# 延时锁

# 自旋锁

## 概述

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。

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。

## 函数

### spin\_lock\_init

### spin\_lock

### spin\_trylock

### spin\_unlock

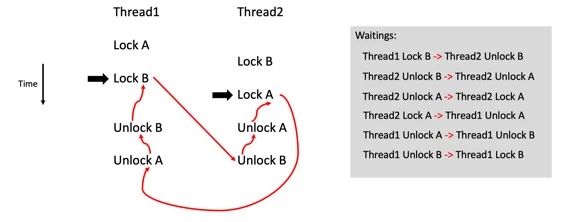
### spin\_is\_locked

# 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

## 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）

### 死锁预测（Prediction）