**鱼还是熊掌：浅谈多进程多线程的选择**

关于多进程和多线程，教科书上最经典的一句话是“进程是资源分配的最小单位，线程是CPU调度的最小单位”，这句话应付考试基本上够了，但如果在工作中遇到类似的选择问题，那就没有这么简单了，选的不好，会让你深受其害。

经常在网络上看到有的XDJM问“多进程好还是多线程好？”、“[**Linux**](http://lib.csdn.net/base/linux)下用多进程还是多线程？”等等期望一劳永逸的问题，我只能说：没有最好，只有更好。根据实际情况来判断，哪个更加合适就是哪个好。

我们按照多个不同的维度，来看看多线程和多进程的对比（注：因为是感性的比较，因此都是相对的，不是说一个好得不得了，另外一个差的无法忍受）。

|  |  |  |  |
| --- | --- | --- | --- |
| **对比维度** | **多进程** | **多线程** | **总结** |
| 数据共享、同步 | 数据共享复杂，需要用IPC；数据是分开的，同步简单 | 因为共享进程数据，数据共享简单，但也是因为这个原因导致同步复杂 | 各有优势 |
| 内存、CPU | 占用内存多，切换复杂，CPU利用率低 | 占用内存少，切换简单，CPU利用率高 | 线程占优 |
| 创建销毁、切换 | 创建销毁、切换复杂，速度慢 | 创建销毁、切换简单，速度很快 | 线程占优 |
| 编程、调试 | 编程简单，调试简单 | 编程复杂，调试复杂 | 进程占优 |
| 可靠性 | 进程间不会互相影响 | 一个线程挂掉将导致整个进程挂掉 | 进程占优 |
| 分布式 | 适应于多核、多机分布式；如果一台机器不够，扩展到多台机器比较简单 | 适应于多核分布式 | 进程占优 |

看起来比较简单，优势对比上是“线程 3.5 v 2.5 进程”，我们只管选线程就是了？

呵呵，有这么简单我就不用在这里浪费口舌了，还是那句话，没有绝对的好与坏，只有哪个更加合适的问题。我们来看实际应用中究竟如何判断更加合适。

**1）需要频繁创建销毁的优先用线程**

原因请看上面的对比。

这种原则最常见的应用就是Web服务器了，来一个连接建立一个线程，断了就销毁线程，要是用进程，创建和销毁的代价是很难承受的

**2）需要进行大量计算的优先使用线程**

所谓大量计算，当然就是要耗费很多CPU，切换频繁了，这种情况下线程是最合适的。

这种原则最常见的是图像处理、[**算法**](http://lib.csdn.net/base/datastructure)处理。

**3）强相关的处理用线程，弱相关的处理用进程**

什么叫强相关、弱相关？理论上很难定义，给个简单的例子就明白了。

一般的Server需要完成如下任务：消息收发、消息处理。“消息收发”和“消息处理”就是弱相关的任务，而“消息处理”里面可能又分为“消息解码”、“业务处理”，这两个任务相对来说相关性就要强多了。因此“消息收发”和“消息处理”可以分进程设计，“消息解码”、“业务处理”可以分线程设计。

当然这种划分方式不是一成不变的，也可以根据实际情况进行调整。

**4）可能要扩展到多机分布的用进程，多核分布的用线程**

原因请看上面对比。

**5）都满足需求的情况下，用你最熟悉、最拿手的方式**

至于“数据共享、同步”、“编程、调试”、“可靠性”这几个维度的所谓的“复杂、简单”应该怎么取舍，我只能说：没有明确的选择方法。但我可以告诉你一个选择原则：如果多进程和多线程都能够满足要求，那么选择你最熟悉、最拿手的那个。

需要提醒的是：虽然我给了这么多的选择原则，但实际应用中基本上都是“进程+线程”的结合方式，千万不要真的陷入一种非此即彼的误区。

**1、进程与线程**

进程是程序执行时的一个实例，即它是程序已经执行到课中程度的[**数据结构**](http://lib.csdn.net/base/datastructure)的汇集。从内核的观点看，进程的目的就是担当分配系统资源（CPU时间、内存等）的基本单位。

线程是进程的一个执行流，是CPU调度和分派的基本单位，它是比进程更小的能独立运行的基本单位。一个进程由几个线程组成（拥有很多相对独立的执行流的用户程序共享应用程序的大部分数据结构），线程与同属一个进程的其他的线程共享进程所拥有的全部资源。

"进程——资源分配的最小单位，线程——程序执行的最小单位"

进程有独立的地址空间，一个进程崩溃后，在保护模式下不会对其它进程产生影响，而线程只是一个进程中的不同执行路径。线程有自己的堆栈和局部变量，但线程没有单独的地址空间，一个线程死掉就等于整个进程死掉，所以多进程的程序要比多线程的程序健壮，但在进程切换时，耗费资源较大，效率要差一些。但对于一些要求同时进行并且又要共享某些变量的并发操作，只能用线程，不能用进程。

总的来说就是：进程有独立的地址空间，线程没有单独的地址空间（同一进程内的线程共享进程的地址空间）。（下面的内容摘自[Linux下的多线程编程](http://fanqiang.chinaunix.net/a4/b8/20010811/0905001105.html)）

使用多线程的**理由之一**是和进程相比，它是一种非常"节俭"的多任务操作方式。我们知道，在[**linux**](http://lib.csdn.net/base/linux)系统下，启动一个新的进程必须分配给它独立的地址空间，建立众多的数据表来维护它的代码段、堆栈段和数据段，这是一种"昂贵"的多任务工作方式。而运行于一个进程中的多个线程，它们彼此之间使用相同的地址空间，共享大部分数据，启动一个线程所花费的空间远远小于启动一个进程所花费的空间，而且，线程间彼此切换所需的时间也远远小于进程间切换所需要的时间。据统计，总的说来，一个进程的开销大约是一个线程开销的30倍左右，当然，在具体的系统上，这个数据可能会有较大的区别。

使用多线程的**理由之二**是线程间方便的通信机制。对不同进程来说，它们具有独立的数据空间，要进行数据的传递只能通过通信的方式进行，这种方式不仅费时，而且很不方便。线程则不然，由于同一进程下的线程之间共享数据空间，所以一个线程的数据可以直接为其它线程所用，这不仅快捷，而且方便。当然，数据的共享也带来其他一些问题，有的变量不能同时被两个线程所修改，有的子程序中声明为static的数据更有可能给多线程程序带来灾难性的打击，这些正是编写多线程程序时最需要注意的地方。

除了以上所说的优点外，不和进程比较，多线程程序作为一种多任务、并发的工作方式，当然有以下的优点：

* 提高应用程序响应。这对图形界面的程序尤其有意义，当一个操作耗时很长时，整个系统都会等待这个操作，此时程序不会响应键盘、鼠标、菜单的操作，而使用多线程技术，将耗时长的操作（time consuming）置于一个新的线程，可以避免这种尴尬的情况。
* 使多CPU系统更加有效。操作系统会保证当线程数不大于CPU数目时，不同的线程运行于不同的CPU上。
* 改善程序结构。一个既长又复杂的进程可以考虑分为多个线程，成为几个独立或半独立的运行部分，这样的程序会利于理解和修改。

在Unix上编程采用多线程还是多进程的争执由来已久，这种争执最常见到在B/S通讯中服务端并发技术 的选型上，比如WEB服务器技术中，Apache是采用多进程的（perfork模式，每客户连接对应一个进程，每进程中只存在唯一一个执行线 程），[**Java**](http://lib.csdn.net/base/java)的Web容器Tomcat、Websphere等都是多线程的（每客户连接对应一个线程，所有线程都在一个进程中）。

从Unix发展历史看，伴随着Unix的诞生多进程就出现了，而多线程很晚才被系统支持，例如Linux直到内核2.6，才支持符合Posix规范的NPTL线程库。进程和线程的特点，也就是各自的优缺点如下：

进程优点：编程、调试简单，可靠性较高。  
进程缺点：创建、销毁、切换速度慢，内存、资源占用大。  
线程优点：创建、销毁、切换速度快，内存、资源占用小。  
线程缺点：编程、调试复杂，可靠性较差。

上面的对比可以归结为一句话：“线程快而进程可靠性高”。线程有个别名叫“轻量级进程”，在有的书籍资料上介绍线程可以十倍、百倍的效率快于进程； 而进程之间不共享数据，没有锁问题，结构简单，一个进程崩溃不像线程那样影响全局，因此比较可靠。我相信这个观点可以被大部分人所接受，因为和我们所接受的知识概念是相符的。

在写这篇文章前，我也属于这“大部分人”，这两年在用[**C语言**](http://lib.csdn.net/base/c)编写的几个C/S通讯程序中，因时间紧总是采用多进程并发技术，而且是比较简单的现场为 每客户fork()一个进程，当时总是担心并发量增大时负荷能否承受，盘算着等时间充裕了将它改为多线程形式，或者改为预先创建进程的形式，直到最近在网 上看到了一篇论文《Linux系统下多线程与多进程性能分析》作者“周丽 焦程波 兰巨龙”，才认真思考这个问题，我自己也做了实验，结论和论文作者的相似，但对大部分人可以说是颠覆性的。

下面是得出结论的实验步骤和过程，结论究竟是怎样的？ 感兴趣就一起看看吧。

实验代码使用周丽论文中的代码样例，我做了少量修改，值得注意的是这样的区别：

论文实验和我的实验时间不同，论文所处的年代linux内核是2.4，我的实验linux内核是2.6，2.6使用的线程库是NPTL，2.4使用的是老的Linux线程库（用进程模拟线程的那个LinuxThread）。

论文实验和我用的机器不同，论文描述了使用的环境：单cpu 机器基本配置为:celeron 2.0 GZ, 256M, Linux 9.2,内核 2.4.8。我的环境是：双核 Intel(R) Xeon(R) CPU 5130  @ 2.00GHz（做实验时，禁掉了一核），512MG内存，Red Hat Enterprise Linux ES release 4 (Nahant Update 4)，内核2.6.9-42。

进程实验代码（fork.c）：

1. #include <stdlib.h>
2. #include <stdio.h>
3. #include <signal.h>
4. #define P\_NUMBER 255 //并发进程数量
5. #define COUNT 5 //每次进程打印字符串数
6. #define TEST\_LOGFILE "logFile.log"
7. FILE \*logFile=NULL;
8. char \*s="hello linux\0";
9. int main()
10. {
11. int i=0,j=0;
12. logFile=fopen(TEST\_LOGFILE,"a+");//打开日志文件
13. for(i=0;i<P\_NUMBER;i++)
14. {
15. if(fork()==0)//创建子进程，if(fork()==0){}这段代码是子进程运行区间
16. {
17. for(j=0;j<COUNT;j++)
18. {
19. printf("[%d]%s\n",j,s);//向控制台输出
20. /\*当你频繁读写文件的时候，Linux内核为了提高读写性能与速度，会将文件在内存中进行缓存，这部分内存就是Cache Memory(缓存内存)。可能导致测试结果不准，所以在此注释\*/
21. //fprintf(logFile,"[%d]%s\n",j,s);//向日志文件输出，
22. }
23. exit(0);//子进程结束
24. }
25. }
27. for(i=0;i<P\_NUMBER;i++)//回收子进程
28. {
29. wait(0);
30. }
32. printf("Okay\n");
33. return 0;
34. }

进程实验代码（thread.c）：

1. #include <pthread.h>
2. #include <unistd.h>
3. #include <stdlib.h>
4. #include <stdio.h>
5. #define P\_NUMBER 255//并发线程数量
6. #define COUNT 5 //每线程打印字符串数
7. #define TEST\_LOG "logFile.log"
8. FILE \*logFile=NULL;
9. char \*s="hello linux\0";
10. print\_hello\_linux()//线程执行的函数
11. {
12. int i=0;
13. for(i=0;i<COUNT;i++)
14. {
15. printf("[%d]%s\n",i,s);//想控制台输出
16. /\*当你频繁读写文件的时候，Linux内核为了提高读写性能与速度，会将文件在内存中进行缓存，这部分内存就是Cache Memory(缓存内存)。可能导致测试结果不准，所以在此注释\*/
17. //fprintf(logFile,"[%d]%s\n",i,s);//向日志文件输出
18. }
19. pthread\_exit(0);//线程结束
20. }
21. int main()
22. {
23. int i=0;
24. pthread\_t pid[P\_NUMBER];//线程数组
25. logFile=fopen(TEST\_LOG,"a+");//打开日志文件
27. for(i=0;i<P\_NUMBER;i++)
28. pthread\_create(&pid[i],NULL,(void \*)print\_hello\_linux,NULL);//创建线程
30. for(i=0;i<P\_NUMBER;i++)
31. pthread\_join(pid[i],NULL);//回收线程
33. printf("Okay\n");
34. return 0;
35. }

两段程序做的事情是一样的，都是创建“若干”个进程/线程，每个创建出的进程/线程打印“若干”条“hello linux”字符串到控制台和日志文件，两个“若干”由两个宏 P\_NUMBER和COUNT分别定义，程序编译指令如下：

gcc -o fork fork.c  
gcc -lpthread -o thread thread.c

实验通过time指令执行两个程序，抄录time输出的挂钟时间（real时间）：

time ./fork  
time ./thread

每批次的实验通过改动宏 P\_NUMBER和COUNT来调整进程/线程数量和打印次数，每批次[**测试**](http://lib.csdn.net/base/softwaretest)五轮，得到的结果如下：

**一、重复周丽论文实验步骤**

**(注：本文平均值算法采用的是去掉一个最大值去掉一个最小值，然后平均)**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 单核（双核机器禁掉一核），进程/线程数：255，打印次数5 | | | | | | |
|  | 第1次 | 第2次 | 第3次 | 第4次 | 第5次 | 平均 |
| 多进程 | 0m0.070s | 0m0.071s | 0m0.071s | 0m0.070s | 0m0.070s | 0m0.070s |
| 多线程 | 0m0.049s | 0m0.049s | 0m0.049s | 0m0.049s | 0m0.049s | 0m0.049s |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 单核（双核机器禁掉一核），进程/线程数：255，打印次数10 | | | | | | |
|  | 第1次 | 第2次 | 第3次 | 第4次 | 第5次 | 平均 |
| 多进程 | 0m0.112s | 0m0.101s | 0m0.100s | 0m0.085s | 0m0.121s | 0m0.104s |
| 多线程 | 0m0.097s | 0m0.089s | 0m0.090s | 0m0.104s | 0m0.080s | 0m0.092s |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 单核（双核机器禁掉一核），进程/线程数：255，打印次数50 | | | | | | |
|  | 第1次 | 第2次 | 第3次 | 第4次 | 第5次 | 平均 |
| 多进程 | 0m0.459s | 0m0.531s | 0m0.499s | 0m0.499s | 0m0.524s | 0m0.507s |
| 多线程 | 0m0.387s | 0m0.456s | 0m0.435s | 0m0.423s | 0m0.408s | 0m0.422s |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 单核（双核机器禁掉一核），进程/线程数：255，打印次数100 | | | | | | |
|  | 第1次 | 第2次 | 第3次 | 第4次 | 第5次 | 平均 |
| 多进程 | 0m1.141s | 0m0.992s | 0m1.134s | 0m1.027s | 0m0.965s | 0m1.051s |
| 多线程 | 0m0.925s | 0m0.899s | 0m0.961s | 0m0.934s | 0m0.853s | 0m0.919s |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 单核（双核机器禁掉一核），进程/线程数：255，打印次数500 | | | | | | |
|  | 第1次 | 第2次 | 第3次 | 第4次 | 第5次 | 平均 |
| 多进程 | 0m5.221s | 0m5.258s | 0m5.706s | 0m5.288s | 0m5.455s | 0m5.334s |
| 多线程 | 0m4.689s | 0m4.578s | 0m4.670s | 0m4.566s | 0m4.722s | 0m4.646s |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 单核（双核机器禁掉一核），进程/线程数：255，打印次数1000 | | | | | | |
|  | 第1次 | 第2次 | 第3次 | 第4次 | 第5次 | 平均 |
| 多进程 | 0m12.680s | 0m16.555s | 0m11.158s | 0m10.922s | 0m11.206s | 0m11.681s |
| 多线程 | 0m12.993s | 0m13.087s | 0m13.082s | 0m13.485s | 0m13.053s | 0m13.074s |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 单核（双核机器禁掉一核），进程/线程数：255，打印次数5000 | | | | | | |
|  | 第1次 | 第2次 | 第3次 | 第4次 | 第5次 | 平均 |
| 多进程 | 1m27.348s | 1m5.569s | 0m57.275s | 1m5.029s | 1m15.174s | 1m8.591s |
| 多线程 | 1m25.813s | 1m29.299s | 1m23.842s | 1m18.914s | 1m34.872s | 1m26.318s |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 单核（双核机器禁掉一核），进程/线程数：255，打印次数10000 | | | | | | |
|  | 第1次 | 第2次 | 第3次 | 第4次 | 第5次 | 平均 |
| 多进程 | 2m8.336s | 2m22.999s | 2m11.046s | 2m30.040s | 2m5.752s | 2m14.137s |
| 多线程 | 2m46.666s | 2m44.757s | 2m34.528s | 2m15.018s | 2m41.436s | 2m40.240s |

本轮实验是为了和周丽论文作对比，因此将进程/线程数量限制在255个，论文也是测试了255个进程/线程分别进行5次，10 次,50 次,100 次,500 次……10000 次打印的用时，论文得出的结果是：**任务量较大时,多进程比多线程效率高;而完成的任务量较小时,多线程比多进程要快，重复打印 600 次时,多进程与多线程所耗费的时间相同。**

虽然我的实验直到1000打印次数时，多进程才开始领先，但考虑到使用的是NPTL线程库的缘故，从而可以证实了论文的观点。从我的实验数据看，多线程和多进程两组数据非常接近，考虑到数据的提取具有瞬间性，因此可以认为他们的速度是相同的。

是不是可以得出这样的结论：**多线程创建、销毁速度快，而多线程切换速度快，**这个结论我们会在第二个试验中继续试图验证

当前的网络环境中，我们更看中高并发、高负荷下的性能，纵观前面的实验步骤，最长的实验周期不过2分钟多一点，因此下面的实验将向两个方向延伸，第一，增加并发数量，第二，增加每进程/线程的工作强度。

**二、增加并发数量的实验**

下面的实验打印次数不变，而进程/线程数量逐渐增加。在实验过程中多线程程序在后四组（线程数350，500，800,1000）的测试中都出现了“段错误”，出现错误的原因和多线程预分配线程栈有关。

实验中的计算机CPU是32位，寻址最大范围是4GB（2的32次方），Linux是按照3GB/1GB的方式来分配内存，其中1GB属于所有进程共享的内核空间，3GB属于用户空间（进程虚拟内存空间）。Linux2.6的默认线程栈大小是8M（通过ulimit -a查看），对于多线程，在创建线程的时候系统会为每一个线程预分配线程栈地址空间，也就是8M的虚拟内存空间。线程数量太多时，线程栈累计的大小将超过进程虚拟内存空间大小（计算时需要排除程序文本、数据、共享库等占用的空间），这就是实验中出现的“段错误”的原因。

Linux2.6的默认线程栈大小可以通过 ulimit -s 命令查看或修改，我们可以计算出线程数的最大上线: (1024\*1024\*1024\*3) / (1024\*1024\*8) = 384，实际数字应该略小与384，因为还要计算程序文本、数据、共享库等占用的空间。在当今的稍显繁忙的WEB服务器上，突破384的并发访问并不是稀 罕的事情，要继续下面的实验需要将默认线程栈的大小减小，但这样做有一定的风险，比如线程中的函数分配了大量的自动变量或者函数涉及很深的栈帧（典型的是 递归调用），线程栈就可能不够用了。可以配合使用POSIX.1规定的两个线程属性guardsize和stackaddr来解决线程栈溢出问 题，guardsize控制着线程栈末尾之后的一篇内存区域，一旦线程栈在使用中溢出并到达了这片内存，程序可以捕获系统内核发出的告警信号，然后使用 malloc获取另外的内存，并通过stackaddr改变线程栈的位置，以获得额外的栈空间，这个动态扩展栈空间办法需要手工编程，而且非常麻烦。

有两种方法可以改变线程栈的大小，使用 ulimit -s 命令改变系统默认线程栈的大小，或者在代码中创建线程时通过pthread\_attr\_setstacksize函数改变栈尺寸，在实验中使用的是第一种，在程序运行前先执行ulimit指令将默认线程栈大小改为1M：

ulimit -s 1024  
time ./thread

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 单核（双核机器禁掉一核），进程/线程数：100 ，打印次数1000 | | | | | | |
|  | 第1次 | 第2次 | 第3次 | 第4次 | 第5次 | 平均 |
| 多进程 | 0m3.834s | 0m3.759s | 0m4.376s | 0m3.936s | 0m3.851s | 0m3.874 |
| 多线程 | 0m3.646s | 0m4.498s | 0m4.219s | 0m3.893s | 0m3.943s | 0m4.018 |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 单核（双核机器禁掉一核），进程/线程数：255 ，打印次数1000 | | | | | | |
|  | 第1次 | 第2次 | 第3次 | 第4次 | 第5次 | 平均 |
| 多进程 | 0m9.731s | 0m9.833s | 0m10.046s | 0m9.830s | 0m9.866s | 0m9.843s |
| 多线程 | 0m9.961s | 0m9.699s | 0m9.958s | 0m10.111s | 0m9.686s | 0m9.873s |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 单核（双核机器禁掉一核），进程/线程数：350  ，打印次数1000 | | | | | | |
|  | 第1次 | 第2次 | 第3次 | 第4次 | 第5次 | 平均 |
| 多进程 | 0m13.773s | 0m13.500s | 0m13.519s | 0m13.474s | 0m13.351s | 0m13.498 |
| 多线程 | 0m12.754s | 0m13.251s | 0m12.813s | 0m16.861s | 0m12.764s | 0m12.943 |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 单核（双核机器禁掉一核），进程/线程数： 500 ，打印次数1000 | | | | | | |
|  | 第1次 | 第2次 | 第3次 | 第4次 | 第5次 | 平均 |
| 多进程 | 0m23.762s | 0m22.151s | 0m23.926s | 0m21.327s | 0m21.429s | 0m22.413 |
| 多线程 | 0m20.603s | 0m20.291s | 0m21.654s | 0m20.684s | 0m20.671s | 0m20.653 |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 单核（双核机器禁掉一核），进程/线程数：800  ，打印次数1000 | | | | | | |
|  | 第1次 | 第2次 | 第3次 | 第4次 | 第5次 | 平均 |
| 多进程 | 0m33.616s | 0m31.757s | 0m31.759s | 0m32.232s | 0m32.498s | 0m32.163 |
| 多线程 | 0m32.050s | 0m32.787s | 0m33.055s | 0m32.902s | 0m32.235s | 0m32.641 |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 单核（双核机器禁掉一核），进程/线程数： 1000 ，打印次数1000 | | | | | | |
|  | 第1次 | 第2次 | 第3次 | 第4次 | 第5次 | 平均 |
| 多进程 | 0m40.301s | 0m41.083s | 0m41.634s | 0m40.247s | 0m40.717s | 0m40.700 |
| 多线程 | 0m41.633s | 0m41.118s | 0m42.700s | 0m42.134s | 0m41.170s | 0m41.646 |

【实验结论】   
当线程/进程逐渐增多时，执行相同任务时，线程所花费时间相对于进程有下降的趋势（本人怀疑后两组数据受系统其他瓶颈的影响），这是不是进一步验证了多线程创建、销毁速度快，而多进程切换速度快。  
  
出现了线程栈的问题，让我特别关心Java线程是怎样处理的，因此用Java语言写了同样的实验程序，Java程序加载虚拟机环境比较耗时，所以没 有用time提取测试时间，而直接将测时写入代码。对Linux上的C编程不熟悉的Java程序员也可以用这个程序去对比理解上面的C语言试验程序。

1. import java.io.File;
2. import java.io.FileNotFoundException;
3. import java.io.FileOutputStream;
4. import java.io.IOException;
6. public class MyThread extends Thread
7. {
8. static int P\_NUMBER = 1000; /\* 并发线程数量 \*/
9. static int COUNT = 1000; /\* 每线程打印字符串次数 \*/
11. static String s = "hello linux\n";
13. static FileOutputStream out = null; /\* 文件输出流 \*/
14. @Override
15. public void run()
16. {
17. for (int i = 0; i < COUNT; i++)
18. {
19. System.out.printf("[%d]%s", i, s); /\* 向控制台输出 \*/
21. StringBuilder sb = new StringBuilder(16);
22. sb.append("[").append(i).append("]").append(s);
23. try
24. {
25. out.write(sb.toString().getBytes());/\* 向日志文件输出 \*/
26. }
27. catch (IOException e)
28. {
29. e.printStackTrace();
30. }
31. }
32. }
34. public static void main(String[] args) throws FileNotFoundException, InterruptedException
35. {
36. MyThread[] threads = new MyThread[P\_NUMBER]; /\* 线程数组 \*/
38. File file = new File("Javalogfile.log");
39. out = new FileOutputStream(file, true); /\* 日志文件输出流 \*/
41. System.out.println("开始运行");
42. long start = System.currentTimeMillis();
44. for (int i = 0; i < P\_NUMBER; i++) //创建线程
45. {
46. threads[i] = new MyThread();
47. threads[i].start();
48. }
50. for (int i = 0; i < P\_NUMBER; i++) //回收线程
51. {
52. threads[i].join();
53. }
55. System.out.println("用时：" + (System.currentTimeMillis() – start) + " 毫秒");
56. return;
57. }
59. }

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 进程/线程数：1000  ，打印次数1000(用得原作者的数据) | | | | | | |
|  | 第1次 | 第2次 | 第3次 | 第4次 | 第5次 | 平均 |
| 多线程 | 65664 ms | 66269 ms | 65546ms | 65931ms | 66540ms | 65990 ms |

Java程序比C程序慢一些在情理之中，但Java程序并没有出现线程栈问题，5次测试都平稳完成，可以用下面的ps指令获得java进程中线程的数量：

diaoyf@ali:~$ ps -eLf | grep MyThread | wc -l  
1010

用ps测试线程数在1010上维持了很长时间，多出的10个线程应该是jvm内部的管理线程，比如用于GC。我不知道Java创建线程时默认栈的大 小是多少，很多资料说法不统一，于是下载了Java的源码jdk-6u21-fcs-src-b07-jrl-17\_jul\_2010.jar（实验环境 安装的是 SUN jdk 1.6.0\_20-b02），但没能从中找到需要的信息。对于jvm的运行，java提供了控制参数，因此再次测试时，通过下面的参数将Java线程栈大 小定义在8192k，和Linux的默认大小一致：

diaoyf@ali:~/tmp1$ java -Xss8192k MyThread

出乎意料的是并没有出现想象中的异常，但用ps侦测线程数最高到达337，我判断程序在创建线程时在栈到达可用内存的上线时就停止继续创建了，程序运行的时间远小于估计值也证明了这个判断。程序虽然没有抛出异常，但运行的并不正常，另一个问题是最后并没有打印出“用时 xxx毫秒”信息。

这次测试更加深了我的一个长期的猜测：Java的Web容器不稳定。因为我是多年编写B/S的Java程序员，WEB服务不稳定常常挂掉也是司空见惯的，除了自己或项目组成员水平不高，代码编写太烂的原因之外，我一直猜测还有更深层的原因，如果就是线程原因的话，这颠覆性可比本篇文章的多进程性能颠覆性要大得多，想想世界上有多少Tomcat、Jboss、Websphere、weblogic在跑着，嘿嘿。

这次测试还打破了以前的一个说法：单CPU上并发超过6、7百，线程或进程间的切换就会占用大量CPU时间，造成服务器效率会急剧下降。但从上面的实验来看，进程/线程数到1000时（这差不多是非常繁忙的WEB服务器了），仍具有很好的线性。

**三、增加每进程/线程的工作强度的实验**

这次将程序打印数据增大，原来打印字符串为：

1. char \*s = "hello linux\0";

现在修改为每次打印256个字节数据:

1. char \*s = "1234567890abcdef\
2. 1234567890abcdef\
3. 1234567890abcdef\
4. 1234567890abcdef\
5. 1234567890abcdef\
6. 1234567890abcdef\
7. 1234567890abcdef\
8. 1234567890abcdef\
9. 1234567890abcdef\
10. 1234567890abcdef\
11. 1234567890abcdef\
12. 1234567890abcdef\
13. 1234567890abcdef\
14. 1234567890abcdef\
15. 1234567890abcdef\
16. 1234567890abcdef\0";

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 单核（双核机器禁掉一核），进程/线程数：255  ，打印次数100 | | | | | | |
|  | 第1次 | 第2次 | 第3次 | 第4次 | 第5次 | 平均 |
| 多进程 | 0m6.977s | 0m7.358s | 0m7.520s | 0m7.282s | 0m7.218s | 0m7.286 |
| 多线程 | 0m7.035s | 0m7.552s | 0m7.087s | 0m7.427s | 0m7.257s | 0m7.257 |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 单核（双核机器禁掉一核），进程/线程数：  255，打印次数500 | | | | | | |
|  | 第1次 | 第2次 | 第3次 | 第4次 | 第5次 | 平均 |
| 多进程 | 0m35.666s | 0m36.009s | 0m36.532s | 0m35.578s | 0m41.537s | 0m36.069 |
| 多线程 | 0m37.290s | 0m35.688s | 0m36.377s | 0m36.693s | 0m36.784s | 0m36.618 |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 单核（双核机器禁掉一核），进程/线程数： 255，打印次数1000 | | | | | | |
|  | 第1次 | 第2次 | 第3次 | 第4次 | 第5次 | 平均 |
| 多进程 | 1m8.864s | 1m11.056s | 1m10.273s | 1m12.317s | 1m20.193s | 1m11.215 |
| 多线程 | 1m11.949s | 1m13.088s | 1m12.291s | 1m9.677s | 1m12.210s | 1m12.150 |

【实验结论】

从上面的实验比对结果看，即使Linux2.6使用了新的NPTL线程库（据说比原线程库性能提高了很多，唉，又是据说！），多线程比较多进程在效率上没有任何的优势，在线程数增大时多线程程序还出现了运行错误，实验可以得出下面的结论：

在Linux2.6上，多线程并不比多进程速度快，考虑到线程栈的问题，多进程在并发上有优势。

**四、多进程和多线程在创建和销毁上的效率比较**

预先创建进程或线程可以节省进程或线程的创建、销毁时间，在实际的应用中很多程序使用了这样的策略，比如Apapche预先创建进程、Tomcat 预先创建线程，通常叫做进程池或线程池。在大部分人的概念中，进程或线程的创建、销毁是比较耗时的，在stevesn的著作《Unix网络编程》中有这样 的对比图（第一卷 第三版 30章 客户/服务器程序设计范式）：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **行号** | **服务器描述** | **进程控制CPU时间（秒，与基准之差）** | | |
| **Solaris2.5.1** | **Digital Unix4.0b** | **BSD/OS3.0** |
| 0 | 迭代服务器（基准测试，无进程控制） | 0.0 | 0.0 | 0.0 |
| 1 | 简单并发服务，为每个客户请求fork一个进程 | 504.2 | 168.9 | 29.6 |
| 2 | 预先派生子进程，每个子进程调用accept |  | 6.2 | 1.8 |
| 3 | 预先派生子进程，用文件锁保护accept | 25.2 | 10.0 | 2.7 |
| 4 | 预先派生子进程，用线程互斥锁保护accept | 21.5 |  |  |
| 5 | 预先派生子进程，由父进程向子进程传递套接字 | 36.7 | 10.9 | 6.1 |
| 6 | 并发服务，为每个客户请求创建一个线程 | 18.7 | 4.7 |  |
| 7 | 预先创建线程，用互斥锁保护accept | 8.6 | 3.5 |  |
| 8 | 预先创建线程，由主线程调用accept | 14.5 | 5.0 |  |

stevens已驾鹤西去多年，但《Unix网络编程》一书仍具有巨大的影响力，上表中stevens比较了三种服务器上多进程和多线程的执行效 率，因为三种服务器所用计算机不同，表中数据只能纵向比较，而横向无可比性，stevens在书中提供了这些测试程序的源码（也可以在网上下载）。书中介 绍了测试环境，两台与服务器处于同一子网的客户机，每个客户并发5个进程（服务器同一时间最多10个连接），每个客户请求从服务器获取4000字节数据， 预先派生子进程或线程的数量是15个。

第0行是迭代模式的基准测试程序，服务器程序只有一个进程在运行（同一时间只能处理一个客户请求），因为没有进程或线程的调度切换，因此它的速度是 最快的，表中其他服务模式的运行数值是比迭代模式多出的差值。迭代模式很少用到，在现有的互联网服务中，DNS、NTP服务有它的影子。第1～5行是多进 程服务模式，期中第1行使用现场fork子进程，2～5行都是预先创建15个子进程模式，在多进程程序中套接字传递不太容易（相对于多线 程），stevens在这里提供了4个不同的处理accept的方法。6～8行是多线程服务模式，第6行是现场为客户请求创建子线程，7～8行是预先创建 15个线程。表中有的格子是空白的，是因为这个系统不支持此种模式，比如当年的BSD不支持线程，因此BSD上多线程的数据都是空白的。

从数据的比对看，现场为每客户fork一个进程的方式是最慢的，差不多有20倍的速度差异，Solaris上的现场fork和预先创建子进程的最大差别是504.2 ：21.5，但我们不能理解为预先创建模式比现场fork快20倍，原因有两个：

1. stevens的测试已是十几年前的了，现在的OS和CPU已起了翻天覆地的变化，表中的数值需要重新测试。

2. stevens没有提供服务器程序整体的运行计时，我们无法理解504.2 ：21.5的实际运行效率，有可能是1504.2 : 1021.5，也可能是100503.2 : 100021.5，20倍的差异可能很大，也可能可以忽略。

因此我写了下面的实验程序，来计算在Linux2.6上创建、销毁10万个进程/线程的绝对用时。

创建10万个进程（forkcreat.c）:

1. #include <stdio.h>
2. #include <signal.h>
3. #include <stdio.h>
4. #include <unistd.h>
5. #include <sys/stat.h>
6. #include <fcntl.h>
7. #include <sys/types.h>
8. #include <sys/wait.h>
9. int count;//子进程创建成功数量
10. int fcount;//子进程创建失败数量
11. int scount;//子进程回收数量
12. /\*信号处理函数–子进程关闭收集\*/
13. void sig\_chld(int signo)
14. {
15. pid\_t chldpid;//子进程id
16. int stat;//子进程的终止状态
18. //子进程回收，避免出现僵尸进程
19. while((chldpid=wait(&stat)>0))
20. {
21. scount++;
22. }
23. }
24. int main()
25. {
26. //注册子进程回收信号处理函数
27. signal(SIGCHLD,sig\_chld);
29. int i;
30. for(i=0;i<100000;i++)//fork()10万个子进程
31. {
32. pid\_t pid=fork();
33. if(pid==-1)//子进程创建失败
34. {
35. fcount++;
36. }
37. else if(pid>0)//子进程创建成功
38. {
39. count++;
40. }
41. else if(pid==0)//子进程执行过程
42. {
43. exit(0);
44. }
45. }
47. printf("count:%d fount:%d scount:%d\n",count,fcount,scount);
48. }

创建10万个线程（pthreadcreat.c）:

1. #include <stdio.h>
2. #include <pthread.h>
3. int count=0;//成功创建线程数量
4. void thread(void)
5. {
6. //啥也不做
7. }
8. int main(void)
9. {
10. pthread\_t id;//线程id
11. int i,ret;
13. for(i=0;i<100000;i++)//创建10万个线程
14. {
15. ret=pthread\_create(&id,NULL,(void \*)thread,NULL);
16. if(ret!=0)
17. {
18. printf("Create pthread error!\n");
19. return(1);
20. }
21. count++;
22. pthread\_join(id,NULL);
23. }
25. printf("count:%d\n",count);
26. }

创建10万个线程的Java程序:

1. public class ThreadTest
2. {
3. public static void main(String[] ags) throws InterruptedException
4. {
5. System.out.println("开始运行");
6. long start = System.currentTimeMillis();
7. for(int i = 0; i < 100000; i++) //创建10万个线程
8. {
9. Thread athread = new Thread(); //创建线程对象
10. athread.start(); //启动线程
11. athread.join(); //等待该线程停止
12. }
14. System.out.println("用时：" + (System.currentTimeMillis() – start) + " 毫秒");
15. }
16. }

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 单核（双核机器禁掉一核），创建销毁10万个进程/线程 | | | | | | |
|  | 第1次 | 第2次 | 第3次 | 第4次 | 第5次 | 平均 |
| 多进程 | 0m8.774s | 0m8.780s | 0m8.475s | 0m8.592s | 0m8.687s | 0m8.684 |
| 多线程 | 0m0.663s | 0m0.660s | 0m0.662s | 0m0.660s | 0m0.661s | 0m0.661 |

|  |
| --- |
| **创建销毁10万个线程（Java）** |
| 12286毫秒 |

从数据可以看出，多线程比多进程在效率上有10多倍的优势，但不能让我们在使用哪种并发模式上定性，这让我想起多年前政治课上的一个场景：在讲到优越性时，面对着几个对此发表质疑评论的调皮男生，我们的政治老师发表了高见，“不能只横向地和当今的发达国家比，你应该纵向地和过去中国几十年的发展历史 比”。政治老师的话套用在当前简直就是真理，我们看看，即使是在赛扬CPU上，创建、销毁进程/线程的速度都是空前的，可以说是有质的飞跃的，平均创建销毁一个进程的速度是0.18毫秒，对于当前服务器几百、几千的并发量，还有预先派生子进程/线程的必要吗？

预先派生子进程/线程比现场创建子进程/线程要复杂很多，不仅要对池中进程/线程数量进行动态管理，还要解决多进程/多线程对accept的“抢” 问题，在stevens的测试程序中，使用了“惊群”和“锁”技术。即使stevens的数据表格中，预先派生线程也不见得比现场创建线程快，在 《Unix网络编程》第三版中，新作者参照stevens的测试也提供了一组数据，在这组数据中，现场创建线程模式比预先派生线程模式已有了效率上的优势。因此我对这一节实验下的结论是：

预先派生进程/线程的模式（进程池、线程池）技术，不仅复杂，在效率上也无优势，在新的应用中可以放心大胆地为客户连接请求去现场创建进程和线程。

我想，这是fork迷们最愿意看到的结论了。

**五、双核系统重复周丽论文实验步骤**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 双核，进程/线程数：255 ，打印次数10 | | | | | | |
|  | 第1次 | 第2次 | 第3次 | 第4次 | 第5次 | 平均(单核倍数) |
| 多进程 | 0m0.061s | 0m0.053s | 0m0.068s | 0m0.061s | 0m0.059s | 0m0.060(1.73) |
| 多线程 | 0m0.054s | 0m0.040s | 0m0.053s | 0m0.056s | 0m0.042s | 0m0.050(1.84) |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 双核，进程/线程数： 255，打印次数100 | | | | | | |
|  | 第1次 | 第2次 | 第3次 | 第4次 | 第5次 | 平均(单核倍数) |
| 多进程 | 0m0.918s | 0m1.198s | 0m1.241s | 0m1.017s | 0m1.172s | 0m1.129(0.93) |
| 多线程 | 0m0.897s | 0m1.166s | 0m1.091s | 0m1.360s | 0m0.997s | 0m1.085(0.85) |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 双核，进程/线程数： 255，打印次数1000 | | | | | | |
|  | 第1次 | 第2次 | 第3次 | 第4次 | 第5次 | 平均(单核倍数) |
| 多进程 | 0m11.276s | 0m11.269s | 0m11.218s | 0m10.919s | 0m11.201s | 0m11.229(1.04) |
| 多线程 | 0m11.525s | 0m11.984s | 0m11.715s | 0m11.433s | 0m10.966s | 0m11.558(1.13) |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 双核，进程/线程数：255 ，打印次数10000 | | | | | | |
|  | 第1次 | 第2次 | 第3次 | 第4次 | 第5次 | 平均(单核倍数) |
| 多进程 | 1m54.328s | 1m54.748s | 1m54.807s | 1m55.950s | 1m57.655s | 1m55.168(1.16) |
| 多线程 | 2m3.021s | 1m57.611s | 1m59.139s | 1m58.297s | 1m57.258s | 1m58.349(1.35) |

【实验结论】

双核处理器在完成任务量较少时，没有系统其他瓶颈因素影响时基本上是单核的两倍，在任务量较多时，受系统其他瓶颈因素的影响，速度明显趋近于单核的速度。

**六、并发服务的不可测性**

看到这里，你会感觉到我有挺进程、贬线程的论调，实际上对于现实中的并发服务具有不可测性，前面的实验和结论只可做参考，而不可定性。对于不可测性，我举个生活中的例子。

这几年在大都市生活的朋友都感觉城市交通状况越来越差，到处堵车，从好的方面想这不正反应了我国GDP的高速发展。如果你7、8年前来到西安市，穿 过南二环上的一些十字路口时，会发现一个奇怪的U型弯的交通管制，为了更好的说明，我画了两张图来说明，第一张图是采用U型弯之前的，第二张是采用U型弯 之后的。

为了讲述的方便，我们不考虑十字路口左拐的情况，在图一中东西向和南北向的车辆交汇在十字路口，用红绿灯控制同一时间只能东西向或南北向通行，一般 的十字路口都是这样管控的。随着车辆的增多，十字路口的堵塞越来越严重，尤其是上下班时间经常出现堵死现象。于是交通部门在不动用过多经费的情况下而采用 了图二的交通管制，东西向车辆行进方式不变，而南北向车辆不能直行，需要右拐到下一个路口拐一个超大的U型弯，这样的措施避免了因车辆交错而引发堵死的次 数，从而提高了车辆的通过效率。我曾经问一个每天上下班乘公交经过此路口的同事，他说这样的改动不一定每次上下班时间都能缩短，但上班时间有保障了，从而 迟到次数减少了。如果今天你去西安市的南二环已经见不到U型弯了，东西向建设了高架桥，车辆分流后下层的十字路口已恢复为图一方式。

从效率的角度分析，在图一中等一个红灯45秒，远远小于图二拐那个U型弯用去的时间，但实际情况正好相反。我们可以设想一下，如果路上的所有运行车 辆都是同一型号（比如说全是QQ3微型车），所有的司机都遵守交规，具有同样的心情和性格，那么图一的通行效率肯定比图二高。现实中就不一样了，首先车辆 不统一，有大车、小车、快车、慢车，其次司机的品行不一，有特别遵守交规的，有想耍点小聪明的，有性子慢的，也有的性子急，时不时还有三轮摩托逆行一下， 十字路口的“死锁”也就难免了。

那么在什么情况下图二优于图一，是否能拿出一个科学分析数据来呢？以现在的科学技术水平是拿不出来的，就像长期的天气预报不可预测一样，西安市的交管部门肯定不是分析各种车辆的运行规律、速度，再进行复杂的社会学、心理学分析做出U型弯的决定的，这就是要说的不可测性。

现实中的程序亦然如此，比如WEB服务器，有的客户在快车道（宽带），有的在慢车道（窄带），有的性子慢（等待半分钟也无所谓），有的性子急（拼命 的进行浏览器刷新），时不时还有一两个黑客混入其中，这种情况每个服务器都不一样，既是是同一服务器每时每刻的变化也不一样，因此说不具有可测性。开发者 和维护者能做的，不论是前面的这种实验测试，还是对具体网站进行的压力测试，最多也就能模拟相当于QQ3通过十字路口的场景。

**结束语**

本篇文章比较了Linux系统上多线程和多进程的运行效率，在实际应用时还有其他因素的影响，比如网络通讯时采用长连接还是短连接，是否采用 select、poll，java中称为nio的机制，还有使用的编程语言，例如Java不能使用多进程，[**PHP**](http://lib.csdn.net/base/php)不能使用多线程，这些都可能影响到并发模 式的选型。

最后还有两点提醒：

1. 文章中的所有实验数据有环境约束。  
2. 由于并行服务的不可测性，文章中的观点应该只做参考，而不要去定性。