并行基础知识

李明明

（以下内容基本都是我根据自己的理解表述的，所以描述会比较白话，比较不够专业，也不够全面，旨在介绍基本概念以及这些概念对我们编程的影响。）

## 32位&64位

32位、64位操作系统，是指CPU一次读取内存的位数，比如一个long long类型的整数是64位，32位架构cpu需要读两次，每次读32位，而64位架构cpu只需要读一次，一次读64位数。

## 进程&线程

进程是资源的概念，线程是工作&调度的概念。

代码段，全局变量，虚拟地址，内存等都是资源，都归属于进程，每个进程相互隔离，进程内多个线程可以共同访问。

代码运行，比如i=i+1，代码只有一份，每个线程都可以运行这一份代码，这一份代码放在磁盘里或者内存里并不会对i产生任何影响，只有线程运行了这一份代码，i才会真正改变。

## cpu核

还是以i=i+1举例，代码放在磁盘里或者内存里并不会对i产生任何影响，只有CPU真正执行这段代码，即cpu把i的值从内存读到cpu寄存器，然后改变i的值（加一），最后再把加一后i的值写回原内存位置，i才真正完成了改变（加一了）。

i=i+1是要完成一个任务，而实际做这个任务的是cpu，即cpu才是真正做事的。

再结合线程的介绍，代码实际运行在线程里，运行线程的（真正做事的）是cpu。

再扩展一点，能独立运行线程的是cpu核，一台服务器一般会有多个核，即可以同时运行多个线程。

服务器cpu信息：

[root@host180 test]# lscpu

架构： x86\_64

CPU 运行模式： 32-bit, 64-bit

Address sizes: 46 bits physical, 57 bits virtual

字节序： Little Endian

CPU: 128

在线 CPU 列表： 0-127

厂商 ID： GenuineIntel

BIOS Vendor ID: Intel(R) Corporation

型号名称： Intel(R) Xeon(R) Gold 6338 CPU @ 2.00GHz

BIOS Model name: Intel(R) Xeon(R) Gold 6338 CPU @ 2.00GHz

CPU 系列： 6

型号： 106

每个核的线程数： 2

每个座的核数： 32

座： 2

步进： 6

CPU 最大 MHz： 3200.0000

CPU 最小 MHz： 800.0000

Caches (sum of all):

L1d: 3 MiB (64 instances)

L1i: 2 MiB (64 instances)

L2: 80 MiB (64 instances)

L3: 96 MiB (2 instances)

NUMA:

NUMA 节点： 2

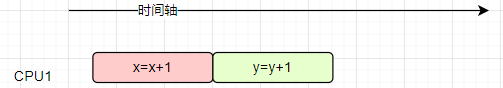
NUMA 节点0 CPU： 0-31,64-95

NUMA 节点1 CPU： 32-63,96-127

## 并发&并行

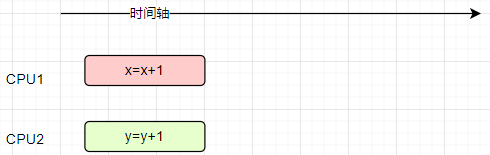
### 串行任务：

一个cpu两个线程，两个任务，任务一：x= x +1;任务二：y = y + 1;(假设任务都是原子的，一个线程执行一个任务，长条不同颜色代表不同线程)



### 并行任务：

两个cpu两个线程，两个任务，任务一：x= x +1;任务二：y = y + 1; (假设任务都是原子的，一个线程执行一个任务，长条不同颜色代表不同线程)

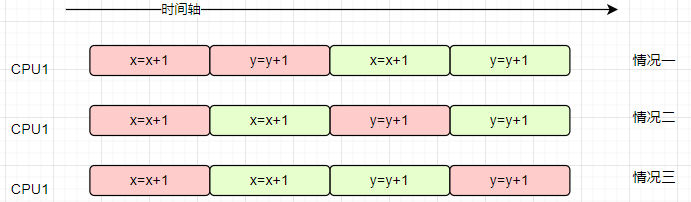


### 调度

前面已经说了，代码运行在线程上，真正运行线程的是cpu核，那么cpu核应该运行哪个线程，就是调度。（应该运行哪个线程是调度算法，就不详细介绍了。）

一个cpu两个线程，两个任务，任务一：x= x +1;任务二：y = y + 1; (假设任务都是原子的，每个线程执行两个任务，长条不同颜色代表不同线程)

假如cpu先调度了线程一，线程一再处理任务二之前可能cpu会调度线程二，暂停线程一代码的运行，转而运行线程二的代码。同样，运行线程二中间过程也可能发生新的调度。



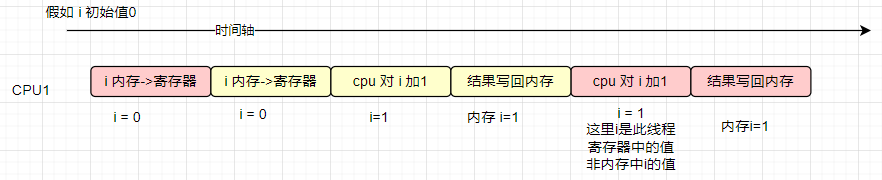
另外，这种情况属于并发。可以和并行对比，并行是真实的同时在处理两个任务，并发是把一个任务切分为不同的小任务，两个任务（线程）交叉的完成各个小任务。

### 原子操作

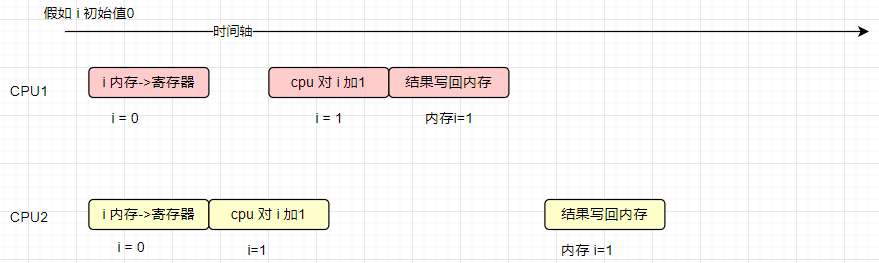
原子就是不可分割，一次完成。

再看上面【cpu核】章节的开始（还是以i=i+1举例，代码放在磁盘里或者内存里并不会对i产生任何影响，只有CPU真正执行这段代码，即cpu把i的值从内存读到cpu寄存器，然后改变i的值（加一），最后再把加一后i的值写回原内存位置，i才真正完成了改变（加一了）），i=i+1这一行代码cpu实际做了三个动作。在这三个动作的间隙，cpu可能会切换调度其它线程，也可能其它cpu会调度其它线程执行相同的代码改变i的值，这样i就不准确了。所以i=i+1，即i++或者++i是非原子的（有三部分组成）。

假如一个cpu，两个线程，都执行i++，i初始值0，我们期望 i 最后是2，但实际最后i的值可能是1。



假如两个cpu，两个线程，都执行i++，i初始值0，我们期望 i 最后是2，但实际最后i的值可能是1。



另外，通过这个例子，再补充进程与线程的区别。前面说了，cpu核调度的是线程，代码运行在线程上，那么cpu该如何执行函数，比如A函数调用B函数，B函数执行完如何回到A函数，cpu由线程一切换到线程二，如何知道线程二从哪个位置开始执行，这些属于栈信息，所以栈是归属于线程的，每个线程有独立的栈（栈上还保存有临时变量）。

从上面的例子还能看到，cpu 把 i 从内存读到寄存器，cpu切换线程时，还会保存寄存器信息到切出线程栈里。但是寄存器不归属线程，也不归属进程，寄存器是cpu的。

### 原子操作的实现

简单的实现方法就是锁内存总线，更高效的方案是只保护要读取的内存。

## 线程与协程

基于自己的理解和产品对协程的使用粗浅的介绍一下，旨在方便对协程的工作模式有所掌握，对编码有所指导。

xstream：对应线程；

ult：可以理解为一个任务，创建ult时会注册勾子函数，此函数为具体处理此ult的方法。（ult是轻量化线程，但是这样叫容易和线程搞混，先简单理解它就是个任务。）

ult池：每个xstream都有ult池，生产者创建ult投入对应xstream的ult池，然后xstream从池中取ult，调用ult中勾子函数处理此ult任务。

前面已经介绍，线程是cpu调度的最小单元，但是协程认为os的调度策略太低效了，协程给自己开了个后门，自己管理调度。

简单理解，cpu决定调度哪个线程，在协程场景就是调度哪个xstream。相当于xstream获得了当前cpu的使用权，让cpu干啥（运行哪些代码）由xstream自己决定。具体表现就是xstream决定处理哪个ult。

每个ult都注册有自己的勾子函数，xstream处理ult1的中间过程可能切换到处理ult2（需要ult1主动让出xstream使用权），后面再继续处理ult1。结合前面【原子操作】章节最后补充内容（比如A函数调用B函数，B函数执行完如何回到A函数，cpu由线程一切换到线程二，如何知道线程二从哪个位置开始执行，这些属于栈信息，所以栈是归属于线程的，每个线程有独立的栈），xstream重新切回ult1也需要知道从哪个位置开始执行代码，所以ult也有自己独立的栈信息。

归纳总结，线程有独立的栈信息，xstream实际是一个线程，ult也有自己独立的栈，ult非独立线程又有线程的特性，所以叫轻量化线程，轻量化表明ult比线程更简单，更容易调度。

注意：  
1、cpu调度的最小单元是线程，xtream是一个线程，如果ult中有代码是表明线程级别把cpu让出，即此xstream失去了cpu使用权，那么此xstream下所有ult都得不到处理了，因为真正做事的是cpu，没有cpu使用权什么都做不了。（线程级别把cpu让出：比如获取线程mutex锁pthread\_mutex\_lock阻塞。）

2、协程锁接口阻塞或者其它阻塞接口、主动让出执行权接口，都只是中断当前ult的执行，xstream仍占有cpu的使用权，可以继续调度其它ult。

3、归属同一个xstream的ult可以并发，但不会并行。很好理解，因为一个xstream对应一个线程，一个线程只能跑在一个cpu核上，一个cpu核没有并行能力。

再改造下前面的例子进行对比，对比的例子是【原子操作】章节（假如一个cpu，两个线程，都执行i++，i初始值0，我们期望 i 最后是2，但实际最后i的值可能是1）。

改造为：一个xstream，两个ult，都执行i++，i初始值0，我们期望 i 最后是2，实际两个ult执行完i的结果是2；

我们前面说过，i++是非原子的，有三个动作要做。但是这三个动作都不会让出执行权，所以当xstream调度到ult1执行i++时，必然能把i++的三个动作执行完，不存在ult1把i从内存读到寄存器，然后xstream调度ult2又去执行i++的情况。

你可能会疑惑，（假如一个cpu，两个线程，都执行i++，i初始值0，我们期望 i 最后是2，但实际最后i的值可能是1），同样是一个cpu，为啥两个线程就可能有问题，而两个协程就没问题？有这个疑问的自己好好想想吧。（提示一点：线程调度是不受线程本身或者所属进程控制的，是由os的调度算法控制的。）