转自：http://www.cnblogs.com/Survivalist/p/11527949.html

目录

进程

我们都知道计算机的核心是CPU，它承担了所有的计算任务；而操作系统是计算机的管理者，它负责任务的调度、资源的分配和管理，统领整个计算机硬件；应用程序则是具有某种功能的程序，程序是运行于操作系统之上的。

进程是一个具有一定独立功能的程序在一个数据集上的一次动态执行的过程，是操作系统进行资源分配和调度的一个独立单位，是应用程序运行的载体。进程是一种抽象的概念，从来没有统一的标准定义。

进程一般由程序、数据集合和进程控制块三部分组成。

- 程序用于描述进程要完成的功能，是控制进程执行的指令集；
- 数据集合是程序在执行时所需要的数据和工作区；
- 程序控制块(Program Control Block，简称PCB)，包含进程的描述信息和控制信息，是进程存在的唯一标志。

进程具有的特征：

- 动态性：进程是程序的一次执行过程，是临时的，有生命期的，是动态产生，动态消亡的；
- 并发性：任何进程都可以同其他进程一起并发执行；
- 独立性：进程是系统进行资源分配和调度的一个独立单位；
- 结构性：进程由程序、数据和进程控制块三部分组成。

线程

在早期的操作系统中并没有线程的概念，进程是能拥有资源和独立运行的最小单位，也是程序执行的最小单位。任务调度采用的是时间片轮转的抢占式调度方式，而进程是任务调度的最小单位，每个进程有各自独立的一块内存，使得各个进程之间内存地址相互隔离。

后来，随着计算机的发展，对CPU的要求越来越高，进程之间的切换开销较大，已经无法满足越来越复杂的程序的要求了。于是就发明了线程。

线程是程序执行中一个单一的顺序控制流程，是程序执行流的最小单元，是处理器调度和分派的基本单位。一个进程可以有一个或多个线程，各个线程之间共享程序的内存空间(也就是所在进程的内存空间)。一个标准的线程由线程ID、当前指令指针(PC)、寄存器和堆栈组成。而进程由内存空间(代码、数据、进程空间、打开的文件)和一个或多个线程组成。
（读到这里可能有的读者迷糊，感觉这和Java的内存空间模型不太一样，但如果你深入的读过深入理解Java虚拟机这本书的话你就会恍然大悟）

如上图，在任务管理器的进程一栏里，有道词典和有道云笔记就是进程，而在进程下又有着多个执行不同任务的线程。

任务调度

线程是什么？要理解这个概念，需要先了解一下操作系统的一些相关概念。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。

在一个进程中，当一个线程任务执行几毫秒后，会由操作系统的内核（负责管理各个任务）进行调度，通过硬件的计数器中断处理器，让该线程强制暂停并将该线程的寄存器放入内存中，通过查看线程列表决定接下来执行哪一个线程，并从内存中恢复该线程的寄存器，最后恢复该线程的执行，从而去执行下一个任务。
上述过程中，任务执行的那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，被暂停的线程任务状态叫做就绪状态，意为等待下一个属于它的时间片的到来。

这种方式保证了每个线程轮流执行，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：

图1：操作系统中的任务调度

进程与线程的区别

前面讲了进程与线程，但可能你还觉得迷糊，感觉他们很类似。的确，进程与线程有着千丝万缕的关系，下面就让我们一起来理一理：

1. 线程是程序执行的最小单位，而进程是操作系统分配资源的最小单位；
2. 一个进程由一个或多个线程组成，线程是一个进程中代码的不同执行路线；
3. 进程之间相互独立，但同一进程下的各个线程之间共享程序的内存空间(包括代码段、数据集、堆等)及一些进程级的资源(如打开文件和信号)，某进程内的线程在其它进程不可见；
4. 调度和切换：线程上下文切换比进程上下文切换要快得多。
   线程与进程关系的示意图：

图2：进程与线程的资源共享关系

图3：单线程与多线程的关系

总之，线程和进程都是一种抽象的概念，线程是一种比进程更小的抽象，线程和进程都可用于实现并发。 在早期的操作系统中并没有线程的概念，进程是能拥有资源和独立运行的最小单位，也是程序执行的最小单位。它相当于一个进程里只有一个线程，进程本身就是线程。所以线程有时被称为轻量级进程(Lightweight Process，LWP）。

图4：早期的操作系统只有进程，没有线程

后来，随着计算机的发展，对多个任务之间上下文切换的效率要求越来越高，就抽象出一个更小的概念——线程，一般一个进程会有多个(也可是一个)线程。

图5：线程的出现，使得一个进程可以有多个线程

多线程与多核

上面提到的时间片轮转的调度方式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。很多操作系统的书都说“同一时间点只有一个任务在执行”。那有人可能就要问双核处理器呢？难道两个核不是同时运行吗？

其实“同一时间点只有一个任务在执行”这句话是不准确的，至少它是不全面的。那多核处理器的情况下，线程是怎样执行呢？这就需要了解内核线程。

多核(心)处理器是指在一个处理器上集成多个运算核心从而提高计算能力，也就是有多个真正并行计算的处理核心，每一个处理核心对应一个内核线程。
内核线程（Kernel Thread，KLT）就是直接由操作系统内核支持的线程，这种线程由内核来完成线程切换，内核通过操作调度器对线程进行调度，并负责将线程的任务映射到各个处理器上。一般一个处理核心对应一个内核线程，比如单核处理器对应一个内核线程，双核处理器对应两个内核线程，四核处理器对应四个内核线程。

现在的电脑一般是双核四线程、四核八线程，是采用超线程技术将一个物理处理核心模拟成两个逻辑处理核心，对应两个内核线程，所以在操作系统中看到的CPU数量是实际物理CPU数量的两倍，如你的电脑是双核四线程，打开“任务管理器\性能”可以看到4个CPU的监视器，四核八线程可以看到8个CPU的监视器。

图6：双核四线程在Windows8下查看的结果

超线程技术就是利用特殊的硬件指令，把一个物理芯片模拟成两个逻辑处理核心，让单个处理器都能使用线程级并行计算，进而兼容多线程操作系统和软件，减少了CPU的闲置时间，提高的CPU的运行效率。这种超线程技术(如双核四线程)由处理器硬件的决定，同时也需要操作系统的支持才能在计算机中表现出来。

程序一般不会直接去使用内核线程，而是去使用内核线程的一种高级接口——轻量级进程（Lightweight Process，LWP），轻量级进程就是我们通常意义上所讲的线程，也被叫做用户线程。由于每个轻量级进程都由一个内核线程支持，因此只有先支持内核线程，才能有轻量级进程。用户线程与内核线程的对应关系有三种模型：一对一模型、多对一模型、多对多模型，在这以4个内核线程、3个用户线程为例对三种模型进行说明。

一对一模型

对于一对一模型来说，一个用户线程就唯一地对应一个内核线程(反过来不一定成立，一个内核线程不一定有对应的用户线程)。这样，如果CPU没有采用超线程技术(如四核四线程的计算机)，一个用户线程就唯一地映射到一个物理CPU的内核线程，线程之间的并发是真正的并发。一对一模型使用户线程具有与内核线程一样的优点，一个线程因某种原因阻塞时其他线程的执行不受影响；此处，一对一模型也可以让多线程程序在多处理器的系统上有更好的表现。

但一对一模型也有两个缺点：

1. 许多操作系统限制了内核线程的数量，因此一对一模型会使用户线程的数量受到限制；
2. 许多操作系统内核线程调度时，上下文切换的开销较大，导致用户线程的执行效率下降。

图7：一对一模型

多对一模型

多对一模型将多个用户线程映射到一个内核线程上，线程之间的切换由用户态的代码来进行，系统内核感受不到线程的实现方式。用户线程的建立、同步、销毁等都在用户态中完成，不需要内核的介入。因此相对一对一模型，多对一模型的线程上下文切换速度要快许多；此外，多对一模型对用户线程的数量几乎无限制。

但多对一模型也有两个缺点：

1. 如果其中一个用户线程阻塞，那么其它所有线程都将无法执行，因为此时内核线程也随之阻塞了；
2. 在多处理器系统上，处理器数量的增加对多对一模型的线程性能不会有明显的增加，因为所有的用户线程都映射到一个处理器上了。

图8：多对一模型

多对多模型

多对多模型结合了一对一模型和多对一模型的优点，将多个用户线程映射到多个内核线程上。由线程库负责在可用的可调度实体上调度用户线程，这使得线程的上下文切换非常快，因为它避免了系统调用。但是增加了复杂性和优先级倒置的可能性，以及在用户态调度程序和内核调度程序之间没有广泛（且高昂）协调的次优调度。

多对多模型的优点有：

1. 一个用户线程的阻塞不会导致所有线程的阻塞，因为此时还有别的内核线程被调度来执行；
2. 多对多模型对用户线程的数量没有限制；
3. 在多处理器的操作系统中，多对多模型的线程也能得到一定的性能提升，但提升的幅度不如一对一模型的高。

图9：多对多模型

在现在流行的操作系统中，大都采用多对多的模型。

查看进程与线程

一个应用程序可能是多线程的，也可能是多进程的，如何查看呢？在Windows下我们只须打开任务管理器就能查看一个应用程序的进程和线程数。按“Ctrl+Alt+Del”或右键快捷工具栏打开任务管理器。

查看进程数和线程数：

图10：查看线程数和进程数

在“进程”选项卡下，我们可以看到一个应用程序包含的线程数。如果一个应用程序有多个进程，我们能看到每一个进程，如在上图中，Google的Chrome浏览器就有多个进程。同时，如果打开了一个应用程序的多个实例也会有多个进程，如上图中我打开了两个cmd窗口，就有两个cmd进程。如果看不到线程数这一列，可以再点击“查看\选择列”菜单，增加监听的列。

查看CPU和内存的使用率：

在性能选项卡中，我们可以查看CPU和内存的使用率，根据CPU使用记录的监视器的个数还能看出逻辑处理核心的个数，如我的双核四线程的计算机就有四个监视器。

图11：查看CPU和内存的使用率

线程的生命周期

当线程的数量小于处理器的数量时，线程的并发是真正的并发，不同的线程运行在不同的处理器上。但当线程的数量大于处理器的数量时，线程的并发会受到一些阻碍，此时并不是真正的并发，因为此时至少有一个处理器会运行多个线程。

在单个处理器运行多个线程时，并发是一种模拟出来的状态。操作系统采用时间片轮转的方式轮流执行每一个线程。现在，几乎所有的现代操作系统采用的都是时间片轮转的抢占式调度方式，如我们熟悉的Unix、Linux、Windows及macOS等流行的操作系统。

我们知道线程是程序执行的最小单位，也是任务执行的最小单位。在早期只有进程的操作系统中，进程有五种状态，创建、就绪、运行、阻塞(等待)、退出。早期的进程相当于现在的只有单个线程的进程，那么现在的多线程也有五种状态，现在的多线程的生命周期与早期进程的生命周期类似。

图12：早期进程的生命周期

进程在运行过程有三种状态：就绪、运行、阻塞，创建和退出状态描述的是进程的创建过程和退出过程。

- 创建：进程正在创建，还不能运行。操作系统在创建进程时要进行的工作包括分配和建立进程控制块表项、建立资源表格并分配资源、加载程序并建立地址空间；

- 就绪：时间片已用完，此线程被强制暂停，等待下一个属于它的时间片到来；
- 运行：此线程正在执行，正在占用时间片；
- 阻塞：也叫等待状态，等待某一事件(如IO或另一个线程)执行完；
- 退出：进程已结束，所以也称结束状态，释放操作系统分配的资源。

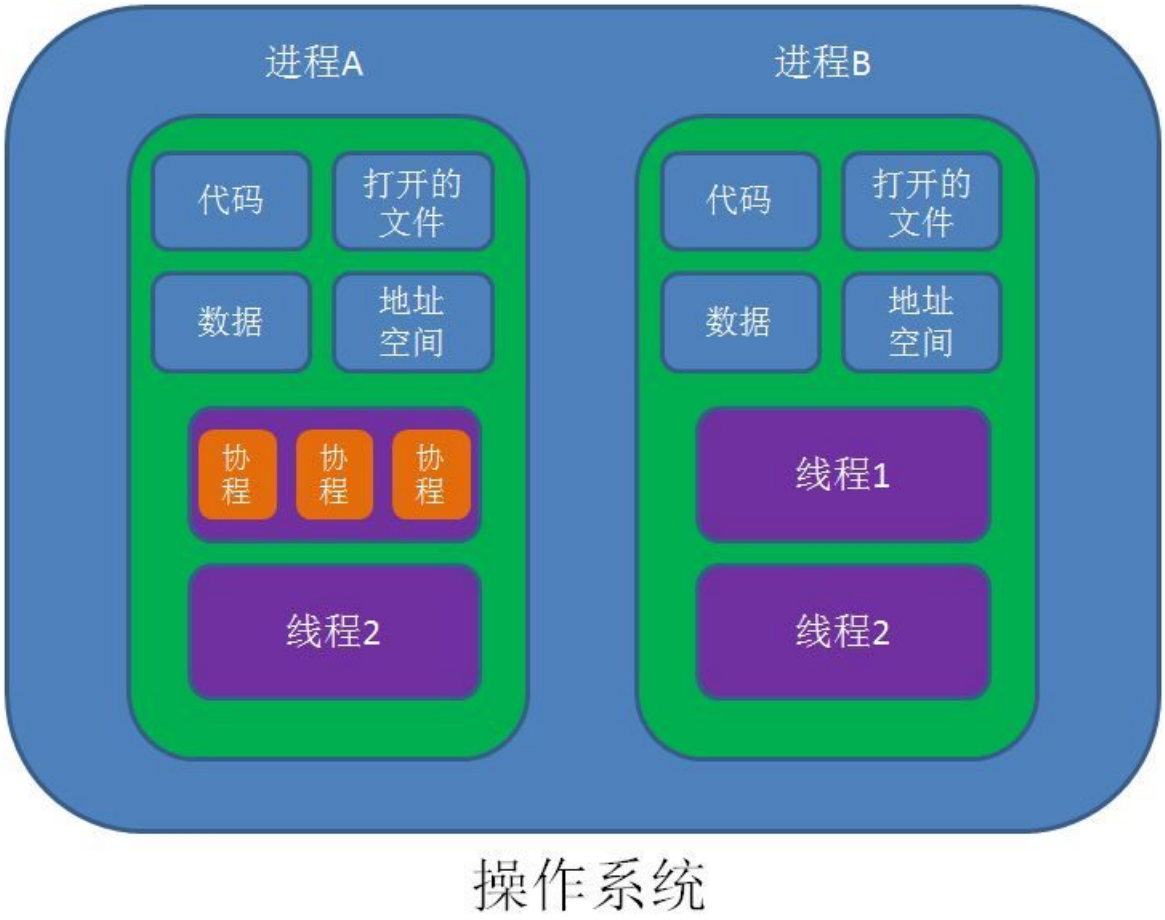
图13：线程的生命周期

- 创建：一个新的线程被创建，等待该线程被调用执行；
- 就绪：时间片已用完，此线程被强制暂停，等待下一个属于它的时间片到来；
- 运行：此线程正在执行，正在占用时间片；
- 阻塞：也叫等待状态，等待某一事件(如IO或另一个线程)执行完；
- 退出：一个线程完成任务或者其他终止条件发生，该线程终止进入退出状态，退出状态释放该线程所分配的资源。

协程

协程，英文Coroutines，是一种基于线程之上，但又比线程更加轻量级的存在，这种由程序员自己写程序来管理的轻量级线程叫做『用户空间线程』，具有对内核来说不可见的特性。

因为是自主开辟的异步任务，所以很多人也更喜欢叫它们纤程（Fiber），或者绿色线程（GreenThread）。正如一个进程可以拥有多个线程一样，一个线程也可以拥有多个协程。



协程的目的

在传统的J2EE系统中都是基于每个请求占用一个线程去完成完整的业务逻辑（包括事务）。所以系统的吞吐能力取决于每个线程的操作耗时。如果遇到很耗时的I/O行为，则整个系统的吞吐立刻下降，因为这个时候线程一直处于阻塞状态，如果线程很多的时候，会存在很多线程处于空闲状态（等待该线程执行完才能执行），造成了资源应用不彻底。

最常见的例子就是JDBC（它是同步阻塞的），这也是为什么很多人都说数据库是瓶颈的原因。这里的耗时其实是让CPU一直在等待I/O返回，说白了线程根本没有利用CPU去做运算，而是处于空转状态。而另外过多的线程，也会带来更多的ContextSwitch开销。

对于上述问题，现阶段行业里的比较流行的解决方案之一就是单线程加上异步回调。其代表派是node.js以及Java里的新秀Vert.x。

而协程的目的就是当出现长时间的I/O操作时，通过让出目前的协程调度，执行下一个任务的方式，来消除ContextSwitch上的开销。

协程的特点

1. 线程的切换由操作系统负责调度，协程由用户自己进行调度，因此减少了上下文切换，提高了效率。
2. 线程的默认Stack大小是1M，而协程更轻量，接近1K。因此可以在相同的内存中开启更多的协程。
3. 由于在同一个线程上，因此可以避免竞争关系而使用锁。
4. 适用于被阻塞的，且需要大量并发的场景。但不适用于大量计算的多线程，遇到此种情况，更好实用线程去解决。

协程的原理

当出现IO阻塞的时候，由协程的调度器进行调度，通过将数据流立刻yield掉（主动让出），并且记录当前栈上的数据，阻塞完后立刻再通过线程恢复栈，并把阻塞的结果放到这个线程上去跑，这样看上去好像跟写同步代码没有任何差别，这整个流程可以称为coroutine，而跑在由 `coroutine` 负责调度的线程称为 `Fiber` 。比如Golang里的 go关键字其实就是负责开启一个 `Fiber` ，让 `func` 逻辑跑在上面。

由于协程的暂停完全由程序控制，发生在用户态上；而线程的阻塞状态是由操作系统内核来进行切换，发生在内核态上。因此，协程的开销远远小于线程的开销，也就没有了ContextSwitch上的开销。

协程和线程的比较

比较项	线程	协程
占用资源	初始单位为1MB,固定不可变	初始一般为 2KB，可随需要而增大
调度所属	由 OS 的内核完成	由用户完成
切换开销	涉及模式切换(从用户态切换到内核态)、16个寄存器、PC、SP...等寄存器的刷新等	只有三个寄存器的值修改 - PC / SP / DX.
性能问题	资源占用太高，频繁创建销毁会带来严重的性能问题	资源占用小,不会带来严重的性能问题
数据同步	需要用锁等机制确保数据的一直性和可见性	不需要多线程的锁机制，因为只有一个线程，也不存在同时写变量冲突，在协程中控制共享资源不加锁，只需要判断状态就好了，所以执行效率比多线程高很多。

再回首能否就能天宽地阔，撒一段青春，留给山河评说，执步向前不回头就能解脱，含笑接受错过，秋风悲过叶落，家燕明年依然返穴，太阳的升月亮的缺，有失也必会有得