# 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。

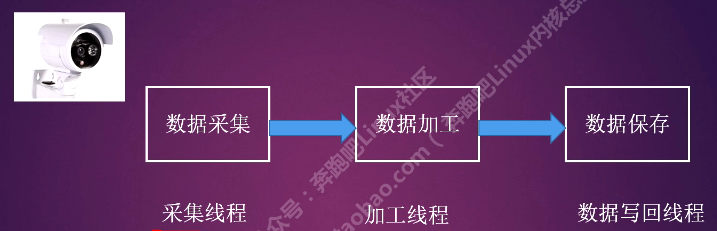
比如单CPU的单进程：



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。

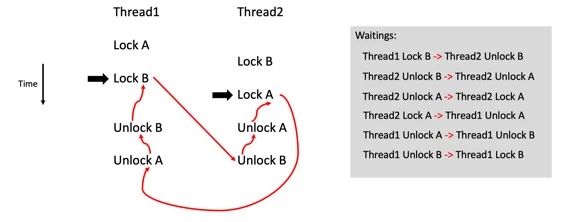
但是，锁根本解决不了什么问题，它只不过是把临界区缩小到加锁和解锁之间的代码，但是仍然有潜在的竞争！所幸的是，锁是采用原子操作实现的，而原子不存在竞争。

## 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。

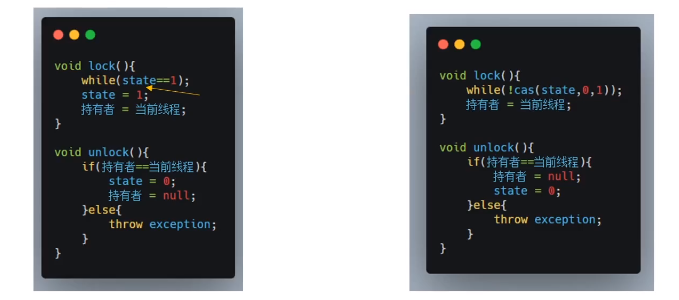
被高度争用（频繁被持有或者长时间持有）的锁会成为系统的瓶颈，严重降低系统性能。即使这样，相比于被几个相互抢夺共享资源的线程撕成碎片，搞得内核崩溃，还是这种同步保护来得更好一点。

## 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倍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。

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。

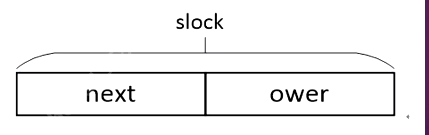
Spinlock锁可以在中断上下文中使用。

### 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\_trylock

#### spin\_unlock

#### spin\_is\_locked

## 读-写自旋锁

## 信号量

## 读-写信号量

## 互斥体

## 完成变量

## BLK：大内核锁

## 顺序锁

## 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）。