# 背景

在使用共享内存的应用程序中，必须特别注意保护共享资源，防止共享资源并发访问。共享资源之所以要防止并发访问，是因为如果多个执行线程同时访问和操作数据，就有可能发生各个线程之间相互覆盖共享数据的情况，造成被访问数据处于不一致状态。

现在Linux内核已经发展成抢占式，且支持多处理器。这意味着内核代码可以同时运行在多个处理器上，如果不加以保护，运行在多个处理器上的内核代码完全可能在同一时刻并发访问共享资源。

## 顺序/并发执行

在单进程的时候代码在内存中是顺序执行的。顺序执行就是程序执行是确定的，也就是说指令按照程序顺序执行。

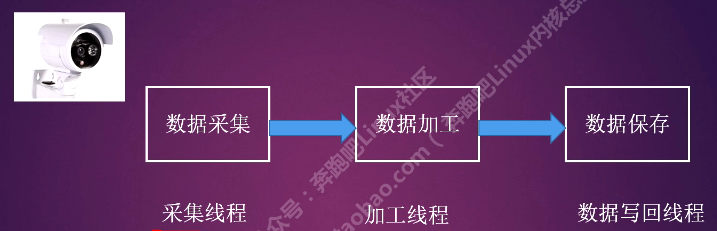
比如单CPU的单进程：



上述数据采集，数据加工，数据保存是串行执行的，需要等待。

在并发执行时，不是顺序执行的，具有不确定性，需要引入多线程的保护机制，比如加锁。

可以采用多个线程执行不同操作：



## 临界区和竞争条件

临界区（也称临界段）就是访问和操作共享数据的代码段。为了避免在临界区中并发访问，开发者必须保证这些代码原子地执行（即执行要么成功要么失败，不允许在结束前被打断），这就是说整个临界区是一个不可分割的指令。

如果两个执行线程可能处于同一个临界区中同时执行，我们就称它是竞争条件（race conditions）。避免并发和防止竞争条件称为同步（synchronization）。

## 加锁

锁有多种多样的形式，并且加锁的粒度范围也各不相同——Linux自身实现了几种不同的锁机制。各种锁机制之间的区别主要在于：当锁已经被其他线程持有，因而不可用时的行为表现——一些锁被争用时会简单地执行忙等待，而另外一些锁会使当前任务睡眠直到锁可用为止。

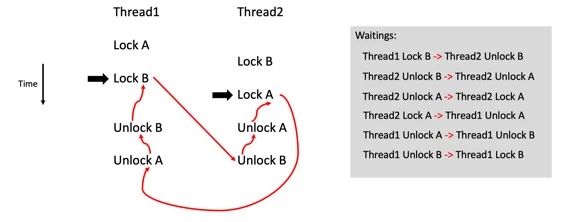
但是，锁根本解决不了什么问题，它只不过是把临界区缩小到加锁和解锁之间的代码，但是仍然有潜在的竞争！所幸的是，锁是采用原子操作实现的，而原子不存在竞争。

## 死锁

### 概述

死锁是多线程和分布式程序中常见的一种严重问题。死锁是毁灭性的，一旦发生，系统很难或者几乎不可能恢复；死锁是随机的，只有满足特定条件才会发生，而如果条件复杂，虽然发生概率很低，但是一旦发生就非常难重现和调试。使用锁而产生的死锁是死锁中的一种常见情况。Linux 内核使用 Lockdep 工具来检测和特别是预测锁的死锁场景。然而，目前 Lockdep 只支持处理互斥锁，不支持更为复杂的读写锁，尤其是递归读锁（Recursive-read lock）。因此，Lockdep 既会出现由读写锁引起的假阳性预测错误，也会出现假阴性预测错误。

在死锁中，因为用锁（Lock）不当而导致的死锁是一个重要死锁来源。锁是同步的一种主要手段，用锁是不可避免的。对于复杂的同步关系，锁的使用会比较复杂。如果使用不当很容易造成锁的死锁。从等待的角度来说，锁的死锁是由于参与线程等待锁的释放，而这种等待构成了等待循环，如 ABBA 死锁：



### 产生条件

死锁产生的必要条件：

1. 互斥条件：进程对资源进行排他性使用，即在一段时间内某资源仅为一个进程所占用。
2. 请求和保持条件：当进程因请求资源而阻塞时，对已获得的资源保持不放。
3. 不可剥夺条件：进程已获得的资源在未使用完之前，不能被剥夺，只能在使用完时由自己释放。
4. 环路等待条件：各个进程组成封闭的环形链，每个进程都等待下一个进程所占用的资源。

### 解决方案

可以将对死锁的解决方案粗略地分为：死锁发现（Detection）、死锁避免（Prevention）和死锁预测（Prediction）。

死锁发现是指在在程序运行中发现死锁实例；死锁避免则是在发现死锁实例即将生成时进一步防止这个实例；而死锁预测则是通过静态或者动态方法找出程序中的潜在死锁，从而从根本上预先消除死锁隐患。

#### 死锁发现（Detection）

#### 死锁避免（Prevention）

防止死锁的方法：

1. 资源一次性分配：破坏请求和保持条件
2. 可剥夺资源：破坏不可剥夺条件
3. 资源有序分配法：破坏循环等待条件

预防死锁的几种策略，会严重损害系统性能。因此在避免死锁时，要施加较弱的限制，从而获得较满意的系统性能。

由于在避免死锁的策略中，允许进程动态地申请资源。因而，系统在进行资源分配之前预先计算资源分配的安全性。若此次分配不会导致系统进入不安全状态，则将资源分配给进程；否则，进程等待。其中最具有代表性的避免死锁的算法是银行家算法。

#### 死锁预测（Prediction）

## 锁争用

锁的争用（lock contention），或简称争用，是指当锁正在被占用时，有其他线程试图获得该锁。由于锁的作用是使程序以串行方式对资源进行访问，所以使用锁无疑会降低系统的性能。

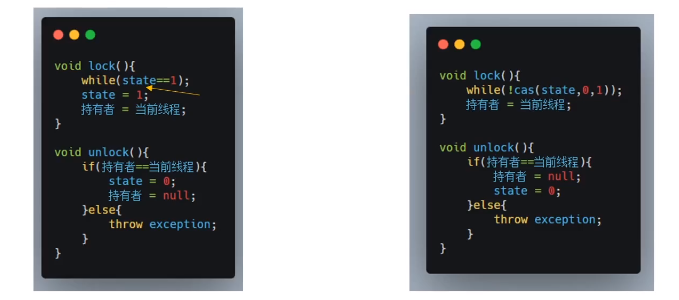
被高度争用（频繁被持有或者长时间持有）的锁会成为系统的瓶颈，严重降低系统性能。即使这样，相比于被几个相互抢夺共享资源的线程撕成碎片，搞得内核崩溃，还是这种同步保护来得更好一点。

## 生产者-消费者模型

# 内核同步方法

## 原子操作

内核提供了两组原子操作接口——一组针对整数进行操作，另一组针对单独的位进行操作。



## 自旋锁

### 背景

我们会遇到这种情况：先得从一个数据结构中移出数据，对其进行操作后，再把它加入到另一个数据结构中。整个执行过程必须是原子的，在数据被更新完毕前，不能被其他代码读取这些数据。显然，简单的原子操作对此无能为力，这就需要使用更加复杂的同步方法——锁来提供保护。

假设我们有一个具有两个处理器core1和core2的计算机，现在在这台计算机上运行的程序中有两个线程：T1和T2分别在处理器core1和core2上运行，两个线程之间共享着一个资源。

首先，我们说明互斥锁的工作原理，互斥锁是一种**sleep-waiting（睡眠等待）**的锁。假设线程T1获取互斥锁并且正在core1上运行时，此时线程T2也想要获取互斥锁（pthread\_mutex\_lock），但是由于T1正在使用互斥锁使得T2被阻塞。当T2处于阻塞状态时，T2被放入到等待队列中去，处理器core2会去处理其他任务而不必一直等待（忙等）。也就是说处理器不会因为线程阻塞而空闲着，它去处理其他事务去了。

而自旋锁就不同了，自旋锁是一种**busy-waiting（忙等待）的锁**。也就是说，如果T1正在使用自旋锁，而T2也去申请这个自旋锁，此时T2肯定得不到这个自旋锁。与互斥锁相反的是，此时运行T2的处理器core2会一直不断地循环检查锁是否可用（自旋锁请求），直到获取到这个自旋锁为止。

### 概述

从自旋锁的名字也可以看出来，如果一个线程想要获取一个被使用的自旋锁，那么它会一直占用CPU请求这个自旋锁使得CPU不能去做其他的事情，直到获取这个锁为止，这就是“自旋锁”的含义。

**自旋锁最多只能被一个可执行线程持有**，如果一个执行线程试图获得一个被已经持有（即所谓的争用）的自旋锁，那么该线程就会一直进行忙循环——旋转——等待锁重新可用。要是锁未被争用，请求锁的执行线程便能立即得到它，继续执行。在任意时间，自旋锁都可以防止多于一个的执行线程同时进入临界区。

一个被争用的自旋锁使得请求它的线程在等待锁重新可用时自旋（特别浪费处理器时间），这种行为是自旋锁的要点。所以**自旋锁不应该被长时间持有**。事实上，这点正是使用自旋锁的初衷：在短时间内进行轻量级加锁。还可以采取另外的方式来处理对锁的争用：让请求线程睡眠，直到锁重新可用时再唤醒它。这样处理器就不必循环等待，可以去执行其他代码（信号量思想）。这也会带来一定的开销——这里有两次明显的上下文切换，被阻塞的线程要换出和换入，与实现自旋锁的少数几行代码相比，上下文切换当然有较多的代码。因此，持有自旋锁的时间最好小于完成两次上下文切换的耗时。当然，我们大多数人都不会无聊到去测量上下文切换的耗时，所以我们让持有自旋锁的时间应尽可能短就可以了。

**警告：自旋锁是不可递归的！**

Linux内核实现的自旋锁时不可递归的，这点不同于自旋锁在其他操作系统中的实现。所以如果你试图得到一个你正持有的锁，你必须自旋，等待自己释放这个锁。但你处于自旋忙等待中，所以你永远没有机会释放锁，于是被自己锁死了。

自旋锁可以使用在中断处理程序中（此处不能使用信号量，因为它们会导致睡眠）。在中断处理程序中使用自旋锁时，一定要在获取锁之前，首先禁止本地中断（在当前处理器上的中断请求），否则，中断处理程序就会打断正持有锁的内核代码，有可能会试图去争用这个已经被持有的自旋锁。这样一来，中断处理程序就会自旋，等待该锁重新可用，但是锁的持有者在这个中断处理程序执行完毕前不可能运行。

**自旋锁与互斥锁：**

**当发生阻塞时，互斥锁可以让CPU去处理其他的任务，而自旋锁让CPU一直不断循环请求获取这个锁**。通过两个含义的对比可以让我们知道**“自旋锁”是比较耗费CPU的**。

**基本概念：**

**忙等待的锁机制。**操作系统中锁的机制分为两类，一类是忙等待，另一类是睡眠等待。Spinlock属于前者，当无法获取spinlock锁时会不断尝试，直到获取锁为止。

同一时刻只能有一个内核代码路径可以获得该锁。

要求spinlock锁持有者尽快完成临界区的执行任务。如果临界区执行时间过长，在锁外面忙等待的CPU比较浪费，特别是spinloc临界区里不能睡眠。

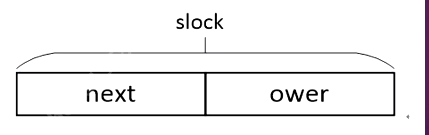
Spinlock锁可以在中断上下文中使用。

### 内核实现

### ticket-based spinlock

Spinlock的问题：在多核处理器中，spinlock锁的争用很激烈（导致不公平，系统性能下降很快）。

当该锁释放时，事实上有可能刚刚释放该锁的CPU马上又获得了该锁的使用权，或者说在同一个NUMA节点上的CPU都有可能抢先获取了该锁，而没有考虑那些已经在锁外面等待了很久的CPU。



### 函数



#### spin\_lock\_init

#### spin\_lock

#### spin\_try\_lock

#### spin\_unlock

#### spin\_is\_locked

## 读-写自旋锁

### 概述

当对某个数据结构的操作可以像这样被划分为读/写或者消费者/生产者两种类别时，类似读/写锁这样的机制就很有帮助了。为此，Linux内核提供了专门的读-写自旋锁。这种自旋锁为读和写分别提供了不同的锁。一个或多个读任务可以并发地持有读者锁；相反，用于写的锁最多只能被一个写任务持有，而且此时不能有并发的读操作。有时把读/写锁叫做共享/排斥锁，或者并发/排斥锁，因为这种锁以共享（对于读者而言）和排斥（对于写者而言）的形式获得使用。

多个读者可以安全地获得同一个读锁，事实上，即使一个线程递归地获得同一读锁也是安全的。这个特性使得读-写自旋锁真正成为一种有用并且常用的优化手段。

### 函数



## 信号量

### 概述

Linux中的信号量是一种睡眠锁。如果有一个任务试图获得一个不可用（已经被占用）的信号量时，信号量会将其推进一个等待队列，然后让其睡眠。这时处理器能够重获自由，从而去执行其他代码。当持有的信号量可用（被释放）后，处于等待队列中的那个任务将被唤醒，并获得该信号量。

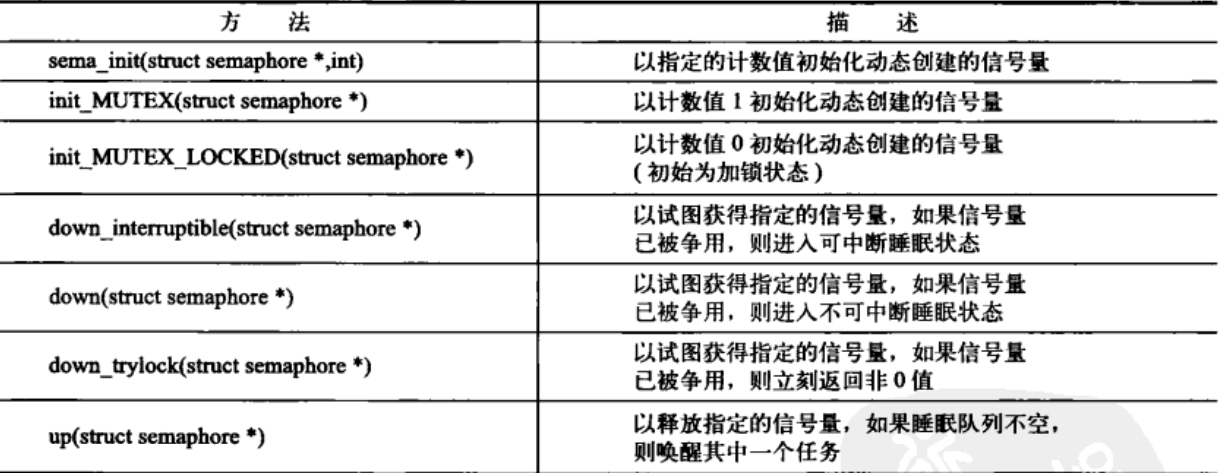
说明：

1. 由于争用信号量的进程在等待锁重新变为可用时会睡眠，所以信号量适用于锁会被长时间持有的情况；
2. 相反，锁被短时间持有时，使用信号量就不太适宜了。因为睡眠、维护等待队列以及唤醒所花费的开销可能比锁占用的全部时间还要长；
3. 由于执行线程在锁被争用时会睡眠，所以只能在进程上下文中才能获取信号量锁，因为在中断上下文中是不能进行调度的；
4. 你可以在持有信号量时去睡眠（当然你也可能并不需要睡眠），因为当其他进程试图获得同一信号量时不会因此而死锁（因为该进程也只是去睡眠而已，而你最终会继续执行的）；
5. 在你占用信号量的同时不能占用自旋锁，因为在你等待信号量时可能会睡眠，而在持有自旋锁时是不允许睡眠的。

### 计数信号量和二值信号量

它可以同时允许任意数量的锁持有者，而自旋锁在一个时刻最多允许一个任务持有它。信号量同时允许的持有者数量可以在声明信号量时指定。这个值称为使用者数量（usage count）或者简单地叫数量（count）。通常情况下，信号量和自旋锁一样，在一个时刻仅允许一个锁持有者。这时计数等于1，这样的信号量被称为**二值信号量**（因为它或者由一个任务持有，或者根本没有任务持有它）或者称为互斥信号量（因为它强制进行互斥）。另一方面，初始化时也可以把数量设置为大于1的非0值。这种情况，信号量被称为**计数信号量（counting semaphone）**，它允许在一个时刻至多有count个锁持有者。计数信号量不能用来进行强制互斥，因为它允许多个执行线程同时访问临界区。相反，这种信号量用来对特定代码加以限制，内核中使用它的机会不多。在使用信号量时，基本上用到的都是互斥信号量（计数等于1的信号量）。

### 函数



## 读-写信号量

与自旋锁一样，信号量也有区分读-写访问的可能。与读-写自旋锁和普通自旋锁之间的关系差不多，读-写信号量也要比普通信号量更具有优势。

## 互斥体

### 背景

信号量适用于那些较为复杂的、未明情况下的互斥访问，比如内核于用户空间复杂的交互行为。这也意味着简单的锁定而使用信号量并不方便，并且信号量也缺乏强制的规则来行使任何形式的自动调试，即使受限的调试也不可能。为了找到一个更简单睡眠锁，内核开发者引入了互斥体（mutex）。

注：互斥体这个称谓所指的是任何可以睡眠的强制互斥锁。

### 概述

mutex的简洁性和高效性源自于相比使用信号量更多的受限性，因此mutex的使用场景相对而言更加严格、更定向。

1. 任何时刻中只有一个任务可以持有mutex，也就是说，mutex的使用计数永远是1；
2. 给mutex上锁者必须负责给其解锁——不能在一个上下文中锁定一个mutex，而在另一个上下文中解锁，这个限制使得**mutex不适合内核同用户空间复杂的同步场景**，最常使用的方式是：在同一上下文中上锁和解锁；
3. 递归地上锁和解锁是不允许的。也就是说，你不能递归地持有同一个锁，同样你也不能再去解锁一个已经被解开的mutex；
4. 当持有一个mutex时，进程不可以退出；
5. mutex不能再中断或下半部中使用，即使使用mutex\_trylock()也不行；
6. mutex只能通过官方API管理：不可被拷贝、手动初始化或者重复初始化。

mutex结构最有用的特色是：通过一个特殊的调试模式，内核可以采用编程方式检查和警告任何践踏其约束法则的行为。当打开内核配置选项CONFIG\_DEBUG\_MUTEXS后，就会有很多检查来确保这些（还有别的一些）约束得以遵守。

#### 自旋锁和互斥体

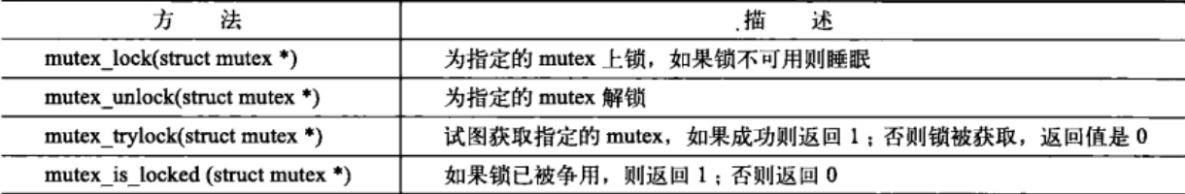
在中断上下文中只能使用自旋锁，而在任务睡眠时只能使用互斥锁。



#### 信号量和互斥体

互斥体和信号量很相似，所幸，它们的标准使用方式都有简单的规范：除非mutex的某个约束妨碍你使用，否则相比信号量要优先使用mutex。当你写新代码时，只要碰到特殊场合（一般是底层代码）才会需要使用信号量。因此，建议首选mutex。如果发现不能满足其约束条件，且没有其他别的选择时，再考虑选择信号量。

### 函数



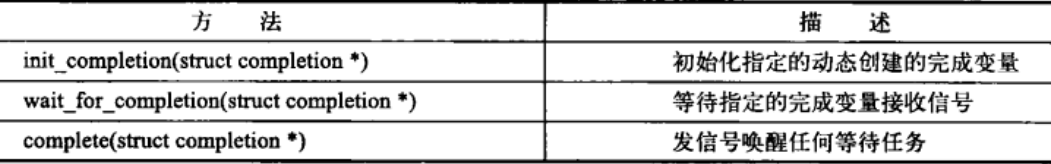
## 完成变量/Completions机制

### 概述

如果在内核中一个任务需要发出信号通知另一任务发生了某个特定事件，利用完成变量（completion variable）是使两个任务得以同步的简单方法。如果一个任务要执行一些工作时，另一个任务就会在完成变量上等待。当这个任务完成工作后，会使用完成变量去唤醒在等待的任务。

听起来很像信号量，的确思想是一样的。事实上，完成变量仅仅提供了代替信号量的一个简单的解决方法。例如，当子进程执行或者退出时，vfork系统调用使用完成变量唤醒父进程。

### 函数



## BKL：大内核锁

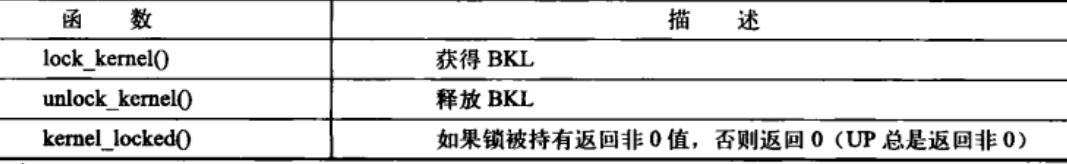
### 概述

BKL（大内核锁）是一个全局自旋锁，使用它主要是为了方便实现从Linux最初的SMP过渡到细粒度加锁机制。

特性：

1. 持有BKL的任务仍然可以睡眠。因为当任务无法被调度时，所加锁会自动被丢弃；当任务被调度时，锁又会被重新获得。当然，这并不是说，当任务持有BKL时，睡眠是安全的，仅仅是可以这样做，因为睡眠不会造成任务死锁。
2. BKL是一种递归锁。一个进程可以多次请求一个锁，并不会像自旋锁那样产生死锁现象。
3. BKL只可以用在进程上下文中，和自旋锁不同，你不能在中断上下文中申请BKL。
4. 新的用户不允许使用BKL。随着内核版本的不断前进，越来越少的驱动和子系统再依赖于BKL了。

### 函数



## 顺序锁/seq锁

顺序锁，通常简称seq锁，是在2.6版本内核中引入的一种新型锁。这种锁提供了一种简单的机制，用于读写共享数据。实现这种锁主要依靠一个序列计数器。当有疑义的数据被写入时，会得到一个锁，并且序列值会增加。在读取数据之前和之后，序列号都被读取。如果读取的序列号值相同，说明在读操作进行的过程中没有被写操作打断过。此外，如果读取的值是偶数，那么就表明写操作没有发生（要明白因为锁的初始值是0，所以写操作会使得值变为奇数，释放的时候变为偶数）。

注：这与MySQL中的GTID思想有点类似。

Seq锁在你遇到如下需求时将是最理想的选择：

1. 你的数据存在很多读者；
2. 你的数据写者很少；
3. 虽然写者很少，但是你希望写优先于读，而且不允许读者让写者饥饿；
4. 你的数据很简单，如简单结构，甚至是简单的整型——在某些场合，你是不能使用原子量的。

## 禁止抢占

## 顺序和屏障

# 比较

互斥锁：对于时间比较长，情况比较复杂的加锁。

自旋锁：对于比较简单的多行代码。

原子操作：单一。

说明：epoll是线程安全的（使用mutex互斥锁给红黑树加锁，还有一种二段锁/锁子树，即spinlock锁定一个节点）。