## 一．上下文切换技术

1. 上下文：程序再执行中的某一个状态。通常情况下我们会调用栈来表示这一个状态，栈记载了每个调用层执行到了哪里，还有执行时的环境情况等
2. 上下文切换：表达了一种从一个上下文切换到另一个上下文执行的技术
3. 调度：决定哪个上下文可以获得接下去的cpu时间的方法

### 进程

#### 概述

进程（Process）是计算机中的程序关于某数据集合上的一次运行活动，是系统进行资源分配和调度的基本单位，是[操作系统](http://baike.baidu.com/view/880.htm" \t "_blank)结构的基础。在早期面向进程设计的计算机结构中，进程是程序的基本执行实体；在当代面向线程设计的计算机结构中，进程是线程的容器。程序是指令、数据及其组织形式的描述，进程是程序的实体。

进程是一个具有独立功能的程序关于某个数据集合的一次运行活动。它可以申请和拥有系统资源，是一个动态的概念，是一个活动的实体。它不只是程序的[代码](http://baike.baidu.com/view/41.htm)，还包括当前的活动，通过[程序计数器](http://baike.baidu.com/view/178145.htm)的值和处理[寄存器](http://baike.baidu.com/view/6159.htm)的内容来表示。

进程的概念主要有两点：第一，进程是一个实体。每一个进程都有它自己的地址空间，一般情况下，包括[文本](http://baike.baidu.com/view/300107.htm)区域（text region）、数据区域（data region）和[堆栈](http://baike.baidu.com/view/93201.htm" \t "_blank)（stack region）。文本区域存储处理器执行的代码；数据区域存储变量和进程执行期间使用的动态分配的内存；堆栈区域存储着活动过程调用的指令和本地变量。第二，进程是一个“执行中的程序”。程序是一个没有生命的实体，只有[处理](http://baike.baidu.com/view/989420.htm)器赋予程序生命时（操作系统执行之），它才能成为一个活动的实体，我们称其为[进程](http://baike.baidu.com/view/19746.htm" \t "_blank)。

进程是一种古老而典型的上下文系统，每个进程有独立的地址空间，资源句柄，他们互相之间不发生干扰。

每个进程在内核中会有一个数据结构进行描述，我们称其为进程描述符。这些描述符包含了系统管理进程所需的信息，并且放在一个叫做任务队列的队列里面。

很显然，当新建进程时，我们需要分配新的进程描述符，并且分配新的地址空间(和父地址空间的映射保持一致，但是两者同时进入COW状态)。这些过程需要一定的开销。

#### 进程状态

进程状态归结为三个最主要的状态：就绪态，运行态，睡眠态。就绪和执行可以互相转换，基本这就是调度的过程。而当执行态程序需要等待某些条件(最典型就是IO)时，就会陷入睡眠态。而条件达成后，一般会自动进入就绪。

#### 阻塞

**关于linux的文件句柄fd：**在Linux中，值为0、1、2的fd分别代表标准输入、标准输出和标准错误输出。在程序中打开文件得到的fd从3开始增长。 fd具体是什么呢?在内核中，每一个进程都有一个私有的“打开文件表”，这个表是一个指针数组，每一个元素都指向一个内核的打开文件对象。而fd，就是这个表的下标。当用户打开一个文件时，内核会在内部生成一个打开文件对象，并在这个表里找到一个空项，让这一项指向生成的打开文件对象，并返回这一项的下标 作为fd。由于这个表处于内核，并且用户无法访问到，因此用户即使拥有fd，也无法得到打开文件对象的地址，只能够通过系统提供的函数来操作。

当进程需要在某个文件句柄上做IO，这个fd又没有数据给他的时候，就会发生阻塞。具体来说，就是记录XX进程阻塞在了XX fd上，然后将进程标记为睡眠态，并调度出去。当fd上有数据时(例如对端发送的数据到达)，就会唤醒阻塞在fd上的进程。进程会随后进入就绪队列，等待合适的时间被调度。

阻塞后的唤醒也是一个很有意思的话题。当多个上下文阻塞在一个fd上(虽然不多见，但是后面可以看到一个例子)，而且fd就绪时，应该唤醒多少个上下文呢？传统上应当唤醒所有上下文，因为如果仅唤醒一个，而这个上下文又不能消费所有数据时，就会使得其他上下文处于无谓的死锁中。

但是有个著名的例子——accept，也是使用读就绪来表示收到的。如果试图用多个线程来accept会发生什么？当有新连接时，所有上下文都会就绪，但是只有第一个可以实际获得fd，其他的被调度后又立刻阻塞。这就是惊群问题thundering herd problem。

现代linux内核已经解决了这个问题，方法惊人的简单——accept方法加锁。

(inet\_connection\_sock.c:inet\_csk\_wait\_for\_connect)

#### 守护进程

Daemon()程序是一直运行的服务端[程序](http://baike.baidu.com/subview/17674/13521310.htm" \t "_blank)，又称为[守护进程](http://baike.baidu.com/view/53123.htm)。通常在系统后台运行，没有控制终端，不与前台交互，Daemon程序一般作为[系统服务](http://baike.baidu.com/view/685551.htm" \t "_blank)使 用。Daemon是长时间运行的进程，通常在系统启动后就运行，在系统关闭时才结束。一般说Daemon程序在后台运行，是因为它没有控制终端，无法和前 台的用户交互。Daemon程序一般都作为服务程序使用，等待客户端程序与它通信。我们也把运行的Daemon程序称作守护进程。

　　所谓守护 线程，是指在程序运行的时候在后台提供一种通用服务的线程，比如垃圾回收线程就是一个很称职的守护者，并且这种线程并不属于程序中不可或缺的部分。因此， 当所有的非守护线程结束时，程序也就终止了，同时会杀死进程中的所有守护线程。反过来说，只要任何非守护线程还在运行，程序就不会终止。

用户线程和守护线程两者几乎没有区别，唯一的不同之处就在于虚拟机的离开：如果用户线程已经全部退出运行了，只剩下守护线程存在了，虚拟机也就退出了。 因为没有了被守护者，守护线程也就没有工作可做了，也就没有继续运行程序的必要了。

### 线程

线程是一种轻量进程，实际上在linux内核中，两者几乎没有差别，除了一点——线程并不产生新的地址空间和资源描述符表，而是复用父进程的。但是无论如何，线程的调度和进程一样，必须陷入内核态

线程，有时被称为轻量级进程(Lightweight Process，LWP），是程序执行流的最小单元。一个标准的线程由线程ID，当前指令[指针](http://baike.baidu.com/view/159417.htm" \t "_blank)(PC），[寄存器](http://baike.baidu.com/view/6159.htm" \t "_blank)集合和[堆栈](http://baike.baidu.com/view/93201.htm)组 成。另外，线程是进程中的一个实体，是被系统独立调度和分派的基本单位，线程自己不拥有系统资源，只拥有一点儿在运行中必不可少的资源，但它可与同属一个 进程的其它线程共享进程所拥有的全部资源。一个线程可以创建和撤消另一个线程，同一进程中的多个线程之间可以并发执行。由于线程之间的相互制约，致使线程 在运行中呈现出间断性。线程也有[就绪](http://baike.baidu.com/view/654230.htm)、[阻塞](http://baike.baidu.com/view/497285.htm)和[运行](http://baike.baidu.com/view/1026025.htm)三种基本状态。就绪状态是指线程具备运行的所有条件，逻辑上可以运行，在等待处理机；运行状态是指线程占有处理机正在运行；阻塞状态是指线程在等待一个事件（如某个信号量），逻辑上不可执行。每一个程序都至少有一个线程，若程序只有一个线程，那就是程序本身。

线程是程序中一个单一的顺序控制流程。进程内一个相对独立的、可调度的执行单元，是系统独立调度和分派CPU的基本单位指[运行](http://baike.baidu.com/view/1026025.htm" \t "_blank)中的程序的调度单位。在单个程序中同时运行多个线程完成不同的工作，称为[多线程](http://baike.baidu.com/view/65706.htm)。

　　线程是程序中一个单一的顺序控制流程。进程内一个相对独立的、可调度的执行单元，是系统独立调度和分派CPU的基本单位指[运行](http://baike.baidu.com/view/1026025.htm" \t "_blank)中的程序的调度单位。在单个程序中同时运行多个线程完成不同的工作，称为[多线程](http://baike.baidu.com/view/65706.htm)。

### 多进程和多线程

进程是资源分配的最小单位，线程是CPU调度的最小单位。线程和进程的区别在于,子进程和父进程有不同的代码和数据空间,而多个线程则共享数据空 间,每个线程有自己的执行堆栈和程序计数器为其执行上下文.多线程主要是为了节约CPU时间,发挥利用,根据具体情况而定. 线程的运行中需要使用计算机的内存资源和CPU。

　　多进程： 进程是程序在计算机上的一次执行活动。当你运行一个程序，你就启动了一个进程。显然，程序是死的(静态的)，进程是活的(动态的)。进程可以分为系统进程 和用户进程。凡是用于完成操作系统的各种功能的进程就是系统进程，它们就是处于运行状态下的操作系统本身;所有由用户启动的进程都是用户进程。进程是操作 系统进行资源分配的单位。 进程又被细化为线程，也就是一个进程下有多个能独立运行的更小的单位。在同一个时间里，同一个计算机系统中如果允许两个或两个以上的进程处于运行状态，这 便是多任务。现代的操作系统几乎都是多任务操作系统，能够同时管理多个进程的运行。 多任务带来的好处是明显的，比如你可以边听mp3边上网，与此同时甚至可以将下载的文档打印出来，而这些任务之间丝毫不会相互干扰。那么这里就涉及到并行 的问题，俗话说，一心不能二用，这对计算机也一样，原则上一个CPU只能分配给一个进程，以便运行这个进程。我们通常使用的计算机中只有一个CPU，也就 是说只有一颗心，要让它一心多用，同时运行多个进程，就必须使用并发技术。实现并发技术相当复杂，最容易理解的是“时间片轮转进程调度算法”，它的思想简 单介绍如下：在操作系统的管理下，所有正在运行的进程轮流使用CPU，每个进程允许占用CPU的时间非常短(比如10毫秒)，这样用户根本感觉不出来 CPU是在轮流为多个进程服务，就好象所有的进程都在不间断地运行一样。但实际上在任何一个时间内有且仅有一个进程占有CPU。 如果一台计算机有多个CPU，情况就不同了，如果进程数小于CPU数，则不同的进程可以分配给不同的CPU来运行，这样，多个进程就是真正同时运行的，这 便是并行。但如果进程数大于CPU数，则仍然需要使用并发技术。 进行CPU分配是以线程为单位的，一个进程可能由多个线程组成，这时情况更加复杂，但简单地说，有如下关系：

　　总线程数<= CPU数量：并行运行

　　总线程数> CPU数量：并发运行

　　并行运行的效率显然高于并发运行，所以在多CPU的计算机中，多任务的效率比较高。但是，如果在多CPU计算机中只运行一个进程(线程)，就不 能发挥多CPU的优势。 这里涉及到多任务操作系统的问题，多任务操作系统(如Windows)的基本原理是:操作系统将CPU的时间片分配给多个线程,每个线程在操作系统指定的 时间片内完成(注意,这里的多个线程是分属于不同进程的).操作系统不断的从一个线程的执行切换到另一个线程的执行,如此往复,宏观上看来,就好像是多个 线程在一起执行.由于这多个线程分属于不同的进程,因此在我们看来,就好像是多个进程在同时执行,这样就实现了多任务

　　多线程：在计算机编程中，一个基本的概念就是同时对多个任务加以控制。许多程序设计问题都要求程序能够停下手头的工作，改为处理其他一些问题， 再返回主进程。可以通过多种途径达到这个目的。最开始的时候，那些掌握机器低级语言的程序员编写一些“中断服务例程”，主进程的暂停是通过硬件级的中断实 现的。尽管这是一种有用的方法，但编出的程序很难移植，由此造成了另一类的代价高昂问题。中断对那些实时性很强的任务来说是很有必要的。但对于其他许多问 题，只要求将问题划分进入独立运行的程序片断中，使整个程序能更迅速地响应用户的请求。

　　最开始，线程只是用于分配单个处理器的处理时间的一种工具。但假如操作系统本身支持多个处理器，那么每个线程都可分配给一个不同的处理器，真正 进入“并行运算”状态。从程序设计语言的角度看，多线程操作最有价值的特性之一就是程序员不必关心到底使用了多少个处理器。程序在逻辑意义上被分割为数个 线程;假如机器本身安装了多个处理器，那么程序会运行得更快，毋需作出任何特殊的调校。根据前面的论述，大家可能感觉线程处理非常简单。但必须注意一个问 题：共享资源!如果有多个线程同时运行，而且它们试图访问相同的资源，就会遇到一个问题。举个例子来说，两个线程不能将信息同时发送给一台打印机。为解决 这个问题，对那些可共享的资源来说(比如打印机)，它们在使用期间必须进入锁定状态。所以一个线程可将资源锁定，在完成了它的任务后，再解开(释放)这个 锁，使其他线程可以接着使用同样的资源。

　　多线程是为了同步完成多项任务，不是为了提高运行效率，而是为了提高资源使用效率来提高系统的效率。线程是在同一时间需要完成多项任务的时候实现的。

　　一个采用了多线程技术的应用程序可以更好地利用系统资源。其主要优势在于充分利用了CPU的空闲时间片，可以用尽可能少的时间来对用户的要求做 出响应，使得进程的整体运行效率得到较大提高，同时增强了应用程序的灵活性。更为重要的是，由于同一进程的所有线程是共享同一内存，所以不需要特殊的数据 传送机制，不需要建立共享存储区或共享文件，从而使得不同任务之间的协调操作与运行、数据的交互、资源的分配等问题更加易于解决。

进程间通信(IPC，Inter-Process Communication)，指至少两个进程或线程间传送数据或信号的一些技术或方法。进程是计算机系统分配资源的最小单位。每个进程都有自己的一部分 独立的系统资源，彼此是隔离的。为了能使不同的进程互相访问资源并进行协调工作，才有了进程间通信。这些进程可以运行在同一计算机上或网络连接的不同计算 机上。

　　进程间通信技术包括消息传递、同步、共享内存和远程过程调用。IPC是一种标准的Unix通信机制。

## 二．传统网络服务模型

### 进程模型

为每个客户分配一个进程。优点是业务隔离，在一个进程中出现的错误不至于影响整个系统，甚至其他进程。Oracle传统上就是进程模型。缺点是进程的分配和释放有非常高的成本。因此Oracle需要连接池来保持连接减少新建和释放，同时尽量复用连接而不是随意的新建连接。

### 线程模型

为每客户分配一个线程。优点是更轻量，建立和释放速度更快，而且多个上下文间的通讯速度非常快。缺点是一个线程出现问题容易将整个系统搞崩溃。

一个例子：

py\_http\_fork\_thread.py

在这个例子中，线程模式和进程模式可以轻易的互换。

如何工作的：

1. 父进程监听服务端口
2. 在有新连接建立的时候，父进程执行fork，产生一个子进程副本
3. 如果子进程需要的话，可以exec(例如CGI)
4. 父进程执行(理论上应当先执行子进程，因为exec执行的快可以避免COW)到accept后，发生阻塞
5. 上下文调度，内核调度器选择下一个上下文，如无意外，应当就是刚刚派生的子进程
6. 子进程进程进入读取处理状态，阻塞在read调用上，所有上下文均进入睡眠态
7. 随着SYN或者数据报文到来，CPU会唤醒对应fd上阻塞的上下文(wait\_queue)，切换到就绪态，并加入调度队列
8. 上下文继续执行到下一个阻塞调用，或者因为时间片耗尽被挂起

评价

* 同步模型，编写自然，每个上下文可以当作其他上下文不存在一样的操作，每次读取数据可以当作必然能读取到。
* 进程模型自然的隔离了连接。即使程序复杂且易崩溃，也只影响一个连接而不是在整个系统。
* 生成和释放开销很大(效率测试的进程fork和线程模式开销测试)，需要考虑复用。
* 进程模式的多客户通讯比较麻烦，尤其在共享大量数据的时候。

注意在python中，虽然有GIL，但是一个线程陷入到网络IO的时候，GIL是解锁的。因此从调用开始到调用结束，减去CPU切换到其他上下文的时间，是可以多线程的。现象是，在此种状况下可以观测到短暂的python CPU用量超过100%。

如果执行多个上下文，可以充分利用这段时间。所观测到的结果就是，只能单核的python，在小范围内，其随着并发数上升，性能居然会跟着上升。如果将这个过程转移到一台物理机上执行，那么基本不能得出这样的结论。这主要是因为虚拟机上内核陷入的开销更高。

## 三．C10k问题

### 描述

当同时连接数在10K左右时，传统模型就不再适用。实际上在效率测试报告的线程切换开销一节可以看到，超过1K后性能就差的一塌糊涂了。

### 进程模型的问题：

在C10K的时候，启动和关闭这么多进程是不可接受的开销。事实上单纯的进程fork模型在C1K时就应当抛弃了。

Apache的prefork模型，是使用预先分配(pre)的进程池。这些进程是被复用的。但即便是复用，本文所描述的很多问题仍不可避免。

### 线程模式的问题

从任何测试都可以表明，线程模式比进程模式更耐久一些，性能更好。但是在面对C10K还是力不从心的。问题是，线程模式的问题出在哪里呢？

### 内存？

有些人可能认为线程模型的失败首先在于内存。如果你这么认为，一定是因为你查阅了非常老的资料，并且没仔细思考过。

你可能看到资料说，一个线程栈会消耗8M内存(linux默认值，ulimit可以看到)，512个线程栈就会消耗4G内存，而10K个线程就是80G。所以首先要考虑调整栈深度，并考虑爆栈问题。

听起来很有道理，问题是——linux的栈是通过缺页来分配内存的(How does stack allocation work in Linux?)，不是所有栈地址空间都分配了内存。因此，8M是最大消耗，实际的内存消耗只会略大于实际需要的内存(内部损耗，每个在4k以内)。但是内存一旦被分配，就很难回收(除非线程结束)，这是线程模式的缺陷。

这个问题提出的前提是，32位下地址空间有限。虽然10K个线程不一定会耗尽内存，但是512个线程一定会耗尽地址空间。然而这个问题对于目前已经成为主流的64位系统来说根本不存在。

### 内核陷入开销？

所谓内核陷入开销，就是指CPU从非特权转向特权，并且做输入检查的一些开销。这些开销在不同的系统上差异很大。

线程模型主要通过陷入切换上下文，因此陷入开销大听起来有点道理。实际上，这也是不成立的。线程在什么时候发生陷入切换？正常情况下，应当是IO阻塞的时候。同样的IO量，难道其他模型就不需要陷入了么？只是非阻塞模型有很大可能直接返回，并不发生上下文切换而已。

### 线程模型的问题在于切换成本高

熟悉linux内核的应该知道，近代linux调度器经过几个阶段的发展。

1. linux2.4的调度器
2. O(1)调度器
3. CFS

实际上直到O(1)，调度器的调度复杂度才和队列长度无关。在此之前，过多的线程会使得开销随着线程数增长(不保证线性)。

O(1)调度器看起来似乎是完全不随着线程的影响。但是这个调度器有显著的缺点——难于理解和维护，并且在一些情况下会导致交互式程序响应缓慢。

CFS使用红黑树管理就绪队列。每次调度，上下文状态转换，都会查询或者变更红黑树。红黑树的开销大约是O(logm)，其中m大约为活跃上下文数(准确的说是同优先级上下文数)，大约和活跃的客户数相当。

因此，每当线程试图读写网络，并遇到阻塞时，都会发生O(logm)级别的开销。而且每次收到报文，唤醒阻塞在fd上的上下文时，同样要付出O(logm)级别的开销。

### 分析

O(logm)的开销看似并不大，但是却是一个无法接受的开销。因为IO阻塞是一个经常发生的事情。每次IO阻塞，都会发生开销。而且决定活跃线程数的是用户，这不是我们可控制的。更糟糕的是，当性能下降，响应速度下降时。同样的用户数下，活跃上下文会上升(因为响应变慢了)。这会进一步拉低性能。

问题的关键在于，http服务并不需要对每个用户完全公平，偶尔某个用户的响应时间大大的延长了是可以接受的。在这种情况下，使用红黑树去组织待处理fd列表（其实是上下文列表），并且反复计算和调度，是无谓的开销。

## 四．多路复用

### 简述

要突破C10K问题，必须减少系统内活跃上下文数(其实未必，例如换一个调度器，例如使用RT的SCHED\_RR)，因此就要求一个上下文同时处理多个链接。而要做到这点，就必须在每次系统调用读取或写入数据时立刻返回。否则上下文持续阻塞在调用上，如何能够复用？这要求fd处于非阻塞状态，或者数据就绪。

上文所说的所有IO操作，其实都特指了他的阻塞版本。所谓阻塞，就是上下文在IO调用上等待直到有合适的数据为止。这种模式给人一种“只要读取数据就必定能读到”的感觉。而非阻塞调用，就是上下文立刻返回。如果有数据，带回数据。如果没有数据，带回错误(EAGAIN)。因此，“虽然发生错误，但是不代表出错”。

但是即使有了非阻塞模式，依然绕不过就绪通知问题。如果没有合适的就绪通知技术，我们只能在多个fd中盲目的重试，直到碰巧读到一个就绪的fd为止。这个效率之差可想而知。

在就绪通知技术上，有两种大的模式——就绪事件通知和异步IO。其差别简要来说有两点。就绪通知维护一个状态，由用户读取。而异步IO由系统调用用户的回调函数。就绪通知在数据就绪时就生效，而异步IO直到数据IO完成才发生回调。

linux下的主流方案一直是就绪通知，其内核态异步IO方案甚至没有被封装到glibc里去。围绕就绪通知，linux总共提出过三种解决方案。我们绕过select和poll方案，看看epoll方案的特性。

另外提一点。有趣的是，当使用了epoll后(更准确说只有在LT模式下)，fd是否为非阻塞其实已经不重要了。因为epoll保证每次去读取的时候都能读到数据，因此不会阻塞在调用上。

### epoll

用户可以新建一个epoll文件句柄，并且将其他fd和这个"epoll fd"关联。此后可以通过epoll fd读取到所有就绪的文件句柄。

epoll有两大模式，ET和LT。LT模式下，每次读取就绪句柄都会读取出完整的就绪句柄。而ET模式下，只给出上次到这次调用间新就绪的句柄。换个说法，如果ET模式下某次读取出了一个句柄，这个句柄从未被读取完过——也就是从没有从就绪变为未就绪。那么这个句柄就永远不会被新的调用返回，哪怕上面其实充满了数据——因为句柄无法经历从非就绪变为就绪的过程。

类似CFS，epoll也使用了红黑树——不过是用于组织加入epoll的所有fd。epoll的就绪列表使用的是双向队列。这方便系统将某个fd加入队列中，或者从队列中解除。

要进一步了解epoll的具体实现，可以参考这篇linux下poll和epoll内核源码剖析。

性能

如果使用非阻塞函数，就不存在阻塞IO导致上下文切换了，而是变为时间片耗尽被抢占（大部分情况下如此），因此读写的额外开销被消除。而epoll的常规操作，都是O(1)量级的。而epoll wait的复制动作，则和当前需要返回的fd数有关(在LT模式下几乎就等同于上面的m，而ET模式下则会大大减少)。

但是epoll存在一点细节问题。epoll fd的管理使用红黑树，因此在加入和删除时需要O(logn)复杂度(n为总连接数)，而且关联操作还必须每个fd调用一次。因此在大连接量下频繁建立和关闭连接仍然有一定性能问题(超短连接)。不过关联操作调用毕竟比较少。如果确实是超短连接，tcp连接和释放开销就很难接受了，所以对总体性能影响不大。

固有缺陷

原理上说，epoll实现了一个wait\_queue的回调函数，因此原理上可以监听任何能够激活wait\_queue的对象。但是epoll的最大问题是无法用于普通文件，因为普通文件始终是就绪的——虽然在读取的时候不是这样。

这导致基于epoll的各种方案，一旦读到普通文件上下文仍然会阻塞。golang为了解决这个问题，在每次调用syscall的时候，会独立的启动一个线程，在独立的线程中进行调用。因此golang在IO普通文件的时候网络不会阻塞。

## 五．事件通知机制下的几种程序设计模型

### 简述

使用通知机制的一大缺憾就是，用户进行IO操作后会陷入茫然——IO没有完成，所以当前上下文不能继续执行。但是由于复用线程的要求，当前线程还需要接着执行。所以，在如何进行异步编程上，又分化出数种方案。

### 用户态调度

首先需要知道的一点就是，异步编程大多数情况下都伴随着用户态调度问题——即使不使用上下文技术。

因为系统不会自动根据fd的阻塞状况来唤醒合适的上下文了，所以这个工作必须由其他人——一般就是某种框架——来完成。

你可以想像一个fd映射到对象的大map表，当我们从epoll中得知某个fd就绪后，需要唤醒某种对象，让他处理fd对应的数据。

当然，实际情况会更加复杂一些。原则上所有不占用CPU时间的等待都需要中断执行，陷入睡眠，并且交由某种机构管理，等待合适的机会被唤醒。例如sleep，或是文件IO，还有lock。更精确的说，所有在内核里面涉及到wait\_queue的，在框架里面都需要做这种机制——也就是把内核的调度和等待搬到用户态来。

当然，其实也有反过来的方案——就是把程序扔到内核里面去。其中最著名的实例大概是微软的http服务器了。

这个所谓的“可唤醒可中断对象”，用的最多的就是协程。

### 协程

（通俗易懂的回答:让原来要使用异步+回调方式写的非人类代码,可以用看似同步的方式写出来）

协程是一种编程组件，可以在不陷入内核的情况进行上下文切换。如此一来，我们就可以把协程上下文对象关联到fd，让fd就绪后协程恢复执行。当然，由于当前地址空间和资源描述符的切换无论如何需要内核完成，因此协程所能调度的，只有在同一进程中的不同上下文而已。

一个程序可以包含多个协程，可以对比与一个进程包含多个线程，因而下面我们来比较协程和线程。我们知道多个线程相对独立，有自己的上下文，切换受系统控制；而协程也相对独立，有自己的上下文，但是其切换由自己控制，由当前协程切换到其他协程由当前协程来控制。

没有啥复杂的东西，考虑清楚需求，就可以很自然的衍生出这些解决方案。

* 一开始大家想要同一时间执行那么三五个程序，大家能一块跑一跑。特别是UI什么的，别一上计算量比较大的玩意就跟死机一样。于是就有了**并发**，从程序员的角度可以看成是多个独立的逻辑流。内部可以是多cpu并行，也可以是单cpu时间分片，能快速的切换逻辑流，看起来像是大家一块跑的就行。
* 但是一块跑就有问题了。我计算到一半，刚把多次方程解到最后一步，你突然插进来，我的中间状态咋办，我用来储存的内存被你覆盖了咋办？所以跑在一个cpu里面的并发都需要处理上下文切换的问题。**进程**就是这样抽象出来个一个概念，搭配虚拟内存、进程表之类的东西，用来管理独立的程序运行、切换。
* 后来一电脑上有了好几个cpu，好咧，大家都别闲着，一人跑一进程。就是所谓的**并行**。
* 因为程序的使用涉及大量的计算机资源配置，把这活随意的交给用户程序，非常容易让整个系统分分钟被搞跪，资源分配也很难做到相对的公平。所以核心的操作需要陷入内核(kernel)，切换到操作系统，让老大帮你来做。
* 有的时候碰着I/O访问，阻塞了后面所有的计算。空着也是空着，老大就直接把CPU切换到其他进程，让人家先用着。当然除了I\O阻塞，还有时钟阻塞等等。一开始大家都这样弄，后来发现不成，太慢了。为啥呀，一切换进程得反复进入内核，置换掉一大堆状态。进程数一高，大部分系统资源就被进程切换给吃掉了。后来搞出**线程**的概念，大致意思就是，这个地方阻塞了，但我还有其他地方的逻辑流可以计算，这些逻辑流是共享一个地址空间的，不用特别麻烦的切换页表、刷新TLB，只要把寄存器刷新一遍就行，能比切换进程开销少点。
* 如果连时钟阻塞、 线程切换这些功能我们都不需要了，自己在进程里面写一个逻辑流调度的东西。那么我们即可以利用到并发优势，又可以避免反复系统调用，还有进程切换造成的开销，分分钟给你上几千个逻辑流不费力。这就是**用户态线程**。
* 从上面可以看到，实现一个用户态线程有两个必须要处理的问题：一是碰着阻塞式I\O会导致整个进程被挂起；二是由于缺乏时钟阻塞，进程需要自己拥有调度线程的能力。如果一种实现使得每个线程需要自己通过调用某个方法，主动交出控制权。那么我们就称这种用户态线程是协作式的，即是**协程**。

**本质上协程就是用户空间下的线程。**

### 这是如何做到的呢？

我们在内核里实行上下文切换的时候，其实是将当前所有寄存器保存到内存中，然后从另一块内存中载入另一组已经被保存的寄存器。对于图灵机来说，当前状态寄存器意味着机器状态——也就是整个上下文。其余内容，包括栈上内存，堆上对象，都是直接或者间接的通过寄存器来访问的。

但是请仔细想想，寄存器更换这种事情，似乎不需要进入内核态么。事实上我们在用户态切换的时候，就是用了类似方案。

C coroutine的实现，基本大多是保存现场和恢复之类的过程。python则是保存当前thread的top frame(greenlet)。

但是非常悲剧的，纯用户态方案(setjmp/longjmp)在多数系统上执行的效率很高，但是并不是为了协程而设计的。setjmp并没有拷贝整个栈(大多数的coroutine方案也不应该这么做)，而是只保存了寄存器状态。这导致新的寄存器状态和老寄存器状态共享了同一个栈，从而在执行时互相破坏。而完整的coroutine方案应当在特定时刻新建一个栈。

而比较好的方案(makecontext/swapcontext)则需要进入内核(sigprocmask)，这导致整个调用的性能非常低。

### 协程与线程的关系

首先我们可以明确，协程不能调度其他进程中的上下文。而后，每个协程要获得CPU，都必须在线程中执行。因此，协程所能利用的CPU数量，和用于处理协程的线程数量直接相关。

作为推论，在单个线程中执行的协程，可以视为单线程应用。这些协程，在未执行到特定位置(基本就是阻塞操作)前，是不会被抢占，也不会和其他CPU上的上下文发生同步问题的。因此，一段协程代码，中间没有可能导致阻塞的调用，执行在单个线程中。那么这段内容可以被视为同步的。

我们经常可以看到某些协程应用，一启动就是数个进程。这并不是跨进程调度协程。一般来说，这是将一大群fd分给多个进程，每个进程自己再做fd-协程对应调度。

基于就绪通知的协程框架

1. 首先需要包装read/write，在发生read的时候检查返回。如果是EAGAIN，那么将当前协程标记为阻塞在对应fd上，然后执行调度函数。
2. 调度函数需要执行epoll(或者从上次的返回结果缓存中取数据，减少内核陷入次数)，从中读取一个就绪的fd。如果没有，上下文应当被阻塞到至少有一个fd就绪。
3. 查找这个fd对应的协程上下文对象，并调度过去。
4. 当某个协程被调度到时，他多半应当在调度器返回的路上——也就是read/write读不到数据的时候。因此应当再重试读取，失败的话返回1。
5. 如果读取到数据了，直接返回。

这样，异步的数据读写动作，在我们的想像中就可以变为同步的。而我们知道同步模型会极大降低我们的编程负担。

### CPS模型

其实这个模型有个更流行的名字——回调模型。之所以扯上CPS这么高大上的玩意，主要是里面涉及不少有趣的话题。

首先是回调模型的大致过程。在IO调用的时候，同时传入一个函数，作为返回函数。当IO结束时，调用传入的函数来处理下面的流程。这个模型听起来挺简单的。

然后是CPS。用一句话来描述这个模型——他把一切操作都当作了IO，无论干什么，结果要通过回调函数来返回。从这个角度来说，IO回调模型只能被视作CPS的一个特例。

例如，我们需要计算1+2\*3，在cps里面就需要这么写：

mul(lambda x: add(pprint.pprint, x, 1), 2, 3)

其中mul和add在python里面如下定义：

add = lambda f, \*nums: f(sum(nums))

mul = lambda f, \*nums: f(reduce(lambda x,y: x\*y, nums))

而且由于python没有TCO，所以这样的写法会产生非常多的frame。

但是要正确理解这个模型，你需要仔细思考一下以下几个问题：

1. 函数的调用过程为什么必须是一个栈？
2. IO过程在什么时间发生？调用发生时，还是回调时？
3. 回调函数从哪里调用？如果当时利用工具去看上下文的话，调用栈是什么样子的？

### 函数组件和返回值

不知道你是否思考过为什么函数调用层级(上下文栈)会被表述为一个栈——是否有什么必要性，必须将函数调用的过程定义为一个栈呢？

原因就是返回值和同步顺序。对于大部分函数，我们需要得到函数计算的返回值。而要得到返回值，调用者就必须阻塞直到被调用者返回为止。因此调用者的执行状态就必须被保存，等到被调用者返回后继续——从这点来说，调用其实是最朴素的上下文切换手段。而对于少部分无需返回的函数，我们又往往需要他的顺序外部效应——例如干掉了某个进程，开了一个灯，或者仅仅是在环境变量里面添加了一项内容。而顺序外部效应同样需要等待被调用者返回以表明这个外部效应已经发生。

那么，如果我们不需要返回值也不需要顺序的外部效应呢？例如启动一个背景程序将数据发送到对端，无需保证发送成功的情况下。或者是开始一个数据抓取行为，无需保证抓取的成功。

通常这种需求我们就凑合着用一个同步调用混过去了——反正问题也不严重。但是对于阻塞相当严重的情况而言，很多人还是会考虑到将这个行为做成异步过程。目前最流行的异步调用分解工具就是mq——不仅异步，而且分布。当然，还有一个更简单的非分布方案——开一个coroutine。

而CPS则是另一个方向——函数的返回值可以不返回调用者，而是返回给第三者。

### IO 过程在什么时间发生

其实这个问题的核心在于——整个回调模型是基于多路复用的还是基于异步IO的？

原则上两者都可以。你可以监听fd就绪，也可以监听IO完成。当然，即使监听IO完成，也不代表使用了内核态异步接口。很可能只是用epoll封装的而已。

### 回调函数的上下文环境

这个问题则需要和上面提到的“用户态调度框架”结合起来说。IO回调注册的实质是将回调函数绑定到某个fd上——就如同将coroutine绑定上去那样。只是coroutine允许你顺序的执行，而callback则会切碎函数。当然，大部分实现中，使用callback也有好处——coroutine的最小切换开销也在50ns，而call本身则只有2ns。

### 状态机模型

状态机模型是一个更难于理解和编程的模型，其本质是每次重入。

想像你是一个周期失忆的病人(就像“一周的朋友”那样)。那么你如何才能完成一项需要跨越周期的工作呢？例如刺绣，种植作物，或者——交一个男朋友。

当然，类比到失忆病人的例子上必须有一点限制。正常的生活技能，还有一些常识性的东西必须不能在周期失忆范围内。例如重新学习认字什么的可没人受的了。

答案就是——做笔记。每次重复失忆后，你需要阅读自己的笔记，观察上次做到哪个步骤，下一个步骤是什么。这需要将一个工作分解为很多步骤，在每个步骤内“重入”直到步骤完成，转移到下一个状态。

同理，在状态机模型解法里，每次执行都需要推演合适的状态，直到工作完成。这个模型已经很少用到了，因为相比回调函数来说，状态机模型更难理解和使用，性能差异也不大。

最后顺带一提，交一个男友的方案和其他几个略有不同，主要靠颜好高冷反差萌，一般人就不要尝试挑战了。。。当然一般人也不会一周失忆一次，毕竟生活不是韩剧也不是日本动漫。。。

## 六．Python中的多进程和多线程

最近在看[Python](http://link.zhihu.com/?target=http%3A//lib.csdn.net/base/11)的多线程，经常我们会听到老手说：**“python下多线程是鸡肋，推荐使用多进程！”**，但是为什么这么说呢？

要知其然，更要知其所以然。所以有了下面的深入研究：

首先强调背景：

1、**GIL是什么？**GIL的全称是Global Interpreter Lock(全局解释器锁)，来源是python设计之初的考虑，为了数据安全所做的决定。

2、每个CPU在同一时间只能执行一个线程（在单核CPU下的多线程其实都只是并发，不是并行，并发和并行从宏观上来讲都是同时处理多路请求的概念。但并发和并行又有区别，并行是指两个或者多个事件在同一时刻发生；而并发是指两个或多个事件在同一时间间隔内发生。）

**在Python多线程下，每个线程的执行方式：**  
1.获取GIL  
2.执行代码直到sleep或者是python虚拟机将其挂起。  
3.释放GIL

**可见，某个线程想要执行，必须先拿到GIL，我们可以把GIL看作是“通行证”，并且在一个python进程中，GIL只有一个。拿不到通行证的线程，就不允许进入CPU执行。**

在python2.x里，GIL的释放逻辑是当前线程遇见IO操作或者ticks计数达到100（ticks可以看作是python自身的一个计数器，专门做用于GIL，每次释放后归零，这个计数可以通过 sys.setcheckinterval 来调整），进行释放。

而每次释放GIL锁，线程进行锁竞争、切换线程，会消耗资源。并且由于GIL锁存在，python里一个进程永远只能同时执行一个线程(拿到GIL的线程才能执行)，这就是为什么在多核CPU上，python的多线程效率并不高。

**那么是不是python的多线程就完全没用了呢？**

在这里我们进行分类讨论：

1、CPU密集型代码(各种循环处理、计数等等)，在这种情况下，ticks计数很快就会达到阈值，然后触发GIL的释放与再竞争（多个线程来回切换当然是需要消耗资源的），所以**python下的多线程对CPU密集型代码并不友好。**

2、IO密集型代码(文件处理、网络爬虫等)，多线程能够有效提升效率(单线程下有IO操作会进行IO等待，造成不必要的时间浪费，而开启多线程能在线程A等待时，自动切换到线程B，可以不浪费CPU的资源，从而能提升程序执行效率)。**所以python的多线程对IO密集型代码比较友好。**

而在python3.x中，GIL不使用ticks计数，改为使用计时器（执行时间达到阈值后，当前线程释放GIL），这样对CPU密集型程序更加友好，**但依然没有解决GIL导致的同一时间只能执行一个线程的问题，所以效率依然不尽如人意。**

**多核多线程比单核多线程更差，原因是单核下多线程，每次释放GIL，唤醒的那个线程都能获取到GIL锁，所以能够无缝执行，但多核下，CPU0释放GIL后，其他CPU上的线程都会进行竞争，但GIL可能会马上又被CPU0拿到，导致其他几个CPU上被唤醒后的线程会醒着等待到切换时间后又进入待调度状态，这样会造成线程颠簸(thrashing)，导致效率更低**

回到最开始的问题：经常我们会听到老手说：**“python下想要充分利用多核CPU，就用多进程”**，原因是什么呢？

原因是：每个进程有各自独立的GIL，互不干扰，这样就可以真正意义上的并行执行，所以在python中，多进程的执行效率优于多线程(仅仅针对多核CPU而言)。

**所以我们能够得出结论：多核下，想做并行提升效率，比较通用的方法是使用多进程，能够有效提高执行效率**

## 七．Python中的协程

#### 简介

协程，又称微线程，纤程。英文名Coroutine。

协程的概念很早就提出来了，但直到最近几年才在某些语言（如Lua）中得到广泛应用。

子程序，或者称为函数，在所有语言中都是层级调用，比如A调用B，B在执行过程中又调用了C，C执行完毕返回，B执行完毕返回，最后是A执行完毕。所以子程序调用是通过栈实现的，一个线程就是执行一个子程序。

子程序调用总是一个入口，一次返回，调用顺序是明确的。而协程的调用和子程序不同。

协程看上去也是子程序，但执行过程中，在子程序内部可中断，然后转而执行别的子程序，在适当的时候再返回来接着执行。

注意，在一个子程序中中断，去执行其他子程序，不是函数调用，有点类似CPU的中断。比如子程序A、B：

def A():

print '1'

print '2'

print '3'

def B():

print 'x'

print 'y'

print 'z'

假设由协程执行，在执行A的过程中，可以随时中断，去执行B，B也可能在执行过程中中断再去执行A，结果可能是：

1

2

x

y

3

z

但是在A中是没有调用B的，所以协程的调用比函数调用理解起来要难一些。看起来A、B的执行有点像多线程，但协程的特点在于是一个线程执行，那和多线程比，协程有何优势？

最大的优势就是协程极高的执行效率。因为子程序切换不是线程切换，而是由程序自身控制，因此，没有线程切换的开销，和多线程比，线程数量越多，协程的性能优势就越明显。

第二大优势就是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。

因为协程是一个线程执行，那怎么利用多核CPU呢？最简单的方法是多进程+协程，既充分利用多核，又充分发挥协程的高效率，可获得极高的性能。

Python对协程的支持还非常有限，用在generator中的yield可以一定程度上实现协程。虽然支持不完全，但已经可以发挥相当大的威力了。

来看例子：

传统的生产者-消费者模型是一个线程写消息，一个线程取消息，通过锁机制控制队列和等待，但一不小心就可能死锁。

如果改用协程，生产者生产消息后，直接通过yield跳转到消费者开始执行，待消费者执行完毕后，切换回生产者继续生产，效率极高：

import time

def consumer():

r = ''

while True:

n = yield r

if not n:

return

print('[CONSUMER] Consuming %s...' % n)

time.sleep(1)

r = '200 OK'

def produce(c):

c.next()

n = 0

while n < 5:

n = n + 1

print('[PRODUCER] Producing %s...' % n)

r = c.send(n)

print('[PRODUCER] Consumer return: %s' % r)

c.close()

if \_\_name\_\_=='\_\_main\_\_':

c = consumer()

produce(c)

执行结果：

[PRODUCER] Producing 1...

[CONSUMER] Consuming 1...

[PRODUCER] Consumer return: 200 OK

[PRODUCER] Producing 2...

[CONSUMER] Consuming 2...

[PRODUCER] Consumer return: 200 OK

[PRODUCER] Producing 3...

[CONSUMER] Consuming 3...

[PRODUCER] Consumer return: 200 OK

[PRODUCER] Producing 4...

[CONSUMER] Consuming 4...

[PRODUCER] Consumer return: 200 OK

[PRODUCER] Producing 5...

[CONSUMER] Consuming 5...

[PRODUCER] Consumer return: 200 OK

注意到consumer函数是一个generator（生成器），把一个consumer传入produce后：

1. 首先调用c.next()启动生成器；
2. 然后，一旦生产了东西，通过c.send(n)切换到consumer执行；
3. consumer通过yield拿到消息，处理，又通过yield把结果传回；
4. produce拿到consumer处理的结果，继续生产下一条消息；
5. produce决定不生产了，通过c.close()关闭consumer，整个过程结束。

整个流程无锁，由一个线程执行，produce和consumer协作完成任务，所以称为“协程”，而非线程的抢占式多任务。

最后套用Donald Knuth的一句话总结协程的特点：“子程序就是协程的一种特例。”

#### Gevent

Python通过yield提供了对协程的基本支持，但是不完全。而第三方的gevent为Python提供了比较完善的协程支持。

gevent是第三方库，通过greenlet实现协程，其基本思想是：

当一个greenlet遇到IO操作时，比如访问网络，就自动切换到其他的greenlet，等到IO操作完成，再在适当的时候切换回来继续执行。由于IO操作非常耗时，经常使程序处于等待状态，有了gevent为我们自动切换协程，就保证总有greenlet在运行，而不是等待IO。

由于切换是在IO操作时自动完成，所以gevent需要修改Python自带的一些标准库，这一过程在启动时通过monkey patch完成：

**from** gevent **import** monkey; monkey.patch\_socket()

**import** gevent

**def** **f**(n):

**for** i **in** range(n):

**print** gevent.getcurrent(), i

g1 = gevent.spawn(f, 5)

g2 = gevent.spawn(f, 5)

g3 = gevent.spawn(f, 5)

g1.join()

g2.join()

g3.join()

运行结果：

<Greenlet at 0x10e49f550: f(5)> 0

<Greenlet at 0x10e49f550: f(5)> 1

<Greenlet at 0x10e49f550: f(5)> 2

<Greenlet at 0x10e49f550: f(5)> 3

<Greenlet at 0x10e49f550: f(5)> 4

<Greenlet at 0x10e49f910: f(5)> 0

<Greenlet at 0x10e49f910: f(5)> 1

<Greenlet at 0x10e49f910: f(5)> 2

<Greenlet at 0x10e49f910: f(5)> 3

<Greenlet at 0x10e49f910: f(5)> 4

<Greenlet at 0x10e49f4b0: f(5)> 0

<Greenlet at 0x10e49f4b0: f(5)> 1

<Greenlet at 0x10e49f4b0: f(5)> 2

<Greenlet at 0x10e49f4b0: f(5)> 3

<Greenlet at 0x10e49f4b0: f(5)> 4

可以看到，3个greenlet是依次运行而不是交替运行。

要让greenlet交替运行，可以通过gevent.sleep()交出控制权：

**def** **f**(n):

**for** i **in** range(n):

**print** gevent.getcurrent(), i

gevent.sleep(0)

执行结果：

<Greenlet at 0x10cd58550: f(5)> 0

<Greenlet at 0x10cd58910: f(5)> 0

<Greenlet at 0x10cd584b0: f(5)> 0

<Greenlet at 0x10cd58550: f(5)> 1

<Greenlet at 0x10cd584b0: f(5)> 1

<Greenlet at 0x10cd58910: f(5)> 1

<Greenlet at 0x10cd58550: f(5)> 2

<Greenlet at 0x10cd58910: f(5)> 2

<Greenlet at 0x10cd584b0: f(5)> 2

<Greenlet at 0x10cd58550: f(5)> 3

<Greenlet at 0x10cd584b0: f(5)> 3

<Greenlet at 0x10cd58910: f(5)> 3

<Greenlet at 0x10cd58550: f(5)> 4

<Greenlet at 0x10cd58910: f(5)> 4

<Greenlet at 0x10cd584b0: f(5)> 4

3个greenlet交替运行，

把循环次数改为500000，让它们的运行时间长一点，然后在操作系统的进程管理器中看，线程数只有1个。

当然，实际代码里，我们不会用gevent.sleep()去切换协程，而是在执行到IO操作时，gevent自动切换，代码如下：

**from** gevent **import** monkey; monkey.patch\_all()

**import** gevent

**import** urllib2

**def** **f**(url):

print('GET: %s' % url)

resp = urllib2.urlopen(url)

data = resp.read()

print('%d bytes received from %s.' % (len(data), url))

gevent.joinall([

gevent.spawn(f, 'https://www.python.org/'),

gevent.spawn(f, 'https://www.yahoo.com/'),

gevent.spawn(f, 'https://github.com/'),

])

运行结果：

GET: https://www.python.org/

GET: https://www.yahoo.com/

GET: https://github.com/

45661 bytes received from https://www.python.org/.

14823 bytes received from https://github.com/.

304034 bytes received from https://www.yahoo.com/.

从结果看，3个网络操作是并发执行的，而且结束顺序不同，但只有一个线程。

**小结**

使用gevent，可以获得极高的并发性能，但gevent只能在Unix/Linux下运行，在Windows下不保证正常安装和运行。

由于gevent是基于IO切换的协程，所以最神奇的是，我们编写的Web App代码，不需要引入gevent的包，也不需要改任何代码，仅仅在部署的时候，用一个支持gevent的WSGI服务器，立刻就获得了数倍的性能提升。具体部署方式可以参考后续“实战”-“部署Web App”一节。