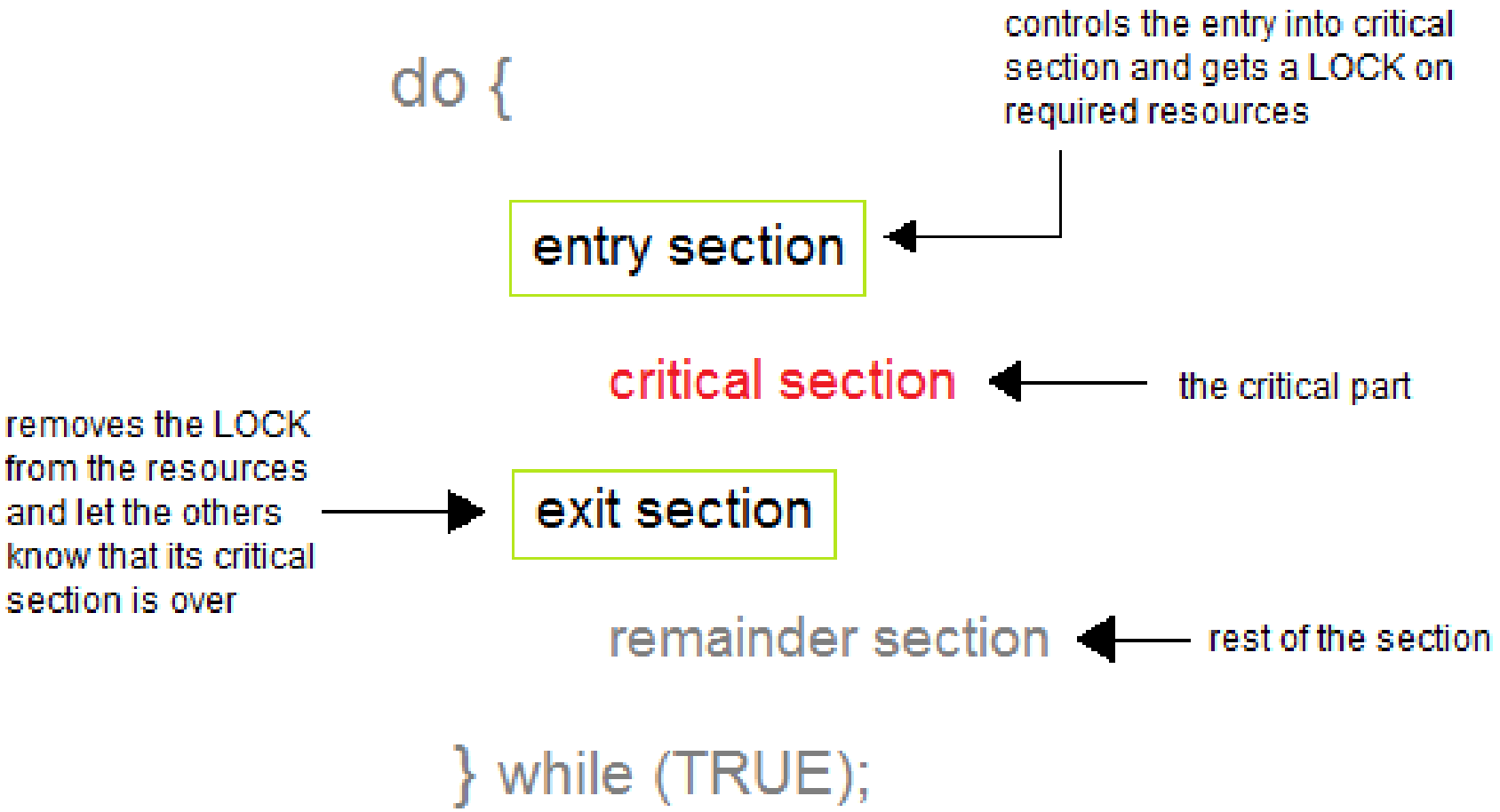
共享队列和加锁

- 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：



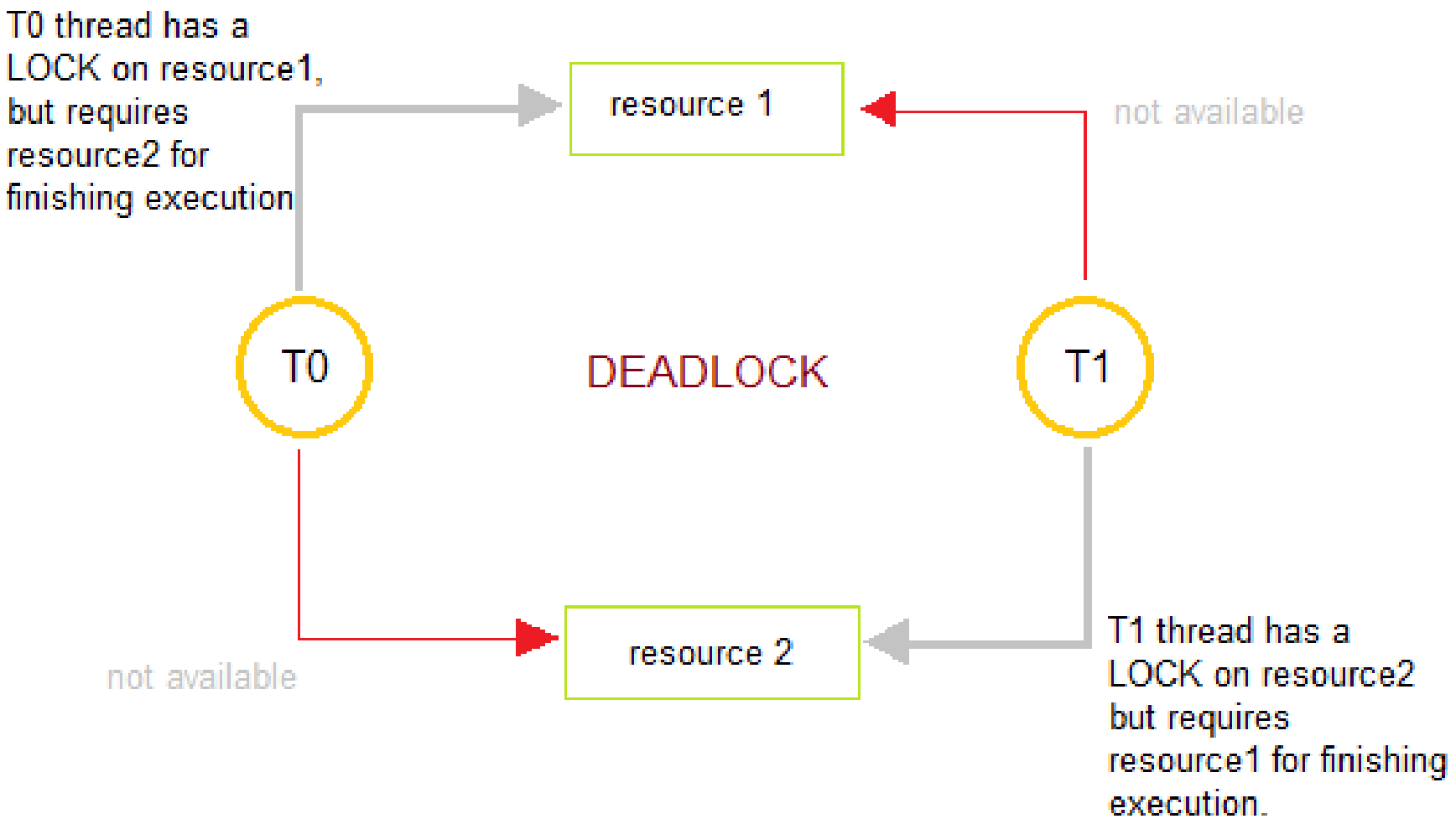
确定保护对象

- 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。

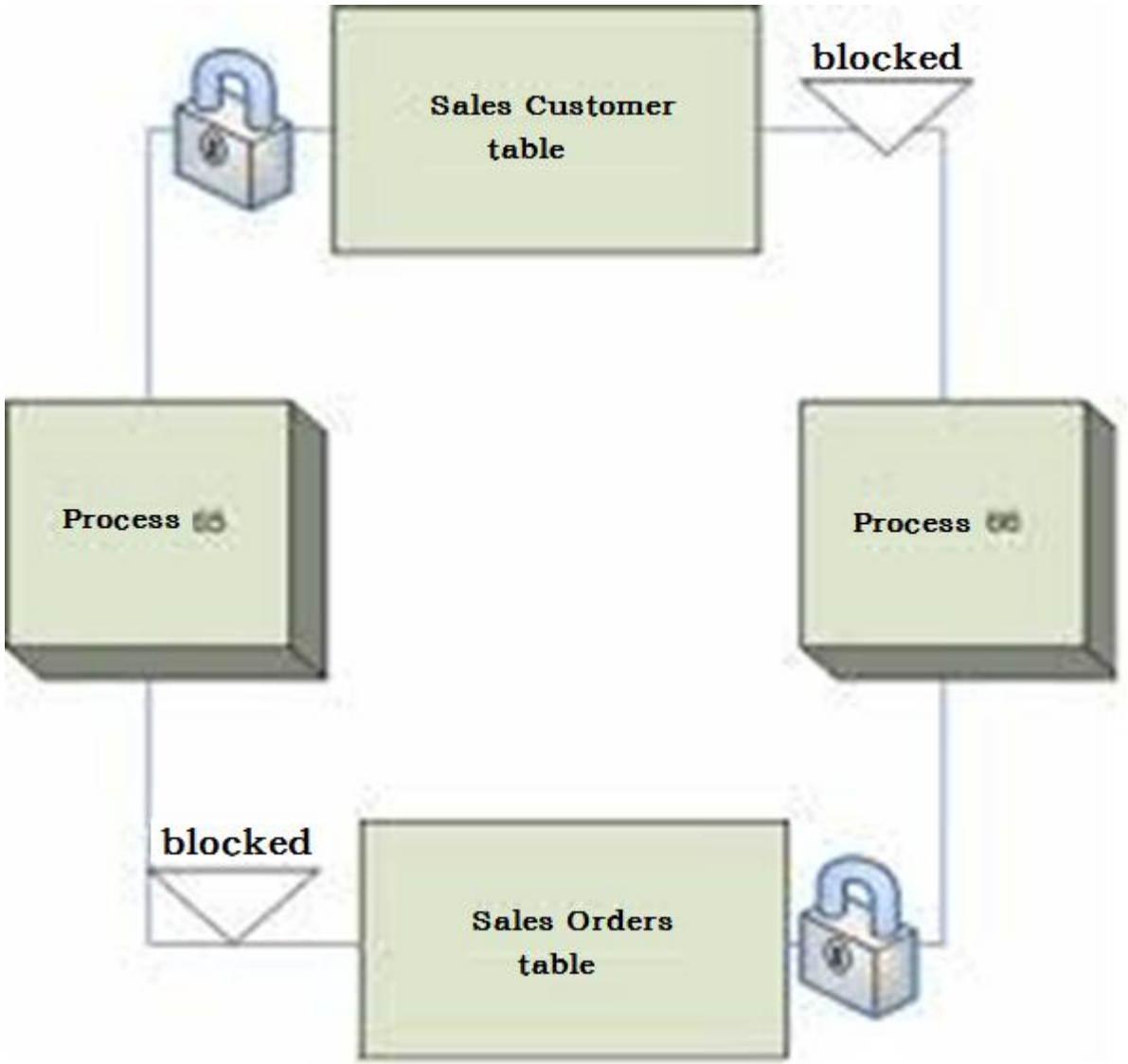


死锁

- 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：如果每一个停止的车，都决心等待其他的车开动后，自己再启动
- 自死锁：一个执行任务试图去获得一个自己已经持有的锁

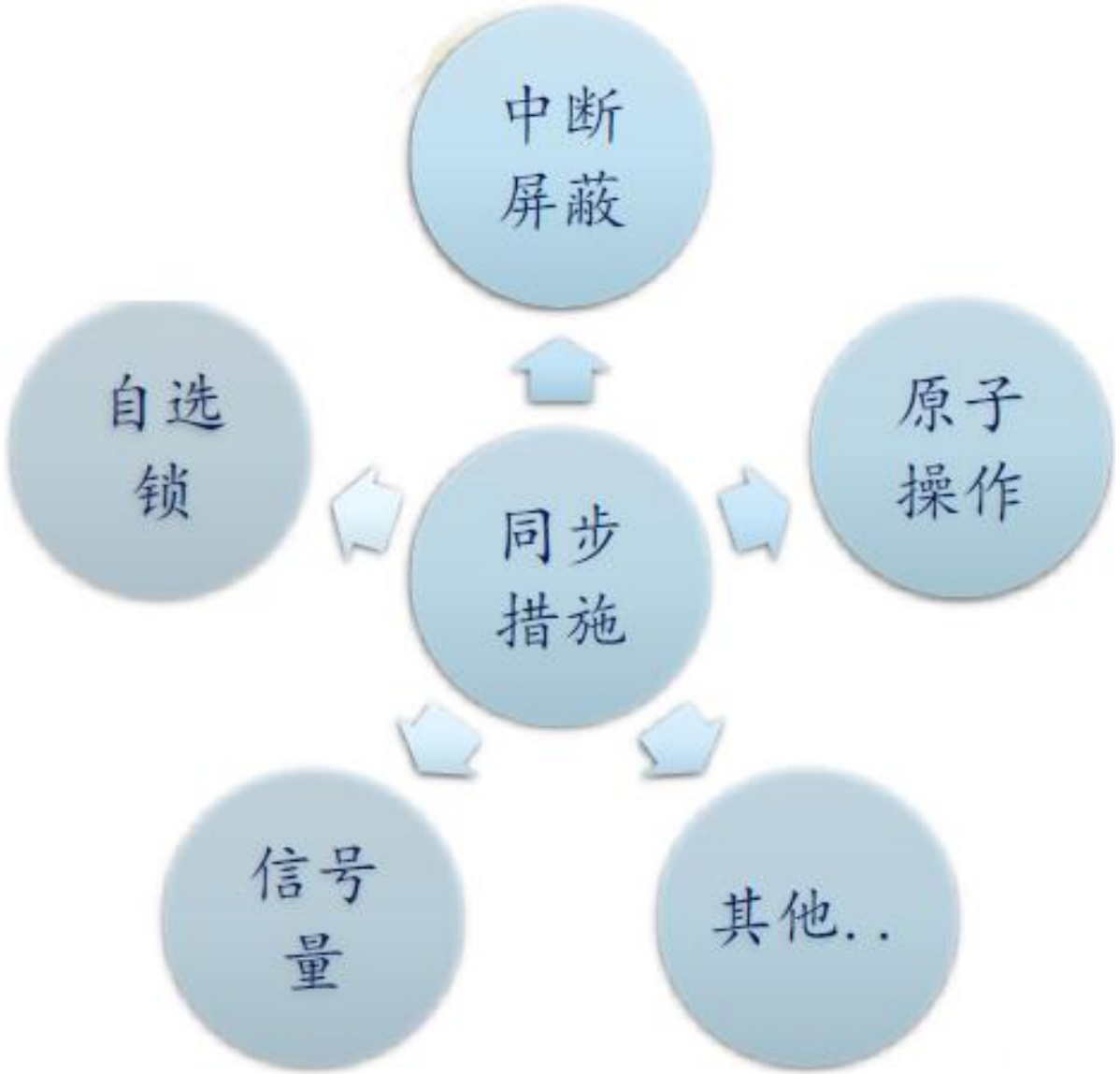


死锁的避免



- 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，设计应力求简单

3 内核同步措施



内核同步措施

- 为了避免并发，防止竞争。
- 内核提供了一组同步方法，来提供对共享数据的保护，
- 原子操作在前面已经介绍过了，下面介绍其他几种。

中断屏蔽

- 在进入临界区之前屏蔽系统的中断，从而保证正在执行的内核任务不被中断处理程序所抢占，防止某些静态条件的发生。在退出临界区后，重新打开中断。

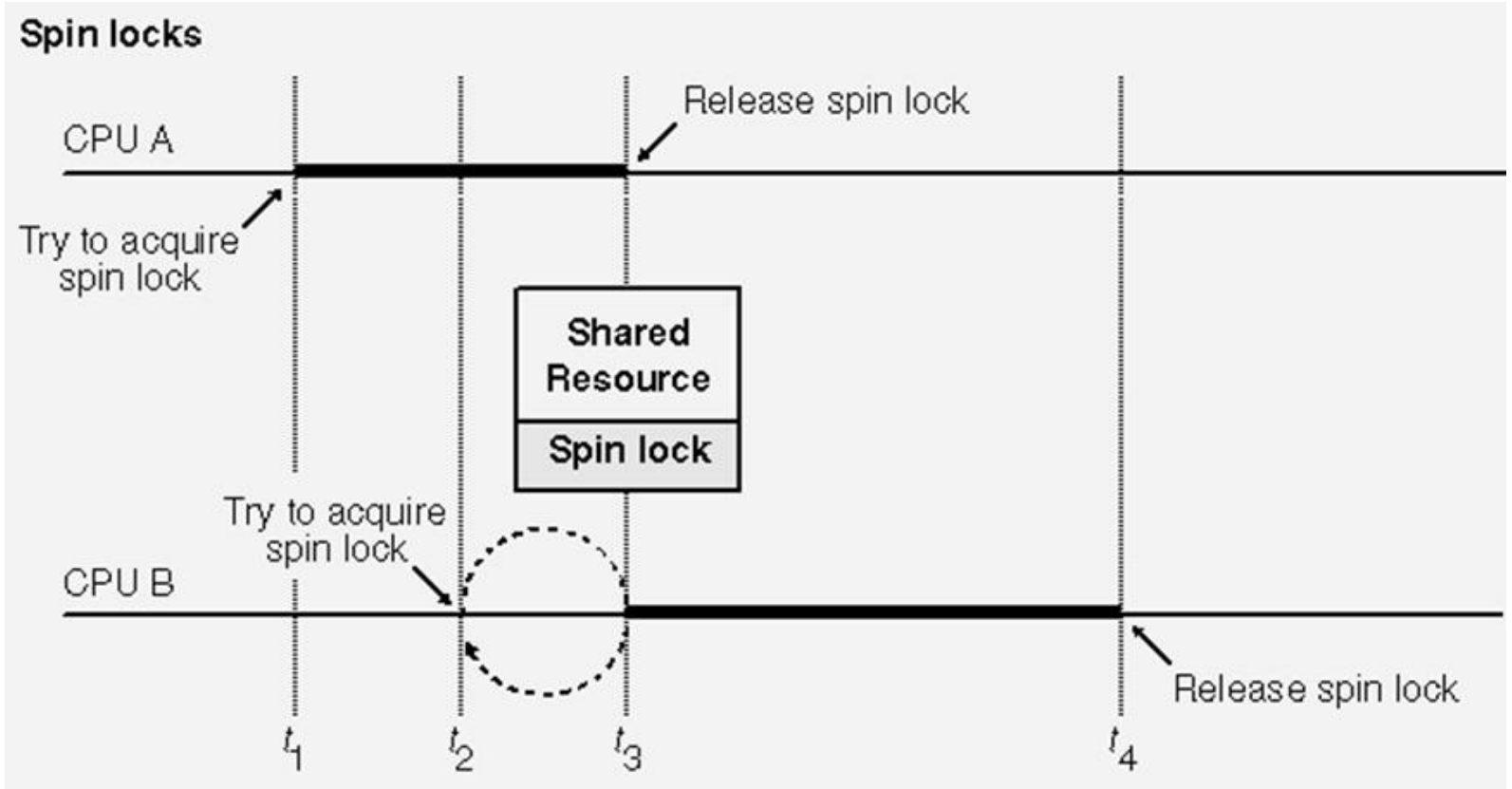
使用
local_irq_disable() //屏蔽中断
... //临界区
local_irq_enable() //开中断

中断屏蔽的缺点

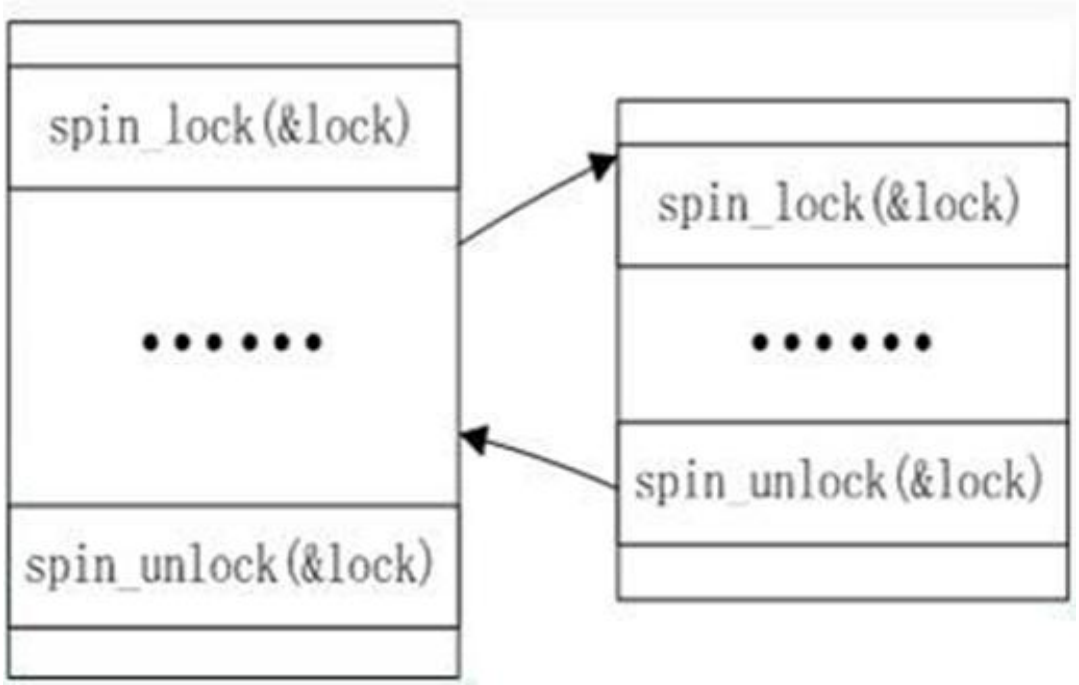
- local_irq_disable()和local_irq_enable()这两个函数都只能禁止和开启本地CPU内的中断，并不能解决多处理器引发的竞态(并行)。
- 在屏蔽中断期间，所有的中断都无法得到处理
  - 禁止/打开当前处理器上所有的中断：
    - local_irq_save()/local_irq_restore()
    - local_irq_disable()/local_irq_enable()
- 因此长时间屏蔽中断是很危险的，有可能造成数据丢失甚至系统崩溃

自旋锁

- 自旋锁是专为防止多处理器并发而引入的一种锁，它在内核中大量应用于中断处理等部分，而对于单处理器来说，可简单采用关闭中断的方式就可以防止中断处理程序的并发执行
- 自旋锁最多只能被一个内核任务持有，如图中，在t1时刻，CPU A上的任务试图获得自选锁，并且成功获得，进入临界区，t2时刻，CPU B上的任务试图申请自旋锁，这个任务就会一直进行忙循环，也就是旋转，直到t3时刻锁被释放。



自旋锁



- 设计自旋锁的初衷，是在短期间内进行轻量级的锁定，所以自旋锁不应该被持有时间过长。
- 自旋锁在内核中，主要用来防止多处理器中，并发访问临界区，防止内核抢占造成的竞争
- 自旋锁不允许任务睡眠，持有自旋锁的任务睡眠，会造成自死锁，
- 因此自旋锁能够在中断上下文中使用。

- 自旋锁的定义如下，在include/linux/spinlock_types.h：

```
typedef struct spinlock {
        union {
                struct raw_spinlock rlock;
#ifdef CONFIG_DEBUG_LOCK_ALLOC
        ...
#endif
        };
} spinlock_t;
```

- 使用自旋锁的基本形式如下：
  - DEFINE_SPINLOCK(mr_lock);/*定义一个自旋锁*/
  - spin_lock(&mr_lock);
  - /*临界区*/
  - spin_unlock(&mr_lock);
- 自旋锁还有很多变种，内核提供了一组API，此处不详述

4 信号量

信号量

- Linux中的信号量是一种睡眠锁。若有一个任务试图获得一个已被持有的信号量时，信号量会将其推入等待队列，然后让其睡眠。
- 这时处理器获得自由，而去执行其它代码。
- 当持有信号量的进程，将信号量释放后，在等待队列中的一个任务将被唤醒，从而便可以获得这个信号量
- 信号量具有睡眠特性，适用于锁会被长时间持有的情况，只能在进程上下文中使用。

内核中如何定义信号量

```
struct semaphore {
   raw_spinlock_t lock;
   unsigned int count;
   struct list_head wait_list;
}s
```

- lock:自旋锁，防止多处理器并行造成错误。
- count：计数器，大于0，表示可用资源数，小于0，其绝对值表示等待的进程数
- wait_list：等待资源的进程队列。
  - 在include下的semaphore.h中

信号量的相关操作

```
void down(struct semaphore *sem)
{
    unsigned long flags;
    raw_spin_lock_irqsave(&sem->lock, flags);
      /*加锁，使信号量的操作在关闭中断的状态下进行，防止多处 理器并发操作造成错误*/
    if (likely(sem->count > 0))
      /*若信号量可用，则将引用计数减1 */
      sem->count--;
    else
      /*如果无信号量可用，则调用__down()函数进入睡眠 等待状态 */
      __down(sem);
    raw_spin_unlock_irqrestore(&sem->lock, flags);
            /* 对信号量的操作解锁 */
}
```

在kernel\locking\semaphore.c

- down()操作，也就是P(proberen)操作，P操作是对资源的申请
- down()操作中调用_down()函数，
- 而__down调用__down_common()，后者是各种down()操作的统一函数。

信号量的相关操作

```
void up(struct semaphore *sem)
{
  unsigned long flags;
  raw_spin_lock_irqsave(&sem->lock, flags);
    /* 对信号量操作进行加锁 */
  if (likely(list_empty(&sem->wait_list)))
    /* 如果该信号量的等待队列为空，则释放信号量 */
     sem->count++;
  else
    /* 否则唤醒该信号量的等待队列队头的进程 */
    __up(sem);
   raw_spin_unlock_irqrestore(&sem->lock, flags);
     /* 对信号 操作进行解锁 */
}
```

在kernel\locking\semaphore.c

- 释放信号量的up()操作，相当于V（verhogen）操作，V操作是释放资源的操作。

信号量的操作函数列表

函数	描述
down(struct semaphore *);	获取信号量，如果不可获取，则进入不可中断睡眠状态（目前已经不建议使用）。
down_interruptible(struct semaphore *);	获取信号量，如果不可获取，则进入可中断睡眠状态
down_killable(struct semaphore *);	获取信号量，如果不可获取，则进入可被致命信号中断的睡眠状态。
down_trylock(struct semaphore *);	尝试获取信号量，如果不能获取，则立刻返回
down_timeout(struct semaphore *, long jiffies);	在给定时间（jiffies）内获取信号量，如果不能够获取，则返回。
up(struct semaphore *);	释放信号量。

信号量与自旋锁的比较

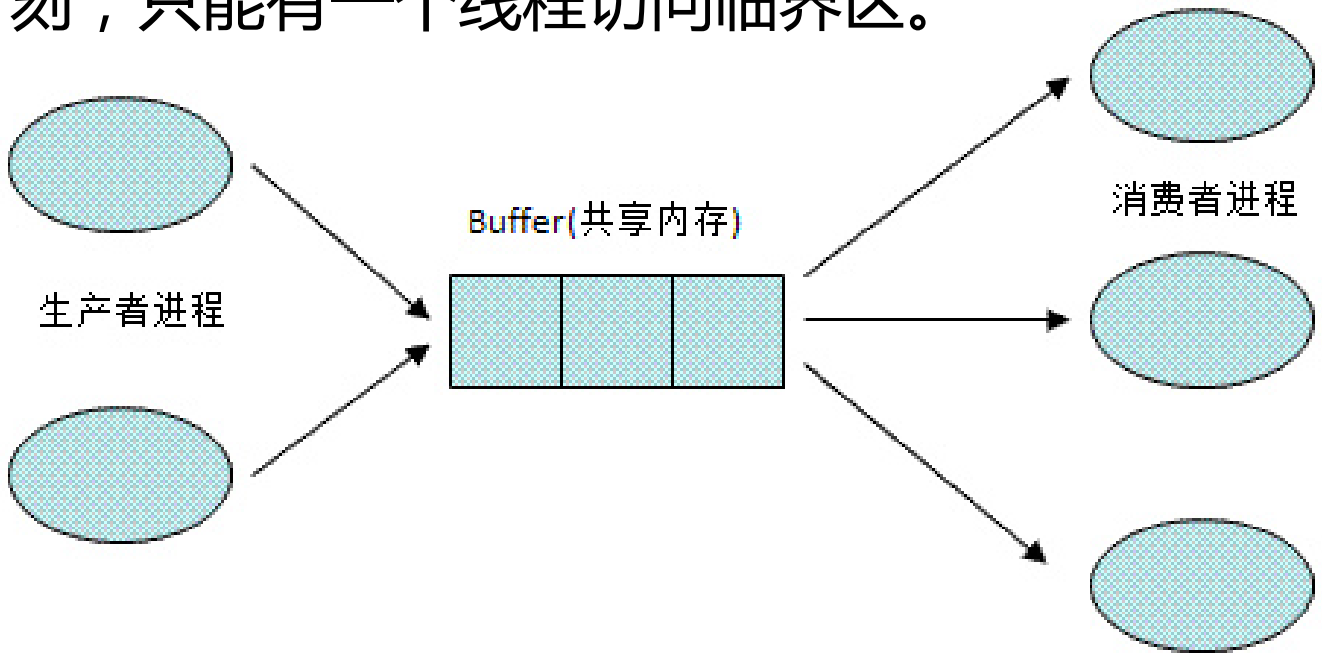
需求	建议的加锁方法
低开销加锁	优先使用自旋锁
短期锁定	优先使用自旋锁
长期加锁	优先使用信号量
中断上下文中加锁	使用自旋锁
持有锁时需要睡眠、调度	使用信号量

内核其它的同步措施

同步措施	含义
互斥锁	互斥锁和信号量为1的信号量含义类似，可以允许睡眠。
完成变量	完成量是基于等到队列机制的，如果在内核中一个任务需要发出信号通知另一个任务发生了某个特定的事件，可以用完成变量。
RCU机制	读-拷贝修改锁机制，允许多个读者同时访问被保护的数据，又允许多个读者和多个写者同时访问被保护的数据。

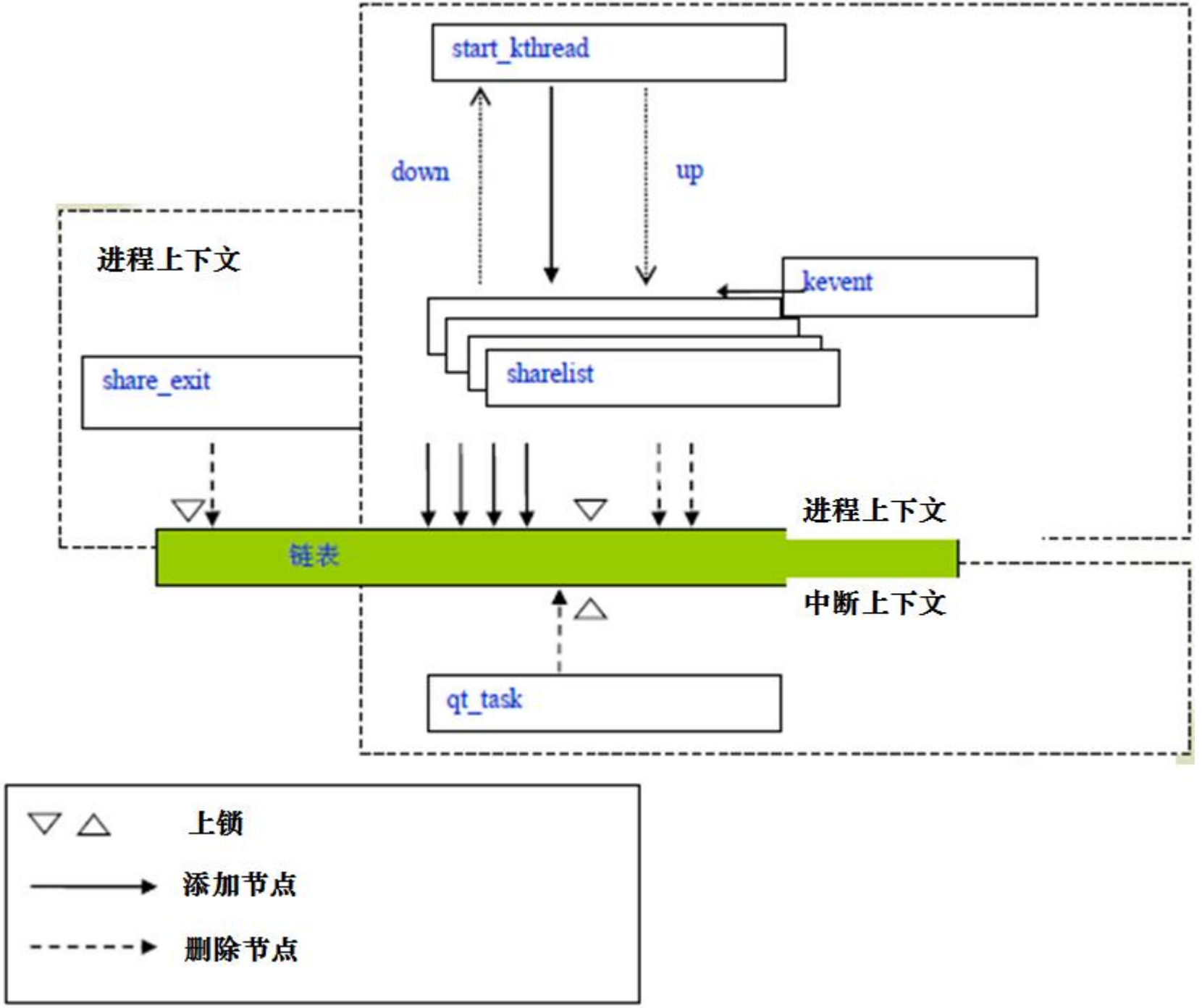
生产者-消费者 并发实例

- 问题描述：一个生产厂家、一个经销商、一个仓库。厂家生产一批产品并放在仓库里，再通知经销商来批发。经销商卖完产品再向厂家下订单，生产厂家才生产下一批产品。
- 问题分析：这是典型的 **生产者-消费者问题**，生产者和消费者用内核线程来模拟，“公共缓冲区”为临界区。
- 同一时刻，只能有一个线程访问临界区。



内核多任务并发控制实例

- 假设存在这样一个的内核共享资源 - 链表。
- 另外构造一个内核多任务访问链表的场景：内核线程向链表加入新节点；内核定时器定时删除结点；系统调用销毁链表。
- 上面三种内核任务并发执行时，有可能会破坏链表数据的完整性，所以我们必须对链表进行同步访问保护，以确保数据一致性。
- 右图是并发控制实例示意



课程思政

课程思政

培养其“实践是检验真理的唯一标准”的求真精神

提及华为鸿蒙系统（HarmonyOS），很多人的第一印象便是，鸿蒙系统是一款近两年才突然走进大众视野的操作系统。其实并非如此，鸿蒙系统的起源可以一直追溯到2012年。

2012年7月2日，华为总裁任正非同诺亚方舟实验室的干部与专家举行座谈。座谈会中，在回答时任华为欧拉实验室终端操作系统开发部部长李金喜的提问时，任正非说道：“如果说这三个操作系统（Android、iOS、Windows Phone 8）都给华为一个平等权利，那我们的操作系统是不需要的，为什么不可以用别人的优势呢？我们现在做终端操作系统是出于战略的考虑，如果他们突然断了我们的‘粮食’——Android系统不给我们用了，Windows Phone 8系统也不给我们用了，我们是不是就‘傻’了？”

培养其“实践是检验真理的唯一标准”的求真精神

于是，2012年，华为在Linux系统的诞生地（芬兰赫尔辛基）组建了自己的终端操作系统研发团队，此后该团队开始从20名工程师逐步壮大。

有人说华为的与众不同之处在于，它在不确定是否能用得上的情况下，仍会坚持为自己打造“备胎”。果不其然，这句话中“不确定的情况”在2019年5月变成了确定——谷歌公司停止了与华为的部分合作，即华为手机将不能使用完整版Android系统。在此背景下，曾经的“备胎”（鸿蒙系统）肩负着前所未有的压力，被推到了历史潮流的浪头。至此，备受国人期待的鸿蒙系统才算正式走进了大众的视野。

我们要培养其“实践是检验真理的唯一标准”的求真精神，坚信在相关科技人才的不懈努力下，华为一定可以尽快构建起鸿蒙系统的应用生态，进而走出一条艰辛但是充满希望的国产操作系统发展之路。

课后练习题

1、请简述并发执行与并行执行的概念，并分析其区别。
2、请对并发执行的原因做一个介绍。
3、请简述哪些条件同时发生时，会导致发生竞争？
4、什么是临界区，临界区有什么特点？
5、什么是死锁，典型的死锁有哪些？
6、为了避免并发，防止竞争，提供对共享数据的保护，内核提供了哪些同步方法？
7、请简述自旋锁spinlock机制？
8、信号量与自旋锁有什么区别，请简述。
9、信号量典型例子是生产者/消费者问题，正确吗？

谢 谢！

THANKS