# 并发面临的问题

## 冲突

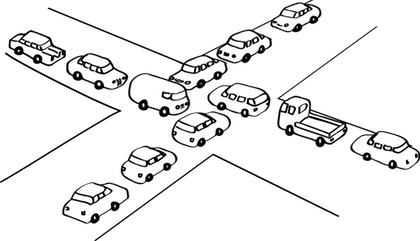
这是并发面临的最基本的问题,如何保持数据的一致性.

一组数据,它们共同构成了一个状态, 一次状态的变化由其中特定的几个变量变化构成.

如果状态变化不完整,就意味着存在冲突.

## 互斥

## 死锁



## 饥饿

# 几种多线程泛型

## UI更新型

在Google的Android应用中,所有对UI的访问/更新操作都必须在主线程中完成.可以证明这种模型是通用的将多个线程中的同一个临界区域序列化的方法.

* 模型

定义一个Update线程,其结构包含一个线程安全的Queue,

还包括: 线程初始化,异常处理,结束调用的回调函数.

定义线程循环:

不断从Queue中取出一个Callable对象 (阻塞式)

调用Callable对象,其中Callable.call用于调用,Callable.event用于通知等待的线程操作已经完成

如果捕获到一般异常,调用异常处理

定义线程结束:

当其中的Callable对象产生StopWorking异常的时候,线程退出,调用线程结束函数

其他线程: 当某一个线程需要使用使用\更新Update线程的数据时, 使用Queue的put操作

如果这个操作是异步的,put普通的callable对象

否则put一个带有event的对象,然后下一步等待event的结束.

缺点:无法定义优先级,当某个线程需要结束某些操作时,由于那些操作已经被放在了队列的前面部分,所以那些操作总是会被先执行

优点:确定性(所有的更新操作一定是串行的), 并且没有竞争条件

* 例子

## 临界区域同步型

无论什么线程,如果确定要以P操作一个在多个线程中涉及的共享变量V,将此定义为P(V)

要求P(V)是原子操作.(注意,V可能是多个变量的集合,即一个由多个变量确定的状态)

* 例子

以典型的生产者消费者为例,两者共用一个queue{buffer, pointer}, 在使用的过程中必须buffer,pointer的一致性.

生产者和消费者都需要操作buffer,pointer

* 大数组冗余

如果有一个很大的数组arr, 大小为n(n>=65535),

支持更新操作.(更新:要么+1,要么不变)

更新的时候, arr中的前半部分是线程安全的, 后半部分是非线程安全的.

测试部分: 开启n(n>=100,n>=20)个线程同时对arr进行更新操作

最终统计arr中前半部分的元素,后半部分的元素. 后半部分如果不一致,就体现了非线程安全性.

# 附录:几种模拟并发问题的典型方法

并发产生冲突的原因在于指令乱序执行.但是无论如何,从时间上来看,所有的指令最终都将排列成一个序列(即使有多核CPU,在遇到同时需要更新某个地址时,总是先锁总线,cpu的微指令是线程安全的)

假设有操作P0,…,Pn 它们都是原子操作

有复合操作CP0,CP1,…CPm

CP0 -> Pi00 Pi01 … Pik0 len=k0-1

CP1->…

CPm -> …

现在并发执行 CP0,CP1,…CPm,

为了计算所有的指令序列排列数, 显然可以先计算出CP1~CPm的结果 CP[1,m]

然后与CP[0]进行组合

组合的计算方式, 如果M是较大的,N是较小的, 等价于将M划分成N+1个区域的方法,

就是C(M,N)

假设每一种组合都是等可能的,其中产生冲突的为M种,其余N种无冲突.

* 使冲突容易检测

对于consumer 和 producer模型, queue{buffer, pointer}可以变成:

queue{buffer,pointer, unassured\_pointer}

其中保证pointer和buffer同步(它们的更新是原子操作),而unassured\_pointer与buffer的更新不在同一个原子内部.

无论何时, 调用下面的checkCorrupted方法:

if unassured\_pointer!=pointer:

return True

else:

return False

不一定能检测到冲突(即冲突可能被修正了,但是冲突毕竟是发生过的),但是如果返回False一定发生了冲突.