

线程

一、生活举例

什么是单线程，什么是多线程？为什么要使用多线程？

1、程序设计的目标

从程序的角度来看，一个好的程序的目标应该是性能与用户体验的平衡。当然一个程序是否能够满足用户的需求暂且不谈，这是业务层面的问题，我们仅仅讨论程序本身。围绕两点来展开，性能与用户体验。

性能：在其他同等条件下，高性能的程序应该可以等同于CPU的利用率，CPU的利用率越高（一直在工作，没有闲下来的时候），程序的性能越高。

体验：这里的体验不只是界面多么漂亮，功能多么顺手，这里的体验指程序的响应速度，响应速度越快，用户体验越好。

2、单线程多任务无阻塞

以生活中食堂打饭的场景作为比喻，假设有这样的场景，小A，小B，小C 在窗口依次排队打饭。 假设窗口负责打饭的阿姨打一个菜需要耗时1秒。如果小A需要2个菜，小B需要3个菜，小C需要2个菜。如下：

阿姨(CPU)：打一个菜需要1秒

小A：2个菜

小B：3个菜

小C：2个菜

那么在这种模型下将所有服务做完阿姨需要耗时 $2 + 3 + 2 = 7$ 秒

阿姨 = CPU

小A,小B,小C = 任务(这里是以任务为概念，表示需要做一些事情)

这种模型下CPU是满负荷不间断运转的，没有空闲，用户体验还不错。这种程序中每个任务的耗时都较小，是非常理想的状态，一般情况下基本不太可能存在

适用于：性能要求不高，任务数不多。

3、单线程多任务IO阻塞

将上面的场景稍微做改动：

阿姨：打一个菜需要1秒

小A：2个菜，但是忘记带钱了，要找同学送过来，估计需要等5分钟可以送到（可以理解为磁盘IO）

小B：3个菜

小C：2个菜

这种情况下小A这里发生了阻塞，实际上小A这里耗费了5分钟也就是 300秒+ 2个菜的时间，也就是302秒，而CPU则空闲了300秒，实际上工作2秒。

所有服务做完花费 $302 + 3 + 2 = 307$ 秒 CPU实际工作7秒，等待300秒。极大浪费了CPU的时钟周期。用户体验很差，因为小A阻塞的时候，后面的所有人都等着，而实际上此时CPU空闲。所以单线程中不要有阻塞出现。

4、多线程多任务异步IO

还是上面的模型，加入一个角色：值日生小哥，他负责事先询问每一个人是否带钱了，如果带钱了则允许打菜，否则把钱准备好了再说。

<1> 值日生小哥问小A准备好打菜了吗，小A说忘带钱了，值日生小哥说，你把钱准备好了再说，小A开始准备（需要300秒，从此刻开始计时）。

<2> 值日生小哥问小B准备好打菜了吗，小B说可以了，阿姨服务小B，耗时3秒（与此同时小A还在准备中，并且已经准备了3秒）

<3> 值日生小哥问小C准备好打菜了吗，小C说可以了，阿姨服务小C，耗时2秒（与此同时小A还在准备中，并且已经准备了5秒，前面的3秒+这里的2秒）

<4> 值日生小哥问小A准备好了没有，小A说还要等一会，阿姨由于没有人过来服务，处于空闲状态（小A还在准备中，他还需要准备295秒，但是这个时候B和C已经服务完了）

<5> 从第1步开始计时有300秒之后，小A准备好了，阿姨服务小A，耗时2秒

整个过程做完耗时 $300 + 2 = 302$ 秒 CPU工作7秒，空闲295秒

值日生小哥相当于select模型中的select功能，负责轮询任务是否可以工作，如果可以则直接工作，否则继续轮询。在小A阻塞的300秒里面，阿姨（CPU）没有傻等，而是在服务后面的人，也就是小B和小C，所以这里与模型3不同的是，这里有5秒CPU是工作的。如果打饭的人越多，这种模型CPU的利用率越高，例如如果有小D，小E，小F.....等需要服务，CPU可以在小A阻塞的300秒期间内继续服务其他人。实际上值日生小哥轮询也会耗时，这个耗时是很少的，几乎可以忽略不计，但是如果任务非常多，这个轮询还是会影响性能的，但是epoll模型已经不使用轮询的方式，相当于A，B，C会主动跟值日生小哥报告，说我准备好了，可以直接打菜了。

这种模式下用户体验好，CPU利用率高（任务越多利用率越高）

阶段小结

在上述的例子中，打饭的窗口只有一个，打饭的阿姨也只有一个。当排队的人数很多（性能要求较高）的时候，需要充分的利用CPU的计算资源（不让CPU闲下来）。如何才能更高的利用CPU？

在步骤4中，引入了一个值日生的角色，判断各个任务是否能够执行。如果能，则执行，如果不能，则等待，下一个顶上。

但是，这种方式还是不能解决性能问题。原因是窗口太少，阿姨太少。

怎么办？增加窗口，增加阿姨。

5、单线程多任务，有耗时计算

回到最开始的模型，如下：

阿姨：打一个菜需要1秒

小A：200个菜

小B：3个菜

小C：2个菜

顺序做完所有任务，需要耗时 $200 + 3 + 2 = 205$ 秒，CPU无空闲，但是用户体验却不是很好，因为显然后面的B，C需要等待小A 200秒的时间，这种情况下是没有IO阻塞的，但是任务A本身太耗CPU了，所以说如果单线程中出现了耗时的操作，一定会影响体验（IO操作或者是耗时的计算都属于耗时的操作，都会导致阻塞，但是这两种导致阻塞的性质是不一样的）。在所有的单线程模型中都不允许出现阻塞的情况，如果出现，那么用户体验是极差的。

出现阻塞的情况一般有2种，一种是IO阻塞，例如典型的如磁盘操作，这种情况下的阻塞会导致CPU空闲等待（当然现代操作系统中如果IO阻塞，操作系统一定会将导致IO阻塞的线程挂起）。这种阻塞的情况，可以通过异步IO的方法避免，这样就避免程序中仅有的单线程被操作系统挂起。另一种情况下是确实有非常多的计算操作，例如一个复杂的加密算法，确实需要消耗非常多的CPU时间，这种情况下CPU并不是空闲的，反而是全负荷工作的。这种CPU密集的工作不适合放在单线程中，虽然CPU的利用率很高，但是用户体验并不是很好。这种情况下使用多线程反而会更好，例如如果3个任务，每个任务都在一个线程中，也就是有3个线程，A任务在ThreadA中，B任务在ThreadB中，C任务在ThreadC中，那么即使A任务的计算量比较大，B，C两个任务所在的线程也不必等待A任务完成之后再工作，他们也有机会得到调度，这是由操作系统来完成的。这样就不会因为某一个任务计算量大，而导致阻塞其他任务而影响体验了。

这种情况下，CPU的利用率很高，但是由于某个人物的本身就非常耗时，同样的也会引起性能问题。

解决办法是：使用多线程。

6、多线程程序

我们将上面的模型改造成多线程的模型是怎样的呢，我们在模型5的基础上添加一个角色，管理员大叔（操作系统的角色）：

阿姨：打一个菜需要1秒

小A：200个菜

小B：3个菜

小C：2个菜

加入管理员大叔之后变成这样的了，小A打两个菜之后，大叔说，你打的菜太多了，不能因为你要打200个菜，让后面的同学都没有机会打菜，你打两个菜之后等一会，让后面的同学也有机会。

大叔让小B打两个菜，然后让小C打两个菜（小C完成），然后再让小A打两个菜（完成之后小A总共就有4个菜了），再让小B打1个菜（此时小B总共打3个菜，完成），然后小A打剩下的196个菜。

CPU的利用率：很高，阿姨在不断的工作

用户体验：不错，即使小A要打200个菜，小B，小C也有机会。当然如果小A说我是帮校长打菜，要快一点（线程优先级高），那也只能先把小A服务完

总耗时： $200 + 3 + 2 +$ （大叔指挥安排所消耗的时间，包括从小C切换回小A的时候，大叔要知道小A上次打的菜是哪两个，这次应该接着打什么菜，这相当于线程上下文切换的开销以及线程环境的保存与恢复），所以并不是线程越多越好，线程非常多的时候大叔估计会焦头烂额吧，要记住这么状态，切换来切换去也耗时间。

此时，还不是多线程。只是在多任务的程序中，加入了管理员角色（负责调度，平衡）。这个管理员就是我们的操作系统。

7、多CPU

真正的多线程。

以4核的CPU为例。

饭堂开辟4个打饭的窗口。

4个窗口是同时进行的。

但是问题在于：

4个窗口的人，同时需要打同一个菜，但是菜的分量只够1个人，怎么办？怎么去控制？如何去调度？

这就是非常著名：并发！

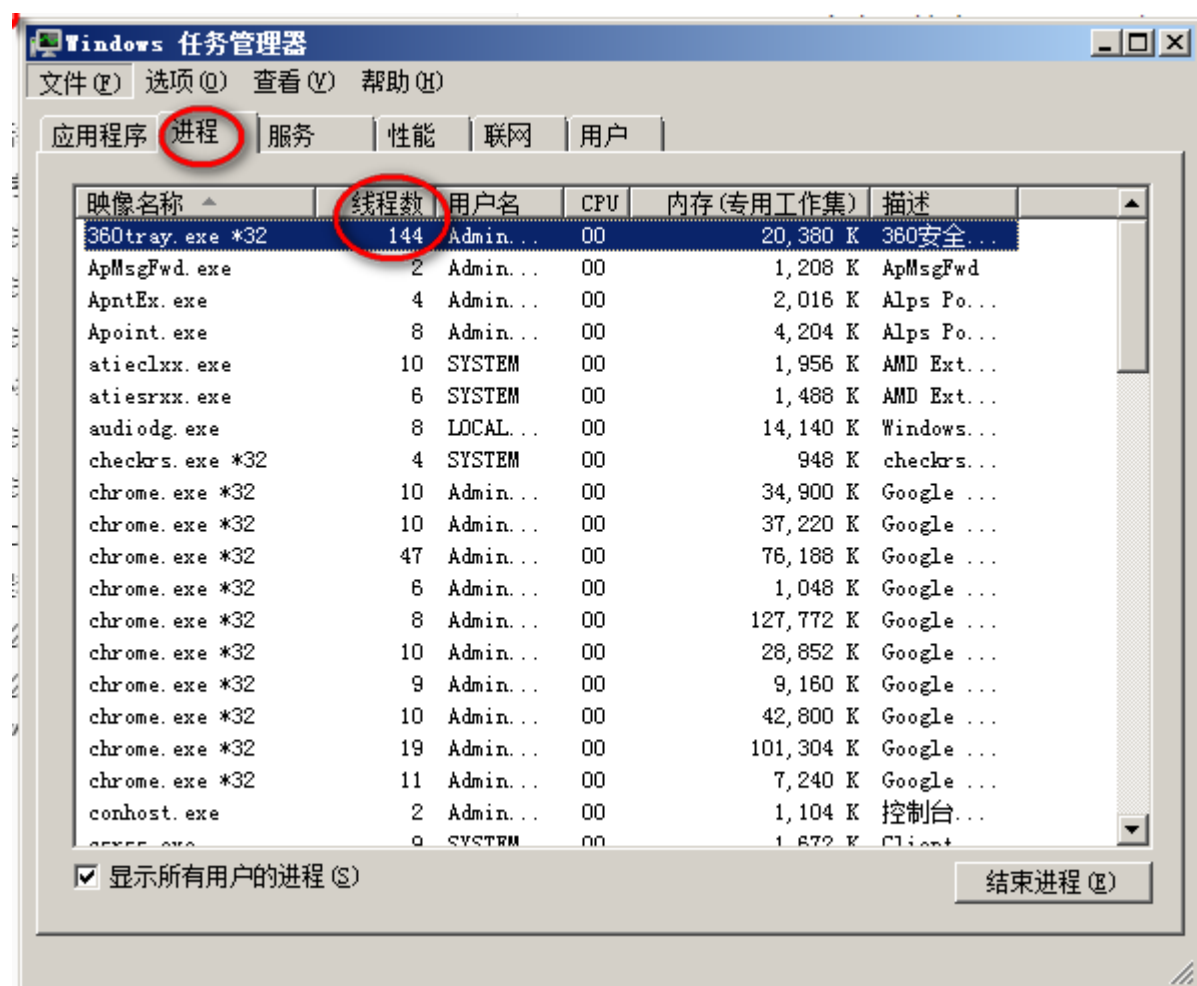
多个线程访问同一个资源。如果控制不当，会造成资源的冲突。

二、线程和进程

1、什么叫进程？

进程：是执行中一段程序，即一旦程序被载入到内存中并准备执行，它就是一个进程。

进程是表示资源分配（内存、CPU、显存）的基本概念，又是调度运行的基本单位，是系统中的并发执行的单位。





2、什么叫线程？

线程：单个进程中执行中每个任务就是一个线程。线程是进程中执行运算的最小单位。

一个线程只能属于一个进程，但是一个进程可以拥有多个线程。多线程处理就是允许一个进程中在同一时刻执行多个任务。

3、操作系统的调度

大部分操作系统(如Windows、Linux)的任务调度是采用时间片轮转的抢占式调度方式，也就是说一个任务执行一小段时间后强制暂停去执行下一个任务，每个任务轮流执行。任务执行的一小段时间叫做时间片，任务正在执行时的状态叫运行状态，任务执行一段时间后强制暂停去执行下一个任务，被暂停的任务就处于就绪状态等待下一个属于它的时间片的到来。这样每个任务都能得到执行，由于CPU的执行效率非常高，时间片非常短，在各个任务之间快速地切换，给人的感觉就是多个任务在“同时进行”，这也就是我们所说的并发(别觉得并发有多高深，它的实现很复杂，但它的概念很简单，就是一句话：多个任务同时执行)。多任务运行过程的示意图如下：

时间片 运行状态:  就绪状态: 



任务1



任务2



任务3

时间  时间的方向