炉边夜话

——多核多线程杂谈

极光 著

跋

多核多线程已经成为当前一个时髦的话题，早在 2005 年 C++大师 Herb Sutter 就说过免费的午餐已经结束，并发编程的时代已经来临。从接触第一个多线程项目以来，已经是第四个年头了，一直想把这几年的点点滴滴写成文章，让刚接触的人少走弯路，这便是我写这本书的初衷。

很早就有了此想法，一是由于自己懒惰，二是业余时间比较少，三是才疏学浅不敢瞎写，所以一直未能成行。趁着这段休闲的日子，将以前发表过的文章集结成册，便于大家阅读。

虽然只是 V1.0 版，但我会不断的更新版本，后续会将自己心得或者项目的开发经验添加进本书，同时也会将以前发表的文章进行细化，讲透讲明白。由于是第一次写书，经验欠缺，只是站在自己的角度看待问题，不知道读者您是如何看待多核多线程编程的，欢迎您的来信，讲述您对多核多线程的疑惑，以及开发过程中碰到的问题及经验，可能我解答不了您的疑问，但我们可以就相关问题进行探讨。欢迎大家给本书提建议和挑错，也许您的建议将决定本书内容的走向，挑错可以避免误导其他读者。

本书所讲述内容针对于 linux 平台，其他平台仅供参考。

欢迎大家给我发 E-Mail:normalnotebook@126.com，就多核多线程问题进行相关讨论。

个人博客地址：http://blog.csdn.net/normalnotebook

原理篇

铁路与多核多线程

如何理解多核多线程这些概念呢?

在自然世界中，总有那么一些事物是类似的。如果我们加以抽象和归纳，就可以得出相同或者相类似的结论，比如铁路系统和多核多线程就有相似之处。

对于任何一名出门在外的人来说，春节能买张回家的火车票，不能不说是人生的一大幸事。相信大家在排队买票的时候，曾经都抱怨过，怎么不多增开几列火车，让大家早点回家呢？当这种想法开始在大脑中萌发时，证明您已经拥有了多线程的思想。对于相同的出发地和目的地，通过增开列车的方法来提高运输效率，就相当于在一个进程中采用多线程的方法来提高程序的吞吐率。

从成本和资源利用率的角度考虑，不可能为不同的目的地修建单独的铁路，而会共享相关路段。当不同列车需要同时使用同一路段时，就会造成道路的竞争，交通一旦拥塞，效率开始变的低下和事故频发。为了解决这个问题，信号灯开始出现在关键岔口上，只有当列车收到允许通过的信息后，才能继续前行。如果从多线程的角度来看待道路竞争引发的问题，可以认为是线程的同步问题。在多线程程序中，可以利用各种锁或者信号量等同步手段来解决资源冲突的矛盾，只有解决了资源冲突问题，才能保证程序的正确性。

是不是多增开几列火车，就可以缓解买票难的问题呢（这里不讨论各方利益的问题）？临客便开始出现在人们的视野中。如果临客和其他列车拥有相同的待遇，在关键路径上，按照 FIFO 的策略，排队一一通过，就有可能造成大量的火车晚点，久而久之，最终造成整个铁路系统的瘫痪。于是引入了优先级的概念，一旦发生资源冲突时，临客就停下来到旁边歇歇，让道给高优先级的大佬们（D/Z/T/K/N 等开头的火车）。从操作系统的角度来观察临客的解决方案，这个过程相当于多线程中的线程调度，让不同的线程拥有不同的优先级和调度策略，来提高程序的整体效率。

解决了临客的调度问题，真的可以解决买票难的问题吗？不能，于是新的方案——大修铁路——又被提上了日程，减少关键资源的冲突，让不同目的地的火车在不同铁路线路上驰骋。这种方法就相当于我们今天谈论的多核技术，让不同功能的进程在不同的核上运行，或者让同一进程的不同功能的线程运行在不同的核上。

将多核多线程与实际生活中的例子进行类比，多核多线程显得并不神秘。它只是用来提高程序运行效率的一种手段，暂时还不会打破现有的编程模式。

测试你对多核多线程的认知程度

目前，多核多线程编程已经成为一种趋势，但大部分程序员还没有从串行程序的思维中走出来。即使有些人对多核多线程的概念有所了解，但也是一知半解，写起多核多线程程序来总是束手束脚。

据 Intel 预测，到 2013 年 CPU 将达到 256 核。掐指头算一算，也就是还有 5 年的时间，但留给我们程序员的时间却很少了。这不是危言耸听，现实情况确实如此。如果从现在就开始重视这一问题，不断的学习，并加以积累，相信不久的将来，也许你就比别人多了一次机会。

我曾经对周围的朋友做过一次有趣的调查，调查对象都曾有过多线程编码经验，以此来了解大家对多核与多线程的认知程度。当然不可否认，由于自身知识水平的有限，问卷存在一定的片面性。

样例程序：

```
1.  #ifdef __cplusplus
2.  extern "C"
3.  {
4.  #endif
5.
6.  #include <stdio.h>
7.  #include <sys/types.h>
8.  #include <sys/time.h>
9.  #include <pthread.h>
10. #include <unistd.h>
11.
12. #define ORANGE_MAX_VALUE      1000000
13. #define APPLE_MAX_VALUE       100000000
14. #define MSECOND               1000000
15.
16. struct apple
17. {
18.     unsigned long long a;
19.     unsigned long long b;
20. };
21.
22.
23. struct orange
24. {
25.     int a[ORANGE_MAX_VALUE];
26.     int b[ORANGE_MAX_VALUE];
27.
28. };
29.
30.
31.
```

```
32.     int main (int argc, const char * argv[]) {
33.         // insert code here...
34.         struct apple test;
35.         struct orange test1={{0},{0}};
36.
37.         unsigned long long sum=0,index=0;
38.         struct timeval tpstart,tpend;
39.         float  timeuse;
40.
41.         test.a= 0;
42.         test.b= 0;
43.
44.         /*get start time*/
45.         gettimeofday(&tpstart,NULL);
46.
47.         for(sum=0;sum<APPLE_MAX_VALUE;sum++)
48.         {
49.             test.a += sum;
50.             test.b += sum;
51.         }
52.
53.         for(index=0;index<ORANGE_MAX_VALUE;index++)
54.         {
55.             sum=test1.a[index]+test1.b[index];
56.         }
57.
58.         /*get start time*/
59.         gettimeofday(&tpend,NULL);
60.
61.         /*calculate time*/
62.         timeuse=MSECOND*(tpend.tv_sec-tpstart.tv_sec)+tpend.tv_usec-tpstart.tv_usec;
63.         timeuse/=MSECOND;
64.         printf("main thread:%x,Used Time:%f\n",pthread_self(),timeuse);
65.
66.         printf("a = %llu,b = %llu,sum=%llu\n",test.a,test.b,sum);
67.
68.         return 0;
69.     }
70.
71.
72.     #ifdef __cplusplus
73.     }
74.     #endif
75.
```

假设有一台酷睿 2 代双核机器，在此机器上对上述程序进行如下优化，您会如何选择呢?

Q1: 您认为样例程序还有优化的空间吗? 如果有，优化后的效率将会提升:

A. 1%~30%　　B. 30%~50%　　C. 50%~90%　　D. 90%以上

Q2: 如果将样例程序修改为两个线程，一个线程用于计算 apple 的和，另外一个线程计算 orange 的和，您认为谁的效率会更高?

A. 两线程　　B. 单线程（样例程序）　　C. 不确定

Q3: 基于 Q2，再将计算 apple 的线程拆成两个线程，一个线程用于计算 apple a 的值(加锁访问)，另外一个线程计算 apple b 的值(加锁访问)，第三个线程计算 orange 的和，您认为谁的效率会更高?

A. 两线程　　B. 单线程（样例程序）　　C. 三线程　　D. 不确定

Q4: 基于 Q2，在双核 CPU 系统上，将计算 apple 的线程绑定到 CPU 0 上运行，将计算 orange 和的线程绑定到 CPU 1 上运行，这种方法称为设置 CPU 亲和力（CPU Affinity，也叫 CPU 绑定）您认为谁的效率会更高?

A. 两线程　B. 单线程（样例程序）　C. 两线程（CPU 绑定）　D. 不确定

Q5: 经过分析发现计算 orange 的和比较快，而计算 apple 的和比较慢。基于 Q3，将计算 apple a 的线程和计算 orange 和的线程绑定到 CPU 0 上运行，将计算 apple b 的线程绑定到 CPU 1 上运行，您认为谁的效率会更高?

A. 三线程　B. 单线程（样例程序）　C. 三线程（CPU 绑定）　D. 不确定

Q6: 在 Q3 中，将程序拆成多线程，需要加锁来访问 apple a 和 b 的值，但由于他们访问的是数据结构中的不同属性，也可以不加锁，此时您认为谁的效率会更高?

A. 加锁访问　　B. 不加锁访问　　C. 不确定

如果想迫不及待的知道答案，可以跳过本节原理篇，直接阅读实践篇或者工具篇的文章。

水煮多线程

计算机系统变得越来越复杂，多线程机制给我们带来了能够继续管理它们的希望。

——*Andrew Koening and Barbara Moo*

线程的基本概念

进程（*process*）和文件（*files*）是 UNIX/Linux 操作系统两个最基本的抽象。进程是处于执行期的程序和它所包含的资源的总和，也就是说一个进程就是处于执行期的程序。一个**线程**（*thread*）就是运行在一个进程上下文中的一个逻辑流，不难看出，线程是进程中最基本的活动对象。

在传统的系统中，一个进程只包含一个线程。但在现代操作系统中，允许一个进程里面可以同时运行多个线程，这类程序就被称为多线程程序。所有的程序都有一个主线程（*main thread*），主线程是进程的控制流或执行线程。在多线程程序中，主线程可以创建一个或多个对等线程（*peer thread*），从这个时间点开始，这些线程就开始并发执行。主线程和对等线程的区别仅在于主线程总是进程中第一个运行的线程。从某种程度上看，线程可以看作是轻量级的进程（*lightweight process*）。在 Linux 操作系统中，内核调度的基本对象是线程，而不是进程，所以进程中的多个线程将由内核自动调度。

每个线程都拥有独立的线程上下文（*thread context*），线程 ID（*Thread ID，TID*），程序计数器（*pc*），线程栈（*stack*），一组寄存器（*register*）和条件码。其中，内核正是通过线程 ID（TID）来识别线程，进行线程调度的。

线程与进程的异同点

线程和进程在很多方面是相似的。相同点主要表现在如下几方面：

1. 比如都具有 ID，一组寄存器，状态，优先级以及所要遵循的调度策略。
2. 每个进程都有一个进程控制块，线程也拥有一个线程控制块（在 Linux 内核，线程控制块与进程控制块用同一个结构体描述，即 struct task_struct），这个控制块包含线程的一些属性信息，操作系统使用这些属性信息来描述线程。
3. 线程和子进程共享父进程中的资源。
4. 线程和子进程独立于它们的父进程，竞争使用处理器资源。
5. 线程和子进程的创建者可以在线程和子进程上实行某些控制，比如，创建者可以取消、挂起、继续和修改线程和子进程的优先级。
6. 线程和子进程可以改变其属性并创建新的资源

除了这些相同点，在很多方面也存在着差异：

1. 主要区别：每个进程都拥有自己的地址空间，但线程没有自己独立的地址空间，而是运行在一个进程里的所有线程共享该进程的整个虚拟地址空间
2. 线程的上下文切换时间开销比进程上下文切换时间开销要小的多
3. 线程的创建开销远远小于进程的创建
4. 子进程拥有自己的地址空间和数据段的拷贝，因此当子进程修改它的变量和数据时，它不会影响父进程中的数据，但线程可以直接访问它进程中的数据段
5. 进程之间通讯必须使用进程间通讯机制，但线程可以与进程中的其他线程直接通讯
6. 线程可以对同一进程中的其他线程实施大量控制，但进程只能对子进程实施控制
7. 改变主线程的属性可能影响进程中其他的线程，但对父进程的修改不影响子进程

深入剖析线程

线程优缺点

线程的优点	线程的缺点
上下文切换需要更少的系统资源	并发读/写需要同步
增加了应用程序的吞吐量	很容易破坏它进程的地址空间
任务间的通讯不需要特殊机制	只存在于单个进程中，因此不能被重用
简化了程序结构	

表 1 线程优缺点对比表

线程属性

POSIX 线程库定义了**线程属性对象**，它封装了线程的创建者可以访问和修改的线程属性。线程属性主要包括如下属性：

1. 作用域（scope）
2. 栈尺寸（stack size）
3. 栈地址（stack address）
4. 优先级（priority）
5. 分离的状态（detached state）
6. 调度策略和参数（scheduling policy and parameters）

线程属性对象可以与一个线程或多个线程相关联。当使用线程属性对象时，它是对线程和线程组行为的配置。使用属性对象的所有线程都将具有由属性对象所定义的所有属性。虽然它们共享属性对象，但它们维护各自独立的线程 ID 和寄存器。

线程可以在两种竞争域内竞争资源：

1. 进程域（process scope）：与同一进程内的其他线程

2. 系统域（system scope）：与系统中的所有线程

作用域属性描述特定线程将与哪些线程竞争资源。一个具有系统域的线程将与整个系统中所有具有系统域的线程按照优先级竞争处理器资源，进行调度。

分离线程是指不需要和进程中其他线程同步的线程。也就是说，没有线程会等待分离线程退出系统。因此，一旦该线程退出，它的资源（如线程 ID）可以立即被重用。

线程的布局嵌入在进程的布局中。进程有代码段、数据段和栈段，而线程与进程中的其他线程共享代码段和数据段，每个线程都有自己的栈段，这个栈段在进程地址空间的栈段中进行分配。线程栈的尺寸在线程创建时设置。如果在创建时没有设置，那么系统将会指定一个默认值，缺省值的大小依赖于具体的系统。

POSIX 线程属性对象中可设置的线程属性及其含义参见下表：

函数	属性	含义
int pthread_attr_setdetachstate (pthread_attr_t* attr ,int detachstate)	detachstate	detachstate 属性控制一个线程是否是可分离的
int pthread_attr_setguardsize (pthread_attr_t* attr ,size_t guardsize)	guardsize	guardsize 属性设置新创建线程栈的保护区大小
int pthread_attr_setinheritsched (pthread_attr_t* attr, int inheritsched)	inheritsched	inheritsched 决定怎样设置新创建线程的调度属性
int pthread_attr_setschedparam (pthread_attr_t* attr , const struct sched_param* restrict param)	param	param 用来设置新创建线程的优先级
int pthread_attr_setschedpolicy (pthread_attr_t* attr, int policy)	policy	Policy 用来设置先创建线程的调度策略
int pthread_attr_setscope (pthread_attr_t* attr , int contentionscope)	contentionscope	contentionscope 用于设置新创建线程的作用域
int pthread_attr_setstack (pthread_attr_t* attr, void* stackader, size_t stacksize)	stackader stacksize	两者共同决定了线程栈的基地址以及堆栈的最小尺寸（以字节为单位）
int pthread_attr_setstackaddr	stackader	stackader 决定了新创建线程的栈的

(pthread_attr_t* attr , void* stackader)		基地址
int pthread_attr_setstacksize (pthread_attr_t* attr, size_t stacksize)	stacksize	stacksize 决定了新创建线程的栈的最小尺寸（以字节为单位）

表 2 线程属性及其含义

线程调度和优先级

进程的调度策略和优先级属于主线程，换句话说就是设置进程的调度策略和优先级只会影响主线程的调度策略和优先级，而不会改变对等线程的调度策略和优先级（注：这句话不完全正确）。每个对等线程可以拥有它自己的独立于主线程的调度策略和优先级。

在 Linux 系统中，进程有三种调度策略：SCHED_FIFO、SCHED_RR 和 SCHED_OTHER，线程也不例外，也具有这三种策略。

在 pthred 库中，提供了一个函数，用来设置被创建的线程的调度属性：是从创建者线程继承调度属性，还是从属性对象设置调度属性。该函数就是：

int pthread_attr_setinheritsched (pthread_attr_t *__attr, int __inherit)

其中，inherit 的值为下列值中的其一：

```
enum
{
    PTHREAD_INHERIT_SCHED, //线程调度属性从创建者线程继承
    PTHREAD_EXPLICIT_SCHED //线程调度属性设置为 attr 设置的属性
};
```

如果在创建新的线程时，调用该函数将参数设置为 PTHREAD_INHERIT_SCHED 时，那么当修改进程的优先级时，该进程中继承这个优先级并且还没有改变其优先级的所有线程也将会跟着改变优先级（现在应该明白我刚才为什么说那句话部正确的原因了吧）。

线程模型

线程的目的就是代表进程执行工作，如果一个进程拥有多个线程，进程就可以根据一定的策略和方法来管理这些线程，并进行相关工作。先辈们通过总结与抽象，创造了线程模型的概念，这里的线程模型就是我们常说的架构，设计模式等，为我们这些后辈们提供了借鉴的意义。

线程模型总览

经过前人的摸索，总结与抽象，提炼出了一些通用的模型：

1. 委托模型（delegation），又称为 boss-worker 模型
2. 对等模型（peer-to-peer）

3. 管道模型

4. 生产者-消费者模型（producer-consumer）

各个模型的含义，如下表所示：

线程模型	含义
委托模型	Boss 线程创建其他线程（worker 线程），并给每个 worker 线程分配任务。Boss 线程可能在每个线程完成它们的任务之前一直等待。
对等模型	所有的线程都具有相同的工作状态。对等线程创建执行任务所需要的所有线程，但不执行委托职责。对等线程可以从单个输入流处理请求，这些输入流被所有线程共享，或者每个线程都有其自己的输入流。
管道模型	类似于装配线流程，分阶段处理输入流，然后传给下一个线程进行处理
生-消模型	生产者线程生产数据给消费者线程使用，数据存储在生产者和消费者共享的存储块中。

表 3 线程模型及其含义

委托模型

Boss 线程创建其他线程（worker 线程），并给每个 worker 线程分配任务。Boss 线程可能在每个线程完成它们的任务之前一直等待。**Boss 线程将任务委托给每个 worker 线程是通过指定一个函数来完成的**。由于每个 worker 线程被分配了任务，所以每个 worker 线程的职责就是执行指定的任务。

基于以上原理，可以演变出两类委托模型：

浪费型：当系统有请求任务时，boss 线程创建 worker 线程。对每一个请求的处理就能够委托给一个 worker 线程。在这种情况下，boss 线程执行事件循环，当有事件发生时，就创建一个 worker 线程，并给它们指派职责。

这类应用实际意义不大，可能常见于试验型或者连接请求不多的情况。因为线程不可能无止境的创建下去，更何况创建太多的线程，势必会影响到系统的性能，这样就造成了性能瓶颈。

节约型：boss 线程创建一个线程池，可以为池中的线程指派任务。当有任务到达时，boss 线程发信号通知 worker 线程进行处理。该 worker 线程处理完成之后，接着处理下一个请求。如果没有请求，那么就挂起自身，直到收到 boss 的通知或者等待一个特定的时间，再次读取请求其中节约型中的 boss 线程的主要目的就是创建所有的 worker 线程，将请求放入队列中，然后唤醒 worker 线程。Worker 线程检查队列中的请求，执行指派的任务。如果没有可供处理的任务，便挂起自己。所有的 boss 线程和 worker 线程并发执行。

大名鼎鼎的 IOCP（IO **完成端口**）就是这类模型。

对等模型

管道模型

生产者－消费者模型

线程同步机制

POSIX 标准定义的类型

互斥信号量

读写锁

条件变量

多重条件变量

面向对象的同步方法

实践篇

利用多核多线程进行程序优化[1]

引子

大家也许还记得 2005 年 3 月 C++大师 Herb Sutter 在 Dr.Dobb’s Journal 上发表了一篇名为《免费的午餐已经结束》的文章。文章指出：现在的程序员对效率、伸缩性、吞吐量等一系列性能指标相当忽视，很多性能问题都仰仗越来越快的 CPU 来解决。但 CPU 的速度在不久的将来，即将偏离摩尔定律的轨迹，并达到一定的极限。所以，越来越多的应用程序将不得不直面性能问题，而解决这些问题的办法就是采用并发编程技术。

正如 Herb Sutter 所说，由于串行处理速度的限制，串行化技术在程序设计中的中流砥柱地位将会被取代，而一个多核与并发编程的时代即将来临。但在这个演变的行进过程当中，有时我们会付出一定的代价，甚至事与愿违，性能可能变得越来越差。

本文不是告诉大家如何找出程序中的热点、如何进行性能分析、如何利用多核与多线程技术来提高性能，而是一剂药引，在利用这些技术的同时，还应该值得思考的一些问题，以便达到事半功倍的效果。

K-Best 测量方法

在检测程序运行时间这个复杂问题上，将采用 Randal E.Bryant 和 David R. O’Hallaron 提出的 K 次最优测量方法。假设重复的执行一个程序，并纪录 K 次最快的时间，如果发现测量的误差 ε 很小，那么用测量的最快值表示过程的真正执行时间， 称这种方法为“K 次最优（K-Best）方法”，要求设置三个参数：

K: 要求在某个接近最快值范围内的测量值数量。

ε: 测量值必须多大程度的接近，即测量值按照升序标号 V1, V2, V3, … , Vi, …， 同时必须满足（1+ε）Vi ≥ Vk

M: 在结束测试之前，测量值的最大数量。

按照升序的方式维护一个 K 个最快时间的数组，对于每一个新的测量值，如果比当前 k 处的值更快，则用最新的值替换数组中的元素 K，然后再进行升序排序，持续不断的进行该过程，并满足误差标准，此时就称测量值已经收敛。如果 M 次后，不能满足误差标准，则称为不能收敛。

[1]最初由 IBM developerWorks 中国网站发表，其网址是 http://www.ibm.com/developerworks/cn

在接下来的所有试验中，采用 K=10，ε=2%，M=200 来获取程序运行时间，同时也对 K 次最优测量方法进行了改进，不是采用最小值来表示程序执行的时间，而是采用 K 次测量值的平均值来表示程序的真正运行时间。由于采用的误差 ε 比较大，在所有试验程序的时间收集过程中，均能收敛，但也能说明问题。

为了可移植性，采用 gettimeofday()来获取系统时钟（system clock）时间，可以精确到微秒。

测试环境

硬件：联想 Dual-core 双核机器，主频 2.4G，内存 2G.

软件：Suse Linunx Enterprise 10，内核版本：linux-2.6.16

软件优化的三个层次

医生治病首先要望闻问切，然后确定病因，最后在对症下药，如果胡乱医治一通，不死也残废。说起来大家都懂的道理，但在软件优化过程中，往往都喜欢犯这样的错误。不分青红皂白，一上来这里改改，那里改改，其结果往往不如人意。

一般将软件优化可分为三个层次：系统层面，应用层面及微架构层面。首先从宏观进行考虑，进行望闻问切，即系统层面的优化，把所有与程序相关的信息收集上来，确定病因。确定病因后，开始从微观上进行优化，即进行应用层面和微架构方面的优化。

1. 系统层面的优化：内存不够，CPU 速度过慢，系统中进程过多等
2. 应用层面的优化：算法优化、并行设计等
3. 微架构层面的优化：分支预测、数据结构优化、指令优化等

软件优化可以在应用开发的任一阶段进行，当然越早越好，这样以后的麻烦就会少很多。

在实际应用程序中，采用最多的应用层面的优化，也会采用微架构层面的优化。将某些优化和维护成本进行对比，往往选择的都是后者。如分支预测优化和指令优化，在大型应用程序中，往往采用的比较少，因为维护成本过高。

本文将从应用层面和微架构层面，对样例程序进行优化。对于应用层面的优化，将采用多线程和 CPU 亲和力技术；在微架构层面，采用 Cache 优化。

并行设计

利用并行程序设计模型来设计应用程序，就必须把自己的思维从线性模型中拉出来，重新审视整个处理流程，从头到尾梳理一遍，将能够并行执行的部分识别出来。

可以将应用程序看成是众多相互依赖的任务的集合。将应用程序划分成多个独立的任务，并确定这些任务之间的相互依赖关系，这个过程称为分解（Decomosition）。分解问题的方式主要有三种：任务分解、数据分解和数据流分解。关于这部分的详细资料，请参看参考资料一。

仔细分析样例程序，运用任务分解的方法，不难发现计算 apple 的值和计算 orange 的值，属于完全不相关的两个操作，因此可以并行。

改造后的两线程程序：

```
1.      #ifdef __cplusplus
2.      extern "C"
3.      {
4.      #endif
5.
6.      #include <stdio.h>
7.      #include <sys/types.h>
8.      #include <sys/time.h>
9.      #include <pthread.h>
10.     #include <unistd.h>
11.
12.     #define ORANGE_MAX_VALUE     1000000
13.     #define APPLE_MAX_VALUE      100000000
14.     #define MSECOND              1000000
15.
16.     struct apple
17.     {
18.        unsigned long long a;
19.        unsigned long long b;
20.     };
21.
22.     struct orange
23.     {
24.        int a[ORANGE_MAX_VALUE];
25.        int b[ORANGE_MAX_VALUE];
26.     };
27.
28.     void* add(void* x)
29.     {
30.        unsigned long long sum=0;
31.        struct timeval tpstart,tpend;
32.        float  timeuse=0;
33.
34.        gettimeofday(&tpstart,NULL);
35.
36.        for(sum=0;sum<APPLE_MAX_VALUE;sum++)
37.        {
```

```
38.             ((struct apple *)x)->a += sum;
39.             ((struct apple *)x)->b += sum;
40.         }
41.
42.
43.         gettimeofday(&tpend,NULL);
44.
45.         timeuse=MSECOND*(tpend.tv_sec-tpstart.tv_sec)+tpend.tv_usec-tpstart.tv_usec;
46.         timeuse/=MSECOND;
47.         printf("add thread:%x,Used Time:%f\n",pthread_self(),timeuse);
48.
49.         return NULL;
50.     }
51.
52.     int main (int argc, const char * argv[]) {
53.         // insert code here...
54.         struct apple test;
55.         struct orange test1={{0},{0}};
56.         pthread_t ThreadA;
57.
58.         unsigned long long sum=0,index=0;
59.         struct timeval tpstart,tpend;
60.         float  timeuse;
61.
62.         test.a= 0;
63.         test.b= 0;
64.
65.         gettimeofday(&tpstart,NULL);
66.
67.         pthread_create(&ThreadA,NULL,add,&test);
68.
69.         for(index=0;index<ORANGE_MAX_VALUE;index++)
70.         {
71.             sum+=test1.a[index]+test1.b[index];
72.         }
73.
74.         pthread_join(ThreadA,NULL);
75.
76.         gettimeofday(&tpend,NULL);
77.
78.         timeuse=MSECOND*(tpend.tv_sec-tpstart.tv_sec)+tpend.tv_usec-tpstart.tv_usec;
79.         timeuse/=MSECOND;
80.         printf("main thread:%x,Used Time:%f\n",pthread_self(),timeuse);
81.
82.         printf("a = %llu,b = %llu,sum=%llu\n",test.a,test.b,sum);
83.
84.         return 0;
85.     }
86.
87.     #ifdef __cplusplus
88.     }
89.     #endif
```

更甚一步，通过数据分解的方法，还可以发现，计算apple的值可以分解为两个线程，

一个用于计算apple a的值，另外一个线程用于计算apple b的值(说明：本方案抽象于实际的应用程序)。但两个线程存在同时访问apple的可能性，所以需要加锁访问该数据结构。

改造后的三线程程序如下：

```
1.  #ifdef __cplusplus
2.  extern "C"
3.  {
4.  #endif
5.
6.  #include <stdio.h>
7.  #include <sys/types.h>
8.  #include <sys/time.h>
9.  #include <pthread.h>
10. #include <unistd.h>
11.
12. #define ORANGE_MAX_VALUE      1000000
13. #define APPLE_MAX_VALUE       100000000
14. #define MSECOND               1000000
15.
16. struct apple
17. {
18.    unsigned long long a;
19.    unsigned long long b;
20.    pthread_rwlock_t rwLock;
21. };
22.
23. struct orange
24. {
25.    int a[ORANGE_MAX_VALUE];
26.    int b[ORANGE_MAX_VALUE];
27.
28. };
29.
30.
31. void* addx(void* x)
32. {
33.    unsigned long long sum=0;
34.    struct timeval tpstart,tpend;
35.    float  timeuse;
36.
37.    gettimeofday(&tpstart,NULL);
38.    pthread_rwlock_wrlock(&((struct apple *)x)->rwLock);
39.    for(sum=0;sum<APPLE_MAX_VALUE;sum++)
40.    {
41.       ((struct apple *)x)->a += sum;
42.    }
43.    pthread_rwlock_unlock(&((struct apple *)x)->rwLock);
44.    gettimeofday(&tpend,NULL);
45.
46.    timeuse=MSECOND*(tpend.tv_sec-tpstart.tv_sec)+tpend.tv_usec-tpstart.tv_usec;
47.    timeuse/=MSECOND;
48.    printf("addx thread:%x,Used Time:%f\n",pthread_self(),timeuse);
```

```
49.
50.         return NULL;
51.     }
52.
53.     void* addy(void* y)
54.     {
55.         unsigned long long sum=0;
56.         struct timeval tpstart,tpend;
57.         float  timeuse;
58.
59.         gettimeofday(&tpstart,NULL);
60.         pthread_rwlock_wrlock(&((struct apple *)y)->rwLock);
61.         for(sum=0;sum<APPLE_MAX_VALUE;sum++)
62.         {
63.             ((struct apple *)y)->b += sum;
64.         }
65.         pthread_rwlock_unlock(&((struct apple *)y)->rwLock);
66.         gettimeofday(&tpend,NULL);
67.
68.         timeuse=MSECOND*(tpend.tv_sec-tpstart.tv_sec)+tpend.tv_usec-tpstart.tv_usec;
69.         timeuse/=MSECOND;
70.         printf("addy thread:%x,Used Time:%f\n",pthread_self(),timeuse);
71.
72.         return NULL;
73.     }
74.
75.     int main (int argc, const char * argv[]) {
76.         // insert code here...
77.         struct apple test;
78.         struct orange test1={{0},{0}};
79.         pthread_t ThreadA,ThreadB;
80.
81.         unsigned long long sum=0,index=0;
82.         struct timeval tpstart,tpend;
83.         float  timeuse;
84.
85.         test.a= 0;
86.         test.b= 0;
87.
88.         gettimeofday(&tpstart,NULL);
89.
90.         pthread_rwlock_init(&test.rwLock,NULL);
91.
92.         pthread_create(&ThreadA,NULL,addx,&test);
93.         pthread_create(&ThreadB,NULL,addy,&test);
94.
95.         for(index=0;index<ORANGE_MAX_VALUE;index++)
96.         {
97.             sum+=test1.a[index]+test1.b[index];
98.         }
99.
100.        pthread_join(ThreadA,NULL);
101.        pthread_join(ThreadB,NULL);
102.
103.        gettimeofday(&tpend,NULL);
```

```
104.
105.        timeuse=MSECOND*(tpend.tv_sec-tpstart.tv_sec)+tpend.tv_usec-tpstart.tv_usec;
106.        timeuse/=MSECOND;
107.        printf("main thread:%x,Used Time:%f\n",pthread_self(),timeuse);
108.
109.        printf("a = %llu,b = %llu,sum=%llu\n",test.a,test.b,sum);
110.
111.        return 0;
112.    }
113.
114.    #ifdef __cplusplus
115.    }
116.    #endif
```

这样改造后，真的能达到我们想要的效果吗？通过 K-Best 测量方法，其结果让我们大失所望，如下图：

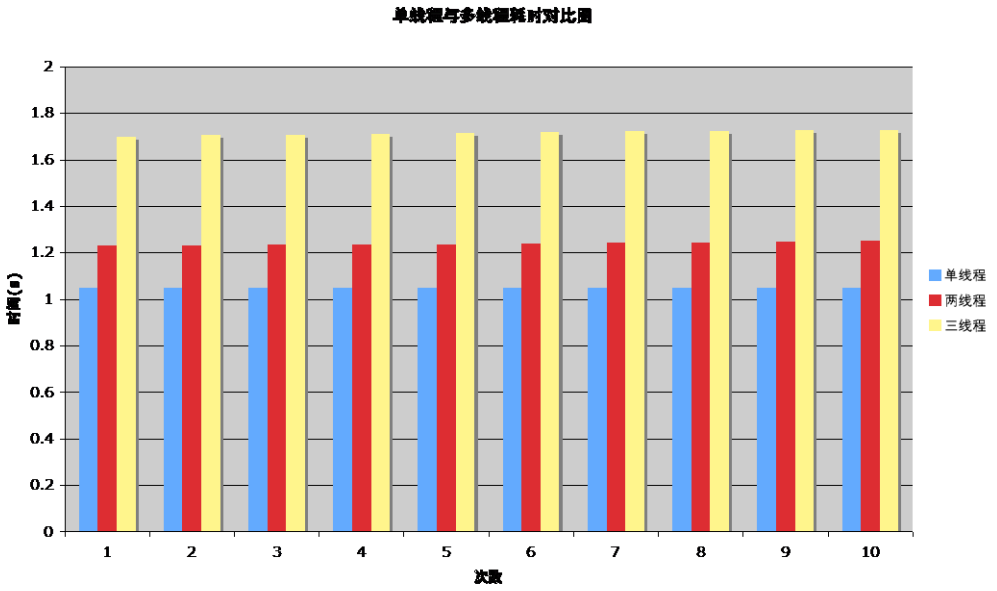

图 1 单线程与多线程耗时对比图

为什么多线程会比单线程更耗时呢？其原因就在于，线程启停以及线程上下文切换会引起额外的开销，所以消耗的时间比单线程多。

为什么加锁后的三线程比两线程还慢呢？其原因也很简单，那把读写锁就是罪魁祸首。通过 Thread Viewer 也可以印证刚才的结果，实际情况并不是并行执行，反而成了串行执行，如下图：

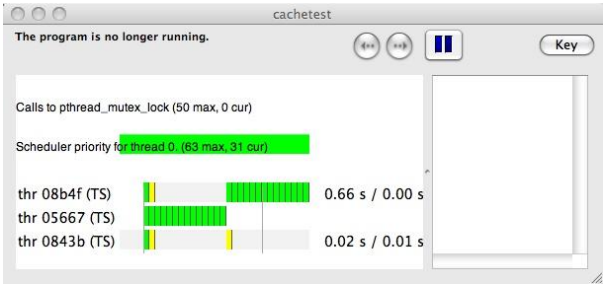

图 2 通过 Viewer 观察三线程运行情况

其中最下面那个线程是主线程，一个是 addx 线程，另外一个是 addy 线程，从图中不难看出，其他两个线程为串行执行。

通过数据分解来划分多线程，还存在另外一种方式，一个线程计算从 1 到 APPLE_MAX_VALUE/2 的值，另外一个线程计算从 APPLE_MAX_VALUE/2+1 到 APPLE_MAX_VALUE 的值，但本文会弃用这种模型，有兴趣的读者可以试一试。

在采用多线程方法设计程序时，如果产生的额外开销大于线程的工作任务，就没有并行的必要。线程并不是越多越好，软件线程的数量尽量能与硬件线程的数量相匹配。最好根据实际的需要，通过不断的调优，来确定线程数量的最佳值。

加锁与不加锁

针对加锁的三线程方案，由于两个线程访问的是 apple 的不同元素，根本没有加锁的必要，所以修改 apple 的数据结构，通过不加锁来提高性能。

```
1.      #ifdef __cplusplus
2.      extern "C"
3.      {
4.      #endif
5.
6.      #include <stdio.h>
7.      #include <sys/types.h>
8.      #include <sys/time.h>
9.      #include <pthread.h>
10.     #include <unistd.h>
11.
12.     #define ORANGE_MAX_VALUE     1000000
13.     #define APPLE_MAX_VALUE      100000000
14.     #define MSECOND              1000000
15.
16.     struct apple
17.     {
18.        unsigned long long a;
19.        unsigned long long b;
20.     };
21.
22.     struct orange
```

```
23.     {
24.         int a[ORANGE_MAX_VALUE];
25.         int b[ORANGE_MAX_VALUE];
26.     };
27.
28.     void* addx(void* x)
29.     {
30.         unsigned long long sum=0;
31.         struct timeval tpstart,tpend;
32.         float  timeuse;
33.
34.         gettimeofday(&tpstart,NULL);
35.
36.         for(sum=0;sum<APPLE_MAX_VALUE;sum++)
37.         {
38.             ((struct apple *)x)->a += sum;
39.         }
40.
41.         gettimeofday(&tpend,NULL);
42.
43.         timeuse=MSECOND*(tpend.tv_sec-tpstart.tv_sec)+tpend.tv_usec-tpstart.tv_usec;
44.         timeuse/=MSECOND;
45.         printf("addx thread:%x,Used Time:%f\n",pthread_self(),timeuse);
46.
47.         return NULL;
48.     }
49.
50.     void* addy(void* y)
51.     {
52.         unsigned long long sum=0;
53.         struct timeval tpstart,tpend;
54.         float  timeuse;
55.
56.         gettimeofday(&tpstart,NULL);
57.
58.         for(sum=0;sum<APPLE_MAX_VALUE;sum++)
59.         {
60.             ((struct apple *)y)->b += sum;
61.         }
62.
63.         gettimeofday(&tpend,NULL);
64.
65.         timeuse=MSECOND*(tpend.tv_sec-tpstart.tv_sec)+tpend.tv_usec-tpstart.tv_usec;
66.         timeuse/=MSECOND;
67.         printf("addy thread:%x,Used Time:%f\n",pthread_self(),timeuse);
68.
69.         return NULL;
70.     }
71.
72.     int main (int argc, const char * argv[]) {
73.         // insert code here...
74.         struct apple test;
75.         struct orange test1={{0},{0}};
76.         pthread_t ThreadA,ThreadB;
77.
```

```
78.         unsigned long long sum=0,index=0;
79.         struct timeval tpstart,tpend;
80.         float  timeuse;
81.
82.         test.a= 0;
83.         test.b= 0;
84.
85.         gettimeofday(&tpstart,NULL);
86.
87.         pthread_create(&ThreadA,NULL,addx,&test);
88.         pthread_create(&ThreadB,NULL,addy,&test);
89.
90.         for(index=0;index<ORANGE_MAX_VALUE;index++)
91.         {
92.            sum+=test1.a[index]+test1.b[index];
93.         }
94.
95.         pthread_join(ThreadA,NULL);
96.         pthread_join(ThreadB,NULL);
97.
98.         gettimeofday(&tpend,NULL);
99.
100.        timeuse=MSECOND*(tpend.tv_sec-tpstart.tv_sec)+tpend.tv_usec-tpstart.tv_usec;
101.        timeuse/=MSECOND;
102.        printf("main thread:%x,Used Time:%f\n",pthread_self(),timeuse);
103.
104.        printf("a = %llu,b = %llu,sum=%llu\n",test.a,test.b,sum);
105.
106.        return 0;
107.     }
108.
109.     #ifdef __cplusplus
110.     }
111.     #endif
```

测试结果如下：

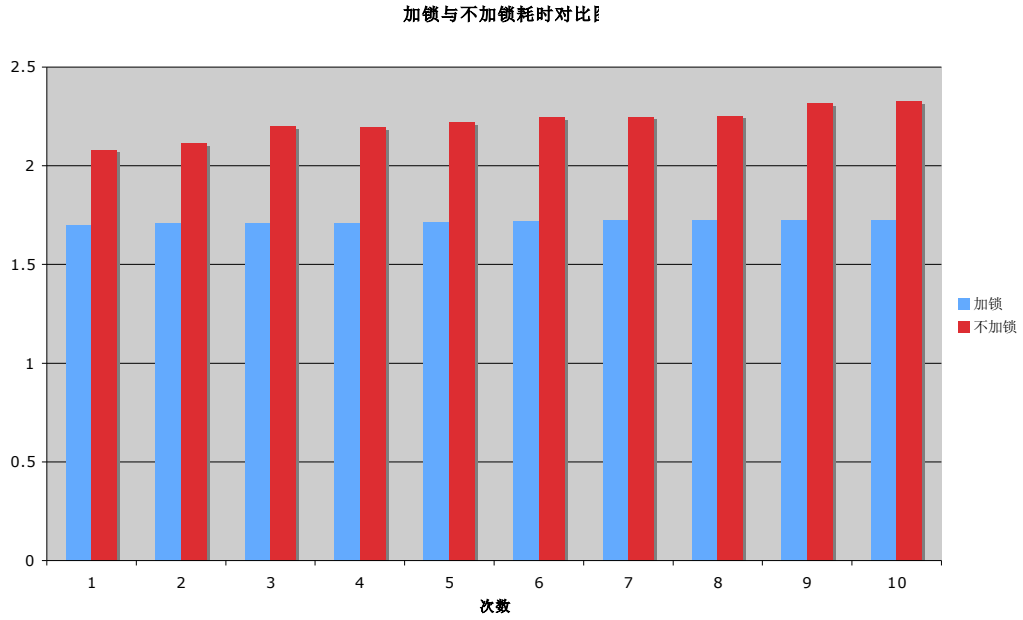

图 3 加锁与不加锁耗时对比图

其结果再一次大跌眼镜，可能有些人就会越来越糊涂了，怎么不加锁的效率反而更低呢？将在针对 Cache 的优化一节中细细分析其具体原因。

在实际测试过程中，不加锁的三线程方案非常不稳定，有时所花费的时间相差 4 倍多。

要提高并行程序的性能，在设计时就需要在较少同步和较多同步之间寻求折中。同步太少会导致错误的结果，同步太多又会导致效率过低。尽量使用私有锁，降低锁的粒度。无锁设计既有优点也有缺点，无锁方案能充分提高效率，但使得设计更加复杂，维护操作困难，不得不借助其他机制来保证程序的正确性。

针对 Cache 的优化

在串行程序设计过程中，为了节约带宽或者存储空间，比较直接的方法，就是对数据结构做一些针对性的设计，将数据压缩(pack)的更紧凑，减少数据的移动，以此来提高程序的性能。但在多核多线程程序中，这种方法往往有时会适得其反。

数据不仅在执行核和存储器之间移动，还会在执行核之间传输。根据数据相关性，其中有两种读写模式会涉及到数据的移动：写后读和写后写，因为这两种模式会引发数据的竞争，表面上是并行执行，但实际只能串行执行，进而影响到性能。

处理器交换的最小单元是 cache 行，或称 cache 块。在多核体系中，对于不共享 cache 的架构来说，两个独立的 cache 在需要读取同一 cache 行时，会共享该 cache 行，如果在其

中一个 cache 中，该 cache 行被写入，而在另一个 cache 中该 cache 行被读取，那么即使读写的地址不相交，也需要在这两个 cache 之间移动数据，这就被称为 cache 伪共享，导致执行核必须在存储总线上来回传递这个 Cache 行，这种现象被称为“乒乓效应”。

同样地，当两个线程写入同一个 cache 的不同部分时，也会互相竞争该 cache 行，也就是写后写的问题。上文曾提到，不加锁的方案反而比加锁的方案更慢，就是互相竞争 cache 的原因。

在 X86 机器上，某些处理器的一个 cache 行是 64 字节，具体可以参看 Intel 的参考手册。

既然不加锁三线程方案的瓶颈在于 Cache，那么让 apple 的两个成员 a 和 b 位于不同的 cache 行中，效率会有所提高吗？

修改后的代码片断如下：

```
#ifdef __cplusplus
extern "C"
{
#endif

#include <stdio.h>
#include <sys/types.h>
#include <sys/time.h>
#include <pthread.h>
#include <unistd.h>

#define ORANGE_MAX_VALUE      1000000
#define APPLE_MAX_VALUE       100000000
#define MSECOND                1000000

struct apple
{
   unsigned long long a;
   char c[128];
   unsigned long long b;
};

struct orange
{
   int a[ORANGE_MAX_VALUE];
   int b[ORANGE_MAX_VALUE];
};


void* addx(void* x)
{
   unsigned long long sum=0;
   struct timeval tpstart,tpend;
```

```
34.         float  timeuse;
35.
36.         gettimeofday(&tpstart,NULL);
37.
38.         for(sum=0;sum<APPLE_MAX_VALUE;sum++)
39.         {
40.             ((struct apple *)x)->a += sum;
41.         }
42.
43.         gettimeofday(&tpend,NULL);
44.
45.         timeuse=MSECOND*(tpend.tv_sec-tpstart.tv_sec)+tpend.tv_usec-tpstart.tv_usec;
46.         timeuse/=MSECOND;
47.         printf("addx thread:%x,Used Time:%f\n",pthread_self(),timeuse);
48.
49.         return NULL;
50.     }
51.
52.     void* addy(void* y)
53.     {
54.         unsigned long long sum=0;
55.         struct timeval tpstart,tpend;
56.         float  timeuse;
57.
58.         gettimeofday(&tpstart,NULL);
59.
60.         for(sum=0;sum<APPLE_MAX_VALUE;sum++)
61.         {
62.             ((struct apple *)y)->b += sum;
63.         }
64.
65.         gettimeofday(&tpend,NULL);
66.
67.         timeuse=MSECOND*(tpend.tv_sec-tpstart.tv_sec)+tpend.tv_usec-tpstart.tv_usec;
68.         timeuse/=MSECOND;
69.         printf("addy thread:%x,Used Time:%f\n",pthread_self(),timeuse);
70.
71.         return NULL;
72.     }
73.
74.     int main (int argc, const char * argv[]) {
75.             // insert code here...
76.         struct apple test;
77.         struct orange test1={{0},{0}};
78.         pthread_t ThreadA,ThreadB;
79.
80.         unsigned long long sum=0,index=0;
81.         struct timeval tpstart,tpend;
82.         float  timeuse;
83.
84.         test.a= 0;
85.         test.b= 0;
86.
87.         gettimeofday(&tpstart,NULL);
88.
```

```
89.         pthread_create(&ThreadA,NULL,addx,&test);
90.         pthread_create(&ThreadB,NULL,addy,&test);
91.
92.         for(index=0;index<ORANGE_MAX_VALUE;index++)
93.         {
94.            sum+=test1.a[index]+test1.b[index];
95.         }
96.
97.         pthread_join(ThreadA,NULL);
98.         pthread_join(ThreadB,NULL);
99.
100.        gettimeofday(&tpend,NULL);
101.
102.        timeuse=MSECOND*(tpend.tv_sec-tpstart.tv_sec)+tpend.tv_usec-tpstart.tv_usec;
103.        timeuse/=MSECOND;
104.        printf("main thread:%x,Used Time:%f\n",pthread_self(),timeuse);
105.
106.        printf("a = %llu,b = %llu,sum=%llu\n",test.a,test.b,sum);
107.
108.        return 0;
109.    }
110.
111.    #ifdef __cplusplus
112.    }
113.    #endif
```

测量结果如下图所示：

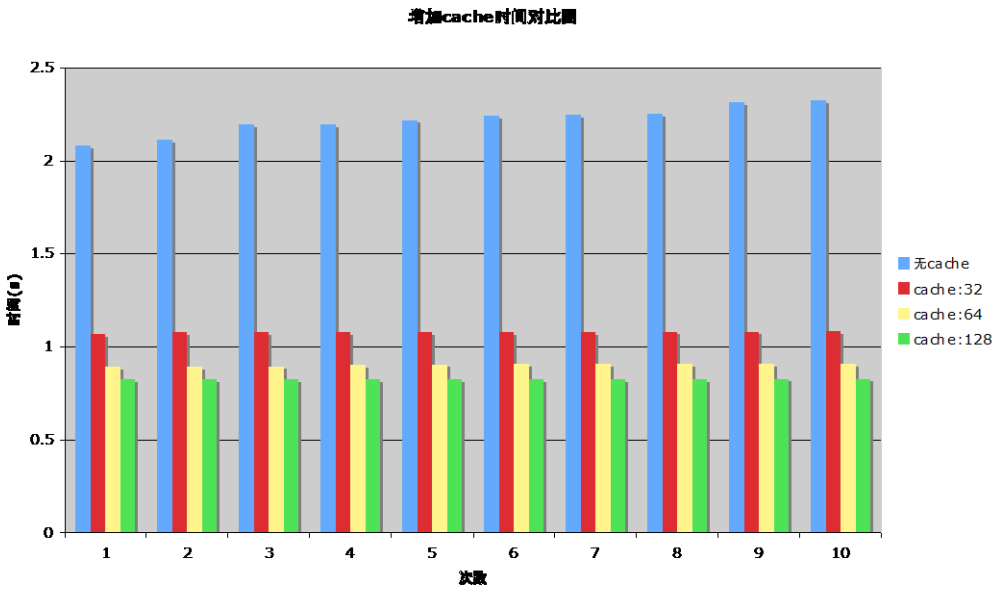

图 4 增加 cache 时间耗时对比图

小小的一行代码，尽然带来了如此高的收益，不难看出，我们是用空间来换时间。当然读者也可以采用更简便的方法：__attribute__((__aligned__(L1_CACHE_BYTES)))来确定 cache 的大小。

如果对加锁三线程方案中的 apple 数据结构也增加一行类似功能的代码，效率也是否会提升呢?

```
1.  #ifdef __cplusplus
2.  extern "C"
3.  {
4.  #endif
5.
6.  #include <stdio.h>
7.  #include <sys/types.h>
8.  #include <sys/time.h>
9.  #include <pthread.h>
10. #include <unistd.h>
11.
12. #define ORANGE_MAX_VALUE      1000000
13. #define APPLE_MAX_VALUE       100000000
14. #define MSECOND               1000000
15.
16. struct apple
17. {
18.     unsigned long long a;
19.     char c[128];
20.     unsigned long long b;
21.     pthread_rwlock_t rwLock;
22. };
23.
24. struct orange
25. {
26.     int a[ORANGE_MAX_VALUE];
27.     int b[ORANGE_MAX_VALUE];
28. };
29.
30. void* addx(void* x)
31. {
32.     unsigned long long sum=0;
33.     struct timeval tpstart,tpend;
34.     float  timeuse;
35.
36.     gettimeofday(&tpstart,NULL);
37.     pthread_rwlock_wrlock(&((struct apple *)x)->rwLock);
38.     for(sum=0;sum<APPLE_MAX_VALUE;sum++)
39.     {
40.         ((struct apple *)x)->a += sum;
41.     }
42.     pthread_rwlock_unlock(&((struct apple *)x)->rwLock);
43.     gettimeofday(&tpend,NULL);
44.
45.     timeuse=MSECOND*(tpend.tv_sec-tpstart.tv_sec)+tpend.tv_usec-tpstart.tv_usec;
46.     timeuse/=MSECOND;
47.     printf("addx thread:%x,Used Time:%f\n",pthread_self(),timeuse);
48.
49.     return NULL;
50. }
51.
```

```
52.     void* addy(void* y)
53.     {
54.         unsigned long long sum=0;
55.         struct timeval tpstart,tpend;
56.         float  timeuse;
57.
58.         gettimeofday(&tpstart,NULL);
59.         pthread_rwlock_wrlock(&((struct apple *)y)->rwLock);
60.         for(sum=0;sum<APPLE_MAX_VALUE;sum++)
61.         {
62.             ((struct apple *)y)->b += sum;
63.         }
64.         pthread_rwlock_unlock(&((struct apple *)y)->rwLock);
65.         gettimeofday(&tpend,NULL);
66.
67.         timeuse=MSECOND*(tpend.tv_sec-tpstart.tv_sec)+tpend.tv_usec-tpstart.tv_usec;
68.         timeuse/=MSECOND;
69.         printf("addy thread:%x,Used Time:%f\n",pthread_self(),timeuse);
70.
71.         return NULL;
72.     }
73.
74.     int main (int argc, const char * argv[]) {
75.             // insert code here...
76.         struct apple test;
77.         struct orange test1={{0},{0}};
78.         pthread_t ThreadA,ThreadB;
79.
80.         unsigned long long sum=0,index=0;
81.         struct timeval tpstart,tpend;
82.         float  timeuse;
83.
84.         test.a= 0;
85.         test.b= 0;
86.
87.         gettimeofday(&tpstart,NULL);
88.
89.         pthread_rwlock_init(&test.rwLock,NULL);
90.
91.         pthread_create(&ThreadA,NULL,addx,&test);
92.         pthread_create(&ThreadB,NULL,addy,&test);
93.
94.         for(index=0;index<ORANGE_MAX_VALUE;index++)
95.         {
96.             sum+=test1.a[index]+test1.b[index];
97.         }
98.
99.         pthread_join(ThreadA,NULL);
100.        pthread_join(ThreadB,NULL);
101.
102.        gettimeofday(&tpend,NULL);
103.
104.        timeuse=MSECOND*(tpend.tv_sec-tpstart.tv_sec)+tpend.tv_usec-tpstart.tv_usec;
105.        timeuse/=MSECOND;
106.        printf("main thread:%x,Used Time:%f\n",pthread_self(),timeuse);
```

```
107.
108.        printf("a = %llu,b = %llu,sum=%llu\n",test.a,test.b,sum);
109.
110.        return 0;
111.    }
112.
113.    #ifdef __cplusplus
114.    }
115.    #endif
```

性能不会有所提升，其原因是加锁的三线程方案效率低下的原因不是 Cache 失效造成的，而是那把锁。

在多核和多线程程序设计过程中，要全盘考虑多个线程的访存需求，不要单独考虑一个线程的需求。在选择并行任务分解方法时，要综合考虑访存带宽和竞争问题，将不同处理器和不同线程使用的数据放在不同的 Cache 行中，将只读数据和可写数据分离开。

CPU 亲和力

CPU 亲和力可分为两大类：软亲和力和硬亲和力。

Linux 内核进程调度器天生就具有被称为 CPU 软亲和力（affinity） 的特性，这意味着进程通常不会在处理器之间频繁迁移。这种状态正是我们希望的，因为进程迁移的频率小就意味着产生的负载小。但不代表不会进行小范围的迁移。

CPU 硬亲和力是指进程固定在某个处理器上运行，而不是在不同的处理器之间进行频繁的迁移。这样不仅改善了程序的性能，还提高了程序的可靠性。

从以上不难看出，在某种程度上硬亲和力比软亲和力具有一定的优势。但在内核开发者不断的努力下，2.6 内核软亲和力的缺陷已经比 2.4 的内核有了很大的改善。

在双核机器上，针对两线程的方案，如果将计算 apple 的线程绑定到一个 CPU 上，将计算 orange 的线程绑定到另外一个 CPU 上，效率是否会有所提高呢?

程序如下：

```
1.      #ifdef __cplusplus
2.      extern "C"
3.      {
4.      #endif
5.
6.      #include <stdio.h>
7.      #include <stdlib.h>
8.      #include <sys/types.h>
9.      #include <linux/unistd.h>
10.     #include <sys/sysinfo.h>
11.     #include <sys/time.h>
12.     #define __USE_GNU
```

```
13.  #include <pthread.h>
14.  #include <sched.h>
15.  #include <ctype.h>
16.  #include <unistd.h>
17.  #include <string.h>
18.  #include <errno.h>
19.
20.  #define gettid() syscall(__NR_gettid)
21.  //static inline _syscall0(pid_t,gettid)
22.
23.  #define ORANGE_MAX_VALUE      1000000
24.  #define APPLE_MAX_VALUE       100000000
25.  #define MSECOND               1000000
26.
27.  struct apple
28.  {
29.     unsigned long long a;
30.     unsigned long long b;
31.  };
32.
33.  struct orange
34.  {
35.     int a[ORANGE_MAX_VALUE];
36.     int b[ORANGE_MAX_VALUE];
37.  };
38.
39.  int cpu_nums = 0;
40.
41.  inline int set_cpu(int i)
42.  {
43.
44.     cpu_set_t mask;
45.
46.     CPU_ZERO(&mask);
47.
48.     if(2 <= cpu_nums)
49.     {
50.        CPU_SET(i,&mask);
51.
52.        if(-1 == sched_setaffinity(gettid(),sizeof(&mask),&mask))
53.        {
54.           return -1;
55.        }
56.     }
57.     return 0;
58.  }
59.
60.  void* add(void* x)
61.  {
62.     unsigned long long sum=0;
63.     struct timeval tpstart,tpend;
64.     float  timeuse;
65.
66.     if(-1 == set_cpu(1))
67.     {
```

```
68.             return NULL;
69.         }
70.
71.         gettimeofday(&tpstart,NULL);
72.
73.         for(sum=0;sum<APPLE_MAX_VALUE;sum++)
74.         {
75.             ((struct apple *)x)->a += sum;
76.             ((struct apple *)x)->b += sum;
77.         }
78.
79.         gettimeofday(&tpend,NULL);
80.
81.         timeuse=MSECOND*(tpend.tv_sec-tpstart.tv_sec)+tpend.tv_usec-tpstart.tv_usec;
82.         timeuse/=MSECOND;
83.         printf("add thread:%x,Used Time:%f\n",pthread_self(),timeuse);
84.
85.         return NULL;
86.     }
87.
88.     int main (int argc, const char * argv[]) {
89.         // insert code here...
90.         struct apple test;
91.         struct orange test1={{0},{0}};
92.         pthread_t ThreadA;
93.
94.         unsigned long long sum=0,index=0;
95.         struct timeval tpstart,tpend;
96.         float  timeuse;
97.
98.         test.a= 0;
99.         test.b= 0;
100.
101.        cpu_nums = sysconf(_SC_NPROCESSORS_CONF);
102.
103.        if(-1 == set_cpu(0))
104.        {
105.            return -1;
106.        }
107.
108.        gettimeofday(&tpstart,NULL);
109.
110.        pthread_create(&ThreadA,NULL,add,&test);
111.
112.
113.        for(index=0;index<ORANGE_MAX_VALUE;index++)
114.        {
115.            sum+=test1.a[index]+test1.b[index];
116.        }
117.
118.        pthread_join(ThreadA,NULL);
119.
120.        gettimeofday(&tpend,NULL);
121.
122.        timeuse=MSECOND*(tpend.tv_sec-tpstart.tv_sec)+tpend.tv_usec-tpstart.tv_usec;
```

```
123.        timeuse/=MSECOND;
124.        printf("main thread:%x,Used Time:%f\n",pthread_self(),timeuse);
125.
126.        printf("a = %llu,b = %llu,sum=%llu\n",test.a,test.b,sum);
127.
128.        return 0;
129.    }
130.
131.    #ifdef __cplusplus
132.    }
133.    #endif
```

测量结果为：

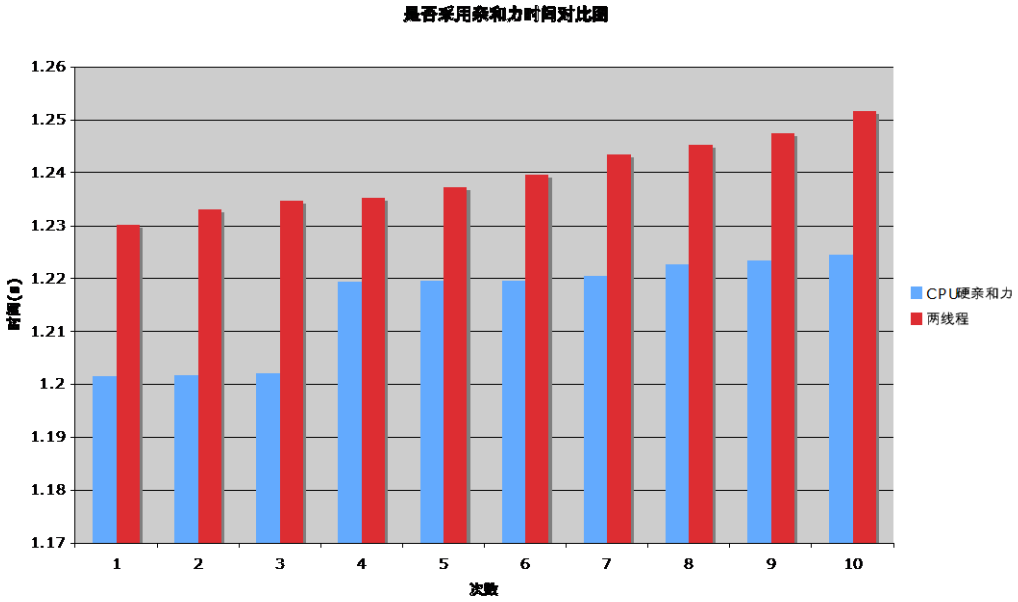

图 5 是否采用硬亲和力时间对比图(两线程)

其测量结果正是我们所希望的，但花费的时间还是比单线程的多，其原因与上面分析的类似。

进一步分析不难发现，样例程序大部分时间都消耗在计算 apple 上，如果将计算 a 和 b 的值，分布到不同的 CPU 上进行计算，同时考虑 Cache 的影响，效率是否也会有所提升呢？

```
1.      #ifdef __cplusplus
2.      extern "C"
3.      {
4.      #endif
5.
6.      #include <stdio.h>
7.      #include <stdlib.h>
8.      #include <sys/types.h>
9.      #include <linux/unistd.h>
10.     #include <sys/sysinfo.h>
```

```
#include <sys/time.h>
#define __USE_GNU
#include <pthread.h>
#include <sched.h>
#include <ctype.h>
#include <unistd.h>
#include <string.h>
#include <errno.h>

//static inline _syscall0(pid_t,gettid)
#define gettid() syscall(__NR_gettid)

#define ORANGE_MAX_VALUE      1000000
#define APPLE_MAX_VALUE       100000000
#define MSECOND               1000000

struct apple
{
   unsigned long long a;
   char c[128];
   unsigned long long b;
};

struct orange
{
   int a[ORANGE_MAX_VALUE];
   int b[ORANGE_MAX_VALUE];
};

int cpu_nums = 0;

inline int set_cpu(int i)
{
   cpu_set_t mask;

   CPU_ZERO(&mask);

   if(2 <= cpu_nums)
   {
      CPU_SET(i,&mask);

      if(-1 == sched_setaffinity(gettid(),sizeof(&mask),&mask))
      {
                  printf("set affinity error\r\n");
         return -1;
      }
   }
   return 0;
}

void* addx(void* x)
{
   unsigned long long sum=0;
   struct timeval tpstart,tpend;
   float  timeuse;
```

```
66.
67.         if(-1 == set_cpu(1))
68.         {
69.             return NULL;
70.         }
71.
72.         gettimeofday(&tpstart,NULL);
73.
74.         for(sum=0;sum<APPLE_MAX_VALUE;sum++)
75.         {
76.             ((struct apple *)x)->a += sum;
77.         }
78.
79.         gettimeofday(&tpend,NULL);
80.
81.         timeuse=MSECOND*(tpend.tv_sec-tpstart.tv_sec)+tpend.tv_usec-tpstart.tv_usec;
82.         timeuse/=MSECOND;
83.         printf("addx thread:%x,Used Time:%f\n",/*pthread_self()*/gettid(),timeuse);
84.
85.         return NULL;
86.     }
87.
88.     void* addy(void* y)
89.     {
90.         unsigned long long sum=0;
91.         struct timeval tpstart,tpend;
92.         float  timeuse;
93.
94.         if(-1 == set_cpu(0))
95.         {
96.             return NULL;
97.         }
98.
99.         gettimeofday(&tpstart,NULL);
100.
101.         for(sum=0;sum<APPLE_MAX_VALUE;sum++)
102.         {
103.             ((struct apple *)y)->b += sum;
104.         }
105.
106.         gettimeofday(&tpend,NULL);
107.
108.         timeuse=MSECOND*(tpend.tv_sec-tpstart.tv_sec)+tpend.tv_usec-tpstart.tv_usec;
109.         timeuse/=MSECOND;
110.         printf("addy thread:%x,Used Time:%f\n",/*pthread_self()*/gettid(),timeuse);
111.
112.         return NULL;
113.     }
114.
115.     int main (int argc, const char * argv[]) {
116.         // insert code here...
117.         struct apple test;
118.         struct orange test1={{0},{0}};
119.         pthread_t ThreadA,ThreadB;
120.
```

```
121.        unsigned long long sum=0,index=0;
122.        struct timeval tpstart,tpend;
123.        float  timeuse;
124.
125.        test.a= 0;
126.        test.b= 0;
127.
128.        cpu_nums = sysconf(_SC_NPROCESSORS_CONF);
129.
130.        if(-1 == set_cpu(0))
131.        {
132.           return 1;
133.        }
134.
135.        gettimeofday(&tpstart,NULL);
136.
137.        pthread_create(&ThreadA,NULL,addx,&test);
138.        pthread_create(&ThreadB,NULL,addy,&test);
139.
140.        for(index=0;index<ORANGE_MAX_VALUE;index++)
141.        {
142.           sum+=test1.a[index]+test1.b[index];
143.        }
144.
145.        pthread_join(ThreadA,NULL);
146.        pthread_join(ThreadB,NULL);
147.
148.        gettimeofday(&tpend,NULL);
149.
150.        timeuse=MSECOND*(tpend.tv_sec-tpstart.tv_sec)+tpend.tv_usec-tpstart.tv_usec;
151.        timeuse/=MSECOND;
152.        printf("main thread:%x,Used Time:%f\n",/*pthread_self()*/gettid(),timeuse);
153.
154.        printf("a = %llu,b = %llu,sum=%llu\n",test.a,test.b,sum);
155.
156.        return 0;
157.     }
158.
159.     #ifdef __cplusplus
160.     }
161.     #endif
```

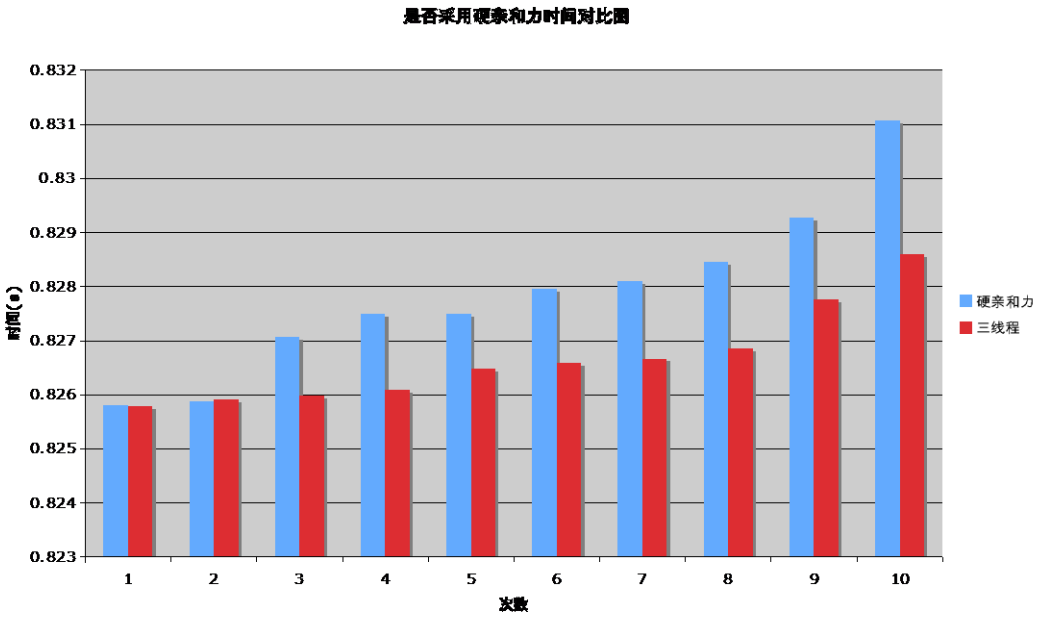

图 6 是否采用硬亲和力时间对比图(三线程)

从时间上观察，设置亲和力的程序所花费的时间略高于采用 Cache 的三线程方案。由于考虑了 Cache 的影响，排除了一级缓存造成的瓶颈，多出的时间主要消耗在系统调用及内核上，可以通过 time 命令来验证：

```
#time ./unlockcachemultiprocess
real  0m0.834s    user  0m1.644s    sys   0m0.004s
#time ./affinityunlockcacheprocess
real  0m0.875s    user  0m1.716s    sys   0m0.008s
```

通过设置 CPU 亲和力来利用多核特性，为提高应用程序性能提供了捷径。同时也是一把双刃剑，如果忽略负载均衡、数据竞争等因素，效率将大打折扣，甚至带来事倍功半的结果。

在进行具体的设计过程中，需要设计良好的数据结构和算法，使其适合于应用的数据移动和处理器的性能特性。

总结

根据以上分析及实验，对所有改进方案的测试时间做一个综合对比，如下图所示：

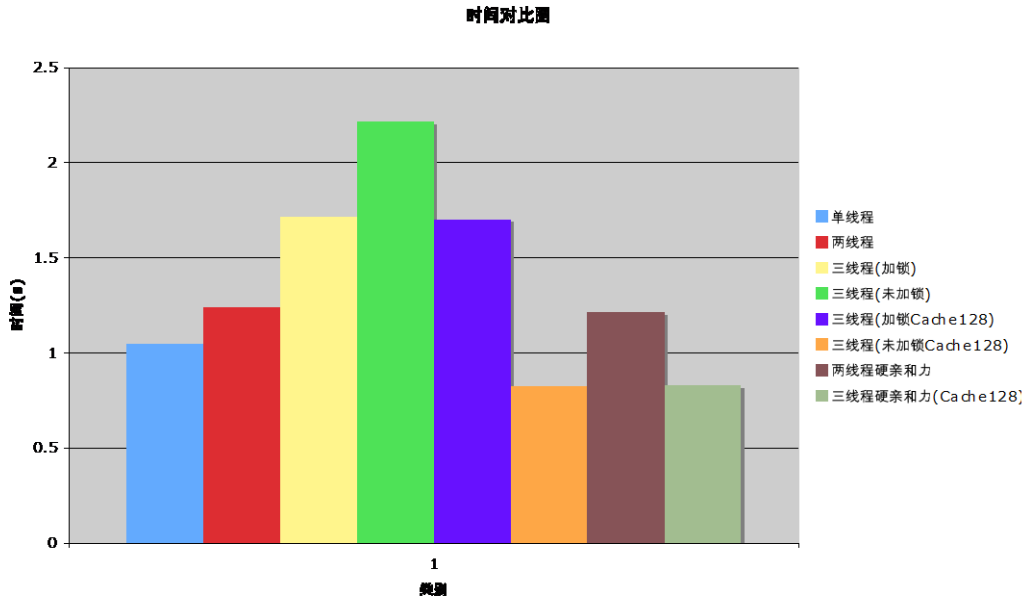

图 7 各方案时间对比图

单线程原始程序平均耗时：1.049046s，最慢的不加锁三线程方案平均耗时：2.217413s，最快的三线程(Cache 为 128)平均耗时：0.826674s，效率提升约 26%。当然，还可以进一步优化，让效率得到更高的提升。

从上图不难得出结论：采用多核多线程并行设计方案，能有效提高性能，但如果考虑不全面，如忽略带宽、数据竞争及数据同步不当等因素，效率反而降低，程序执行越来越慢。

如果抛开本文开篇时的限制，采用上文曾提到的另外一种数据分解模型，同时结合硬亲和力对样例程序进行优化，测试时间为 0.54s，效率提升了 92%。

软件优化是一个贯穿整个软件开发周期，从开始设计到最终完成一直进行的连续过程。在优化前，需要找出瓶颈和热点所在。正如最伟大的 C 语言大师 Rob Pike 所说：

如果你无法断定程序会在什么地方耗费运行时间，瓶颈经常出现在意想不到的地方，所以别急于胡乱找个地方改代码，除非你已经证实那儿就是瓶颈所在。

将这句话送给所有的优化人员，和大家共勉。

参考资料：

1. Shameem Akhter and Jason Roberts. 李宝峰，富弘毅，李韬译.《多核程序设计技术》，电子工业出版社，2007

2. Richard Gerber, Kevin B.Smith, Aart J.C.Bik and Xinmin Tian. 王涛，单久龙，孙广中译.《软件优化技术》，电子工业出版社，2007
3. Eric S. Raymond. 姜宏，何源，蔡晓俊译.《UNIX 编程艺术》， 电子工业出版社，2006
4. Rebort Love. 《CPU Affinity》http://www.linuxjournal.com/article/6799
5. Eli Dow. 《管理处理器的亲和性(affinity)》，http://www.ibm.com/developerworks/cn/linux/l-affinity.html

工具篇

利用 Oprofile 对多核多线程进行性能分析[2]

工欲善其事，必先利其器

---墨子

性能分析工具简介

在对应用程序不断调优的过程中，除了制定完备的测试基准（Benchmark）外，还需要一把直中要害的利器——性能分析工具。

根据工具的复杂度和所提供的功能，可以将性能工具分为两个层次：

1. 基本的计时工具

   在普通生活中，秒表是最简单的计时工具。根据该思想，可以将计时函数放在代码的任意位置并多次调用，这样就可以测量出整个应用或者某一部分的运行时间。这种分析方法不够精细，误差大。

2. 软件分析工具

   目前，主要有两种不同类型的软件分析工具：采样和插桩。

   - 采样型分析工具

     主要通过周期性中断，来纪录相关的性能信息，如处理器指令指针、线程 id、处理器 id 和事件计数器等。这种方法开销小，精确度高。在 Linux 系统中，比较常见的有 Oprofile 和 Intel VTune 性能分析器等。

   - 插桩型分析工具

     即可以使用直接的二进制插桩，也可以通过编译器在应用中插入分析代码。这种方式与自己在应用中增加计时函数类似，同时带来的开销大，但提供了更多的功能，如调用树，调用次数和函数开销等。在 Linux 系统中，比较常见的有 gprof 和 Intel VTune 性能分析器等。

本文将利用采样型工具 Oprofile，对多核多线程程序进行性能分析，起一个抛砖引玉的作用。

衡量性能收益的方法

[2] 最初由《程序员》2009/05 发表，其网址是 http://www.csdn.net

随着科学技术的不断发展，计算机系统结构朝着多核的方向发展，从而将并发编程推到了聚光灯下，但如何去衡量并行程序设计所带来的性能收益呢?

不得不想起 1967 年 Gene Amdahl 所做出的杰出贡献，他提出的 Amdahl 定律能够计算出并行程序相对于最优串行算法在性能提升上的理论最大值。

Amdahl 定律

$$加速比=\frac{1}{S+(1-S)/n+H(n)}$$

其中，S 表示执行程序中串行部分的比例，n 表示处理器核的数量，$H(n)$表示系统开销。

由于 Amdahl 定律本身做出了几个假设，但这些假设在现实世界中又不一定是正确的，因此使计算机界心灰意冷了很多年，认为根据 Amdahl 定律，开发更大的并行性所带来的性能收益可能是微不足道的，一直到 Gustafson 定律的出现，才改变了现状。

在 Sandia 实验室工作的基础上，E.Barsis 提出了 Gustafson 定律:

$$扩展加速比=N+(1-N)*S$$

其中，S 表示执行程序中串行部分的比例，N 表示处理器核的数量。

幸运的是，Shi 于 1996 年证明 Gustafson 定律和 Amdahl 定律是等效的。

Oprofile 工作原理简介

根据 CPU 系统结构的不同，Oprofile 支持两种采样方式：基于事件(Event Based)的采样和基于时间(Time Based)的采样。

如果 CPU 内部存在性能计数寄存器，则 Oprofile 基于事件采样，记录特定事件（如分支预测事件）发生的次数，当达到设定的定值时就采样一次。反之，则基于时间采样，主要是借助于操作系统的时钟中断机制，每当时钟中断发生时就采样一次。不难看出，基于时间的采样方式，要求被测程序不能屏蔽中断，其精度也低于事件采样。

对于 x86 体系结构，不同型号的 CPU，采样方式也不同，具体细节如下表所示:

处理器	CPU_TYPE	采样方式
Athlon	i386/athlon	Event Based

Pentium Pro	i386/ppro	Event Based
Pentium II	i386/pii	Event Based
Pentium III	i386/piii	Event Based
Pentium M (P6 core)	i386/p6_mobile	Event Based
Pentium 4 (non-HT)	i386/p4	Event Based
Pentium 4 (HT)	i386/p4-ht	Event Based
Dual Core	Timer	Time Based
Core 2 Duo	Timer	Time Based

表 4 x86 各处理器采样方式

Oprofile 主要分为两部分，其中一部分是内核模块(oprofile.ko)，另外一部分是用户空间的守护进程(oprofiled)。前者主要负责访问性能计数寄存器或者注册基于时间采样的函数，并将采样结果置于内核的缓冲区中。后者在后台运行，负责从内核空间收集数据，并写入采样文件中，其交互流程如图所示：

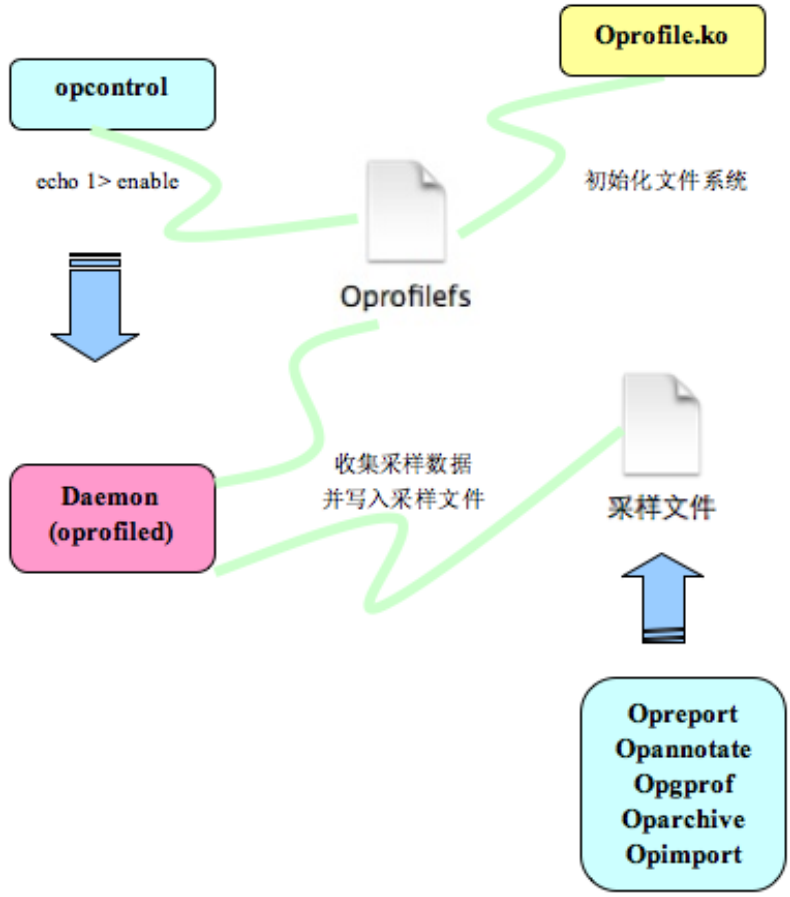

图 8oprofile 交互流程图

安装 Oprofile

oprofile.ko 内核模块已经被集成到 linux 2.6 内核中，所以只需要安装前端工具，可以从 oprofile 官方网站下载源码来进行安装，当前最新版本为 0.9.4。

该源码包是通过 automake 和 autoconf 生成的 makefile，所以只需要以 root 用户进入 oprofile 目录，运行./configure、make 及 make install 来完成所有的安装。 在目前大部分发行版中， 安装时可能缺少 popt 库，请另行下载安装。

Oprofile 工具链提供了 6 大工具，供用户控制 oprofile 和分析样本。其中 opcontrol 是一个 bash 脚本程序，主要用来控制 oprofile 的启动、暂停及设置，其他工具主要是对采样数据进行分析。

观察本书开始的样例程序， S（串行部分的比例）所占比例非常小，基本为 0， 根据 Amdahl 定律，在双核系统（n=2）上，则加速比为：加速比＝1/(0+1/2+1%)＝1.96，也就是说效率可以提高 96%。

样例程序运行时间为：1.049046s，那么经过优化后，能不能达到理论值呢？请随着本

文开始。

追踪热点

既然要充分利用双核的特性，则不得不对样例程序进行改造，进行并行程序设计。

可以将应用程序看成是众多相互依赖的任务的集合。将应用程序划分成多个独立的任务，并确定这些任务之间的相互依赖关系，这个过程称为分解（Decomosition）。分解问题的方式主要有三种：任务分解、数据分解和数据流分解。

运用任务分解的方法，不难发现计算 apple 的值和计算 orange 的值，属于完全不相关的两个操作，因此可以并行。

改造后的两线程程序：

```
1.      #ifdef __cplusplus
2.      extern "C"
3.      {
4.      #endif
5.
6.      #include <stdio.h>
7.      #include <sys/types.h>
8.      #include <sys/time.h>
9.      #include <pthread.h>
10.     #include <unistd.h>
11.
12.     #define ORANGE_MAX_VALUE     1000000
13.     #define APPLE_MAX_VALUE      100000000
14.     #define MSECOND              1000000
15.
16.     struct apple
17.     {
18.        unsigned long long a;
19.        unsigned long long b;
20.     };
21.
22.     struct orange
23.     {
24.        int a[ORANGE_MAX_VALUE];
25.        int b[ORANGE_MAX_VALUE];
26.     };
27.
28.     void* add(void* x)
29.     {
30.        unsigned long long sum=0;
31.        struct timeval tpstart,tpend;
32.        float  timeuse=0;
33.
34.        gettimeofday(&tpstart,NULL);
35.
36.        for(sum=0;sum<APPLE_MAX_VALUE;sum++)
```

```
37.         {
38.             ((struct apple *)x)->a += sum;
39.             ((struct apple *)x)->b += sum;
40.         }
41.
42.         gettimeofday(&tpend,NULL);
43.
44.         timeuse=MSECOND*(tpend.tv_sec-tpstart.tv_sec)+tpend.tv_usec-tpstart.tv_usec;
45.         timeuse/=MSECOND;
46.         printf("add thread:%x,Used Time:%f\n",pthread_self(),timeuse);
47.
48.         return NULL;
49.     }
50.
51.     int main (int argc, const char * argv[]) {
52.             // insert code here...
53.         struct apple test;
54.         struct orange test1={{0},{0}};
55.         pthread_t ThreadA;
56.
57.         unsigned long long sum=0,index=0;
58.         struct timeval tpstart,tpend;
59.         float  timeuse;
60.
61.         test.a= 0;
62.         test.b= 0;
63.
64.         gettimeofday(&tpstart,NULL);
65.
66.         pthread_create(&ThreadA,NULL,add,&test);
67.
68.
69.         for(index=0;index<ORANGE_MAX_VALUE;index++)
70.         {
71.             sum=test1.a[index]+test1.b[index];
72.         }
73.
74.         pthread_join(ThreadA,NULL);
75.
76.         gettimeofday(&tpend,NULL);
77.
78.         timeuse=MSECOND*(tpend.tv_sec-tpstart.tv_sec)+tpend.tv_usec-tpstart.tv_usec;
79.         timeuse/=MSECOND;
80.         printf("main thread:%x,Used Time:%f\n",pthread_self(),timeuse);
81.
82.         printf("a = %llu,b = %llu,sum=%llu\n",test.a,test.b,sum);
83.
84.         return 0;
85.     }
86.
87.     #ifdef __cplusplus
88.     }
89.     #endif
```

现在通过 oprofile 来对多线程程序进行性能分析，收集热点信息。oprofile 的功能非常强大，可以对每个线程进行单独采样，也可以对每个 CPU 单独采样，这些都是通过 opcontrol 的--separate 选项来完成的。separate 选项值含义如下：

separate 选项值	含义
None	默认值
Lib	对每个应用程序的所有 lib 进行采样
Kernel	对每个应用程序的内核及内核模块采样
Thread	对每个线程或任务采样
cpu	对每个 CPU 进行采样
all	以上所有选项的功能

表 5separate 选项值含义

操作步骤如下：

```
# opcontrol –init
# opcontrol --separate=thread --no-vmlinux
# opcontrol --start
Using 2.6+ OProfile kernel interface.
Using log file /var/lib/oprofile/samples/oprofiled.log
Daemon started.
Profiler running.
# ./twothreadprocess
add thread:b7dc7b90,Used Time:1.225483
main thread:b7dc8ad0,Used Time:1.230151
a = 4999999950000000,b = 4999999950000000,sum=0
# opcontrol --shutdown
Stopping profiling.
Killing daemon.
```

识别出并行程序中的重载

运行 opreporte，查看采样结果：

```
# opreport -l ./twothreadprocess
CPU: CPU with timer interrupt, speed 0 MHz (estimated)
```

Profiling through timer interrupt

Processes with a thread ID of 5237

Processes with a thread ID of 5238

samples	%	samples	%	symbol name
30	100.000	0	0	main
0	0	343	100.000	add

运行时间为：1.230151s，与理论值相差甚远，反而运行时间越来越长，原因何在呢?

通过分析结果，不难看出 add 线程负载非常重，而 main 负载较轻，负载不均衡，因此重点分析对象为 add 线程。根据多线程数据分解的原理，将计算 apple 值的过程一分为二，main 线程也参与部分计算。由于都是循环，为了使负载均衡，则 add 线程里面应该循环(ORANGE_MAX_VALUE+APPLE_MAX_VALUE)/2 次。修改后的程序如下：

```
1.  #ifdef __cplusplus
2.  extern "C"
3.  {
4.  #endif
5.
6.  #include <stdio.h>
7.  #include <sys/types.h>
8.  #include <sys/time.h>
9.  #include <pthread.h>
10. #include <unistd.h>
11.
12. #define ORANGE_MAX_VALUE    1000000
13. #define APPLE_MAX_VALUE     100000000
14. #define MSECOND             1000000
15. #ifndef MIDLLE_VALUE
16.    #define MIDLLE_VALUE     49500000
17. #endif
18.
19. struct apple
20. {
21.    unsigned long long a;
22.    unsigned long long b;
23. };
24.
25. struct orange
26. {
27.    int a[ORANGE_MAX_VALUE];
28.    int b[ORANGE_MAX_VALUE];
29. };
30.
31. unsigned long long value[2][2];
32.
33. void* add(void* x)
34. {
35.    unsigned long long sum=0;
36.    unsigned long long temp1,temp2;
37.    struct timeval tpstart,tpend;
38.    float  timeuse=0;
```

```
39.
40.         gettimeofday(&tpstart,NULL);
41.
42.         temp1 = ((struct apple *)x)->a;
43.         temp2 = ((struct apple *)x)->b;
44.
45.         for(sum=MIDLLE_VALUE;sum<APPLE_MAX_VALUE;sum++)
46.         {
47.             temp1 += sum;
48.             temp2 += sum;
49.         }
50.
51.         value[0][0]=temp1;
52.         value[1][0]=temp2;
53.
54.         gettimeofday(&tpend,NULL);
55.
56.         timeuse=MSECOND*(tpend.tv_sec-tpstart.tv_sec)+tpend.tv_usec-tpstart.tv_usec;
57.         timeuse/=MSECOND;
58.         printf("add thread:%x,Used Time:%f\n",pthread_self(),timeuse);
59.
60.         return NULL;
61.     }
62.
63.     int main (int argc, const char * argv[]) {
64.         // insert code here...
65.         struct apple test;
66.         struct orange test1={{0},{0}};
67.         pthread_t ThreadA;
68.
69.         unsigned long long sum=0,index=0;
70.         unsigned long long temp1,temp2;
71.         struct timeval tpstart,tpend;
72.         float  timeuse;
73.
74.         test.a= 0;
75.         test.b= 0;
76.
77.         gettimeofday(&tpstart,NULL);
78.
79.         temp1 = test.a;
80.         temp2 = test.b;
81.
82.         pthread_create(&ThreadA,NULL,add,&test);
83.
84.         for(sum=0;sum<MIDLLE_VALUE;sum++)
85.         {
86.             temp1 += sum;
87.             temp2 += sum;
88.         }
89.
90.         value[0][1]=temp1;
91.         value[1][1]=temp2;
92.
93.         sum=0;
```

```
94.
95.         for(index=0;index<ORANGE_MAX_VALUE;index++)
96.         {
97.            sum=test1.a[index]+test1.b[index];
98.         }
99.
100.        pthread_join(ThreadA,NULL);
101.
102.        test.a = value[0][0]+value[0][1];
103.        test.b = value[1][0]+value[1][1];
104.
105.        gettimeofday(&tpend,NULL);
106.
107.        timeuse=MSECOND*(tpend.tv_sec-tpstart.tv_sec)+tpend.tv_usec-tpstart.tv_usec;
108.        timeuse/=MSECOND;
109.        printf("main thread:%x,Used Time:%f\n",pthread_self(),timeuse);
110.
111.        printf("a = %llu,b = %llu,sum=%llu\n",test.a,test.b,sum);
112.
113.        return 0;
114.    }
115.
116.
117.    #ifdef __cplusplus
118.    }
119.    #endif
```

重新运行 oprofile，再次收集采样数据，改造后的程序是否达到了负载均衡呢?

```
# opreport -l ./twothreadprofile
CPU: CPU with timer interrupt, speed 0 MHz (estimated)
Profiling through timer interrupt
Processes with a thread ID of 5242
Processes with a thread ID of 5243
samples  %      samples  %       symbol name
135   100.000     0       0            main
0          0        156   100.000      add
```

运行时间为：0.629821s，时间有了显著的提高，效率也提升了 66%，已经逼近理论值，但还是有一点差距。

继续分析，不难看出，负载还是没有达到均衡，main 线程还是有点轻。因此继续增大 MIDLLE_VALUE 的值到 53500000，再次运行 oprofile，看看效果如何：

```
# opreport -l ./twothreadprofilebig
CPU: CPU with timer interrupt, speed 0 MHz (estimated)
Profiling through timer interrupt
Processes with a thread ID of 5248
Processes with a thread ID of 5249
samples  %      samples  %       symbol name
```

```
145   100.000     0       0            main
0          0        146   100.000      add
```

运行时间为：0.540942s，比样例程序时间大约减少了一半，效率提升了 92%，已经非常接近于理论值。相信通过不断的微调，将会越来越接近理论值。同时在计算理论值时，所假设的系统开销比较低，仅仅为 1%。

由于 Linux 内核进程调度器天生具有 CPU 软亲和力（affinity）的特性，这就意味着进程通常不会在处理器之间频繁的迁移。在实际应用中，如果不存在数据竞争的影响，应用的不同部分分布到不同的 CPU 上运行，可能会带来更高的收益。

将样例程序的多线程版本绑定到不同的 CPU 上运行，效率会有所提升吗?

修改后的程序如下：

```
1.      #ifdef __cplusplus
2.      extern "C"
3.      {
4.      #endif
5.
6.      #include <stdio.h>
7.      #include <stdlib.h>
8.      #include <sys/types.h>
9.      #include <linux/unistd.h>
10.     #include <sys/sysinfo.h>
11.     #include <sys/time.h>
12.     #define __USE_GNU
13.     #include <pthread.h>
14.     #include <sched.h>
15.     #include <ctype.h>
16.     #include <unistd.h>
17.     #include <string.h>
18.     #include <errno.h>
19.
20.     #define gettid() syscall(__NR_gettid)
21.
22.     #define ORANGE_MAX_VALUE      1000000
23.     #define APPLE_MAX_VALUE       100000000
24.     #define MSECOND                1000000
25.     #ifndef MIDDLE_VALUE
26.        #define MIDDLE_VALUE       49500000
27.     #endif
28.
29.     struct apple
30.     {
31.        unsigned long long a;
32.        unsigned long long b;
33.     };
34.
35.     struct orange
36.     {
37.        int a[ORANGE_MAX_VALUE];
38.        int b[ORANGE_MAX_VALUE];
```

```
39.     };
40.
41.     int cpu_nums = 0;
42.     unsigned long long value[2][2];
43.
44.     inline int set_cpu(int i)
45.     {
46.
47.         cpu_set_t mask;
48.
49.         CPU_ZERO(&mask);
50.
51.         if(2 <= cpu_nums)
52.         {
53.             CPU_SET(i,&mask);
54.
55.             if(-1 == sched_setaffinity(gettid(),sizeof(&mask),&mask))
56.             {
57.                 return -1;
58.             }
59.         }
60.         return 0;
61.     }
62.
63.     void* add(void* x)
64.     {
65.         unsigned long long sum=0;
66.         unsigned long long temp1,temp2;
67.         struct timeval tpstart,tpend;
68.         float  timeuse;
69.
70.         if(-1 == set_cpu(1))
71.         {
72.             return NULL;
73.         }
74.
75.         gettimeofday(&tpstart,NULL);
76.
77.         temp1=((struct apple *)x)->a;
78.         temp2=((struct apple *)x)->b;
79.
80.         for(sum=MIDDLE_VALUE;sum<APPLE_MAX_VALUE;sum++)
81.         {
82.             temp1 += sum;
83.             temp2 += sum;
84.         }
85.
86.         value[0][0]=temp1;
87.         value[1][0]=temp2;
88.
89.         gettimeofday(&tpend,NULL);
90.
91.         timeuse=MSECOND*(tpend.tv_sec-tpstart.tv_sec)+tpend.tv_usec-tpstart.tv_usec;
92.         timeuse/=MSECOND;
93.         printf("add thread:%x,Used Time:%f\n",pthread_self(),timeuse);
```

```
94.
95.         return NULL;
96.     }
97.
98.     int main (int argc, const char * argv[]) {
99.         // insert code here...
100.        struct apple test;
101.        struct orange test1={{0},{0}};
102.        pthread_t ThreadA;
103.
104.        unsigned long long sum=0,index=0;
105.        unsigned long long temp1,temp2;
106.        struct timeval tpstart,tpend;
107.        float  timeuse;
108.
109.        test.a= 0;
110.        test.b= 0;
111.
112.        cpu_nums = sysconf(_SC_NPROCESSORS_CONF);
113.
114.        if(-1 == set_cpu(0))
115.        {
116.            return -1;
117.        }
118.
119.        gettimeofday(&tpstart,NULL);
120.
121.        temp1=test.a;
122.        temp2=test.b;
123.
124.        pthread_create(&ThreadA,NULL,add,&test);
125.
126.        for(sum=0;sum<MIDDLE_VALUE;sum++)
127.        {
128.            temp1 += sum;
129.            temp2 += sum;
130.        }
131.
132.        value[0][1]=temp1;
133.        value[1][1]=temp2;
134.
135.        sum=0;
136.
137.        for(index=0;index<ORANGE_MAX_VALUE;index++)
138.        {
139.            sum+=test1.a[index]+test1.b[index];
140.        }
141.
142.        pthread_join(ThreadA,NULL);
143.
144.        test.a=value[0][0]+value[0][1];
145.        test.b=value[1][0]+value[1][1];
146.
147.        gettimeofday(&tpend,NULL);
148.
```

```
149.        timeuse=MSECOND*(tpend.tv_sec-tpstart.tv_sec)+tpend.tv_usec-tpstart.tv_usec;
150.        timeuse/=MSECOND;
151.        printf("main thread:%x,Used Time:%f\n",pthread_self(),timeuse);
152.
153.        printf("a = %llu,b = %llu,sum=%llu\n",test.a,test.b,sum);
154.
155.        return 0;
156.    }
157.
158.    #ifdef __cplusplus
159.    }
160.    #endif
```

修改 opcontrol 的***--separate*** 参数为 cpu，然后开始采样：

```
# opcontrol --separate=cpu --no-vmlinux
# opcontrol --reset
# opcontrol --start
Using 2.6+ OProfile kernel interface.
Using log file /var/lib/oprofile/samples/oprofiled.log
Daemon started.
Profiler running.
# ./affinitytwoprofile
……
# opcontrol --shutdown
Stopping profiling.
```

采样结果如下：

```
# opreport -l ./affinitytwoprofile
CPU: CPU with timer interrupt, speed 0 MHz (estimated)
Profiling through timer interrupt
Samples on CPU 0
Samples on CPU 1
Samples on CPU all
samples  %        samples  %        samples  %        symbol name
147      100.000  0        0        147      50.9559  main
0        0        143      100.000  143      49.0441  add
```

程序运行时间为：0.575131s，与没有绑定之前的效果接近，但不代表 CPU 绑定没有用处，只是本样例程序运行时间较短，工作量不大，不适合使用 CPU 绑定而已。

oprofile 分析多核程序与分析多线程程序类似，通过采样数目来识别重载，然后开始一步一步的优化。

总结

根据以上分析及实验，对所有改进方案的测试时间做一个综合对比，如下图所示：

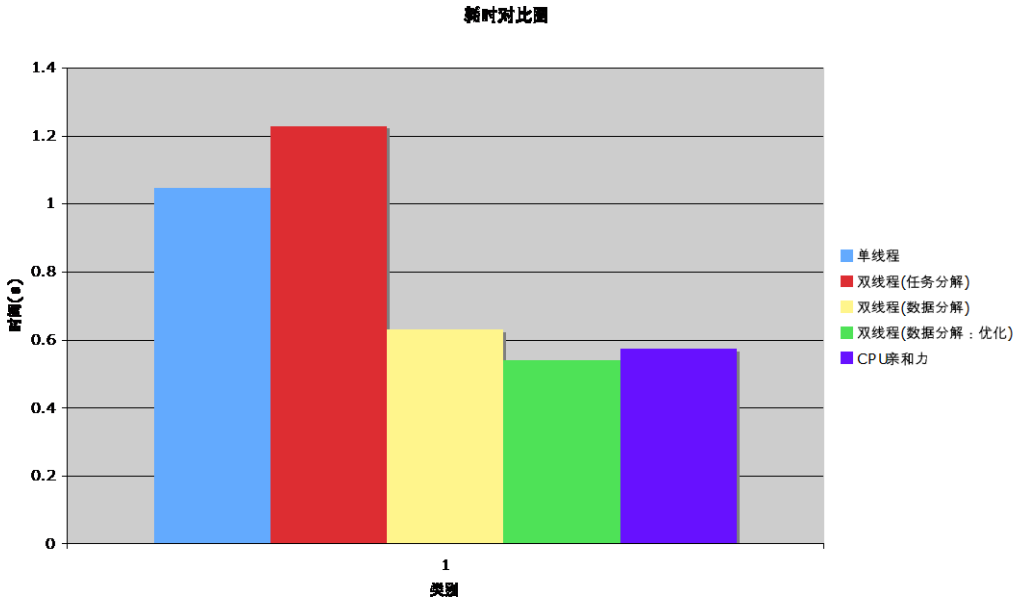

图 9 各优化时间对比图

利用 Oprofile，一步一步的不断调优，最终使优化后的结果接近于理论值，让我们见证了优化工具所具有的魅力。但 Oprofile 不仅仅只有这些功能，关于更多的其他功能，请参看官方网站介绍或者本文参考资料所列出的资料 2 和 3。

参考资料

1. Oprofile 官方网站

2. PrPrasanna S. Panchamukhi，《用 OProfile 彻底了解性能》， IBM Developerworks

3. John Engel，《使用 OProfile for Linux on POWER 识别性能瓶颈》， IBM Developerworks

4. 杨小华，《利用多核多线程进行程序优化》， IBM Developerworks

例解 autoconf 和 automake 生成 Makefile 文件[3]

本文介绍了在 linux 系统中，通过 Gnu autoconf 和 automake 生成 Makefile 的方法。主要探讨了生成 Makefile 的来龙去脉及其机理，接着详细介绍了配置 Configure.in 的方法及其规则。

引子

无论是在 Linux 还是在 Unix 环境中，make 都是一个非常重要的编译命令。不管是自己进行项目开发还是安装应用软件，我们都经常要用到 make 或 make install。利用 make 工具，我们可以将大型的开发项目分解成为多个更易于管理的模块，对于一个包括几百个源文件的应用程序，使用 make 和 makefile 工具就可以轻而易举的理顺各个源文件之间纷繁复杂的相互关系。

但是如果通过查阅 make 的帮助文档来手工编写 Makefile,对任何程序员都是一场挑战。幸而有 GNU 提供的 Autoconf 及 Automake 这两套工具使得编写 makefile 不再是一个难题。

本文将介绍如何利用 GNU Autoconf 及 Automake 这两套工具来协助我们自动产生 Makefile 文件，并且让开发出来的软件可以像大多数源码包那样，只需"./configure", "make","make install" 就可以把程序安装到系统中。

模拟需求

假设源文件按如下目录存放，如下图所示，运用 autoconf 和 automake 生成 makefile 文件。

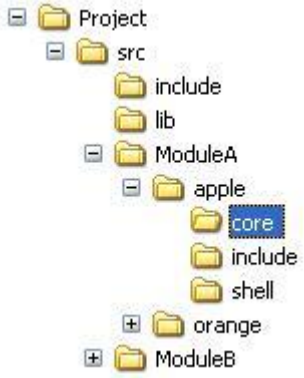

图 10 文件目录结构

假设 src 是我们源文件目录，include 目录存放其他库的头文件，lib 目录存放用到的库文件，然后开始按模块存放，每个模块都有一个对应的目录，模块下再分子模块，如 apple、orange。每个子目录下又分 core，include，shell 三个目录，其中 core 和 shell 目录存放.c 文件，include 的存放.h 文件，其他类似。

[3] 最初由 IBM developerWorks 中国网站发表，其网址是 http://www.ibm.com/developerworks/cn

样例程序功能：基于多线程的数据读写保护。

工具简介

所必须的软件：autoconf/automake/m4/perl/libtool（其中 libtool 非必须）。

autoconf 是一个用于生成可以自动地配置软件源码包，用以适应多种 UNIX 类系统的 shell 脚本工具，其中 autoconf 需要用到 m4，便于生成脚本。automake 是一个从 Makefile.am 文件自动生成 Makefile.in 的工具。为了生成 Makefile.in，automake 还需用到 perl，由于 automake 创建的发布完全遵循 GNU 标准，所以在创建中不需要 perl。libtool 是一款方便生成各种程序库的工具。

目前 automake 支持三种目录层次：flat、shallow 和 deep。

1. flat 指的是所有文件都位于同一个目录中。

即所有源文件、头文件以及其他库文件都位于当前目录中，且没有子目录。Termutils 就是这一类。

2. shallow 指的是主要的源代码都储存在顶层目录，其他各个部分则储存在子目录中。

即主要源文件在当前目录中，而其它一些实现各部分功能的源文件位于各自不同的目录。automake 本身就是这一类。

3. deep 指的是所有源代码都被储存在子目录中；顶层目录主要包含配置信息。

即所有源文件及自己写的头文件位于当前目录的一个子目录中，而当前目录里没有任何源文件。 GNU cpio 和 GNU tar 就是这一类。

flat 类型是最简单的，deep 类型是最复杂的。不难看出，我们的模拟需求正是基于第三类 deep 型，也就是说我们要做挑战性的事情：)。注：我们的测试程序是基于多线程的简单程序。

生成 Makefile 的来龙去脉

首先进入 project 目录，在该目录下运行一系列命令，创建和修改几个文件，就可以生成符合该平台的 Makefile 文件，操作过程如下：

1. 运行 autoscan 命令
2. 将 configure.scan 文件重命名为 configure.in，并修改 configure.in 文件
3. 在 project 目录下新建 Makefile.am 文件，并在 core 和 shell 目录下也新建 makefile.am 文件
4. 在 project 目录下新建 NEWS、 README、 ChangeLog 、AUTHORS 文件

5. 将/usr/share/automake-1.X/目录下的 depcomp 和 complie 文件拷贝到本目录下

6. 运行 aclocal 命令

7. 运行 autoconf 命令

8. 运行 automake -a 命令

9. 运行./confiugre 脚本

可以通过下图看出产生 Makefile 的流程，如图所示：

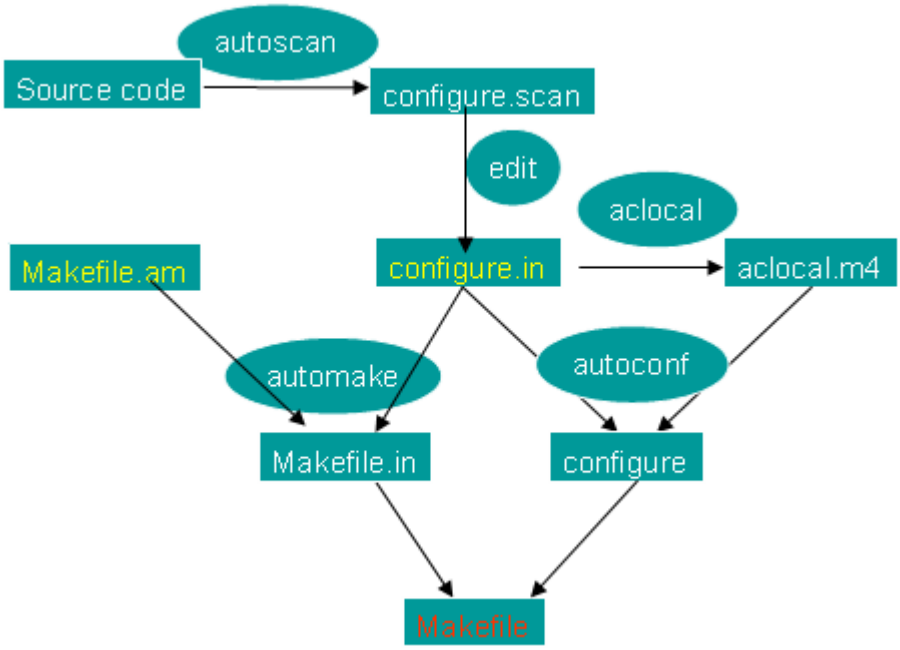

图 11 生成 Makefile 流程图

Configure.in 的八股文

当我们利用 autoscan 工具生成 confiugre.scan 文件时，我们需要将 confiugre.scan 重命名为 confiugre.in 文件。confiugre.in 调用一系列 autoconf 宏来测试程序需要的或用到的特性是否存在，以及这些特性的功能。

下面我们就来目睹一下 confiugre.scan 的庐山真面目：

```
# Process this file with autoconf to produce a configure script.
AC_PREREQ(2.59)
AC_INIT(FULL-PACKAGE-NAME, VERSION, BUG-REPORT-ADDRESS)
AC_CONFIG_SRCDIR([config.h.in])
AC_CONFIG_HEADER([config.h])
# Checks for programs.
AC_PROG_CC
# Checks for libraries.
```

```
# FIXME: Replace `main' with a function in `-lpthread':
AC_CHECK_LIB([pthread], [main])
# Checks for header files.
# Checks for typedefs, structures, and compiler characteristics
# Checks for library functions.
AC_OUTPUT
```

每个 configure.scan 文件都是以 AC_INIT 开头，以 AC_OUTPUT 结束。我们不难从文件中看出 confiugre.in 文件的一般布局：

```
AC_INIT
    测试程序
    测试函数库
    测试头文件
    测试类型定义
    测试结构
    测试编译器特性
    测试库函数
    测试系统调用
AC_OUTPUT
```

上面的调用次序只是建议性质的，但我们还是强烈建议不要随意改变对宏调用的次序。

现在就开始修改该文件：

```
$mv configure.scan configure.in
$vim configure.in
```

修改后的结果如下：

```
#                    -*- Autoconf -*-
# Process this file with autoconf to produce a configure script.
AC_PREREQ(2.59)
AC_INIT(test, 1.0, normalnotebook@126.com)
AC_CONFIG_SRCDIR([src/ModuleA/apple/core/test.c])
AM_CONFIG_HEADER(config.h)
AM_INIT_AUTOMAKE(test,1.0)
# Checks for programs.
AC_PROG_CC
# Checks for libraries.
# FIXME: Replace `main' with a function in `-lpthread':
AC_CHECK_LIB([pthread], [pthread_rwlock_init])
AC_PROG_RANLIB
# Checks for header files.
# Checks for typedefs, structures, and compiler characteristics.
```

```
# Checks for library functions.
AC_OUTPUT([Makefile
        src/lib/Makefile
         src/ModuleA/apple/core/Makefile
         src/ModuleA/apple/shell/Makefile
         ])
```

其中要将 AC_CONFIG_HEADER([config.h])修改为：AM_CONFIG_HEADER(config.h)，并加入 AM_INIT_AUTOMAKE(test,1.0)。由于我们的测试程序是基于多线程的程序，所以要加入 AC_PROG_RANLIB，不然运行 automake 命令时会出错。在 AC_OUTPUT 输入要创建的 Makefile 文件名。

由于我们在程序中使用了读写锁，所以需要对库文件进行检查，即 AC_CHECK_LIB([pthread], [main])，该宏的含义如下：

宏	含义
AC_CHECK_LIB （lib,function[,action_if_founc [,action_if_not_found, [,other_libs]]]）	该宏用来检查 lib 库中是否存在指定的函数。当测试成功时，执行 shell 命令 action_if_found 或者当 action_if_found 为空时，在输出变量 LIBS 中添加 -llib。aciton_if_not_founc 把 -lother_libs 选项传给 link 命令

其中，LIBS 是 link 的一个选项，详细请参看后续的 Makefile 文件。由于我们在程序中使用了读写锁，所以我们测试 pthread 库中是否存在 pthread_rwlock_init 函数。

由于我们是基于 deep 类型来创建 makefile 文件，所以我们需要在四处创建 Makefile 文件。即：project 目录下，lib 目录下，core 和 shell 目录下。

Autoconf 提供了很多内置宏来做相关的检测，限于篇幅关系，我们在这里对其他宏不做详细的解释，具体请参看参考文献 1 和参考文献 2，也可参看 autoconf 信息页。

实战 Makefile.am

Makefile.am 是一种比 Makefile 更高层次的规则。只需指定要生成什么目标，它由什么源文件生成，要安装到什么目录等构成。

下表列出了可执行文件、静态库、头文件和数据文件，四种书写 Makefile.am 文件个一般格式。

文件类型	书写格式
可执行文件	bin_PROGRAMS = foo foo_SOURCES =xxxx.c foo_LDADD = foo_LDFLAGS = foo_DEPENDENCIES =
静态库	lib_LIBRARIES = libfoo.a foo_a_SOURCES = foo_a_LDADD = foo_a_LIBADD = foo_a_LDFLAGS =
头文件	include_HEADERS = foo.h
数据文件	data_DATA = data1 data2

表 6Makefile.am 一般格式

对于可执行文件和静态库类型，如果只想编译，不想安装到系统中，可以用 noinst_PROGRAMS 代替 bin_PROGRAMS，noinst_LIBRARIES 代替 lib_LIBRARIES。

Makefile.am 还提供了一些全局变量供所有的目标体使用：

变量	含义
INCLUDES	比如链接时所需要的头文件
LDADD	比如链接时所需要的库文件
LDFLAGS	比如链接时所需要的库文件选项标志
EXTRA_DIST	源程序和一些默认的文件将自动打入.tar.gz 包，其它文件若要进入.tar.gz 包可以用这种办法，比如配置文件，数据文件等等。
SUBDIRS	在处理本目录之前要递归处理哪些子目录

表 7Makefile.am 中可用的全局变量

在 Makefile.am 中尽量使用相对路径，系统预定义了两个基本路径：

路径变量	含义
$(top_srcdir)	工程最顶层目录，用于引用源程序
$(top_builddir)	定义了生成目标文件上最上层目录，用于引用.o 等编译出来的目标文件。

表 8 Makefile.am 中可用的路径变量

在上文中我们提到过安装路径，automake 设置了默认的安装路径：

1. 标准安装路径

默认安装路径为：$(prefix) = /usr/local，可以通过./configure --prefix=<new_path>的方法来覆盖。

其它的预定义目录还包括：bindir = $(prefix)/bin, libdir = $(prefix)/lib, datadir = $(prefix)/share, sysconfdir = $(prefix)/etc 等等。

2. 定义一个新的安装路径

比如 test, 可定义 testdir = $(prefix)/test, 然后 test_DATA =test1 test2，则 test1，test2 会作为数据文件安装到$(prefix)/ /test 目录下。

我们首先需要在工程顶层目录下（即 project/）创建一个 Makefile.am 来指明包含的子目录：

```
SUBDIRS=src/lib src/ModuleA/apple/shell src/ModuleA/apple/core
CURRENTPATH=$(shell /bin/pwd)
INCLUDES=-I$(CURRENTPATH)/src/include -I$(CURRENTPATH)/src/ModuleA/apple/include
export INCLUDES
```

由于每个源文件都会用到相同的头文件，所以我们在最顶层的 Makefile.am 中包含了编译源文件时所用到的头文件，并导出，见蓝色部分代码。

我们将 lib 目录下的 swap.c 文件编译成 libswap.a 文件，被 apple/shell/apple.c 文件调用，那么 lib 目录下的 Makefile.am 如下所示：

```
noinst_LIBRARIES=libswap.a
libswap_a_SOURCES=swap.c
INCLUDES=-I$(top_srcdir)/src/includ
```

细心的读者可能就会问：怎么表 9 中给出的是 bin_LIBRARIES，而这里是 noinst_LIBRARIES? 这是因为如果只想编译，而不想安装到系统中，就用 noinst_LIBRARIES 代替 bin_LIBRARIES，对于可执行文件就用 noinst_PROGRAMS 代替 bin_PROGRAMS。对于安装的情况，库将会安装到$(prefix)/lib 目录下，可执行文件将会安装到${prefix}/bin。如果想安装该库，则 Makefile.am 示例如下：

```
bin_LIBRARIES=libswap.a
libswap_a_SOURCES=swap.c
INCLUDES=-I$(top_srcdir)/src/include
swapincludedir=$(includedir)/swap
swapinclude_HEADERS=$(top_srcdir)/src/include/swap.h
```

最后两行的意思是将 swap.h 安装到${prefix}/include /swap 目录下。

接下来，对于可执行文件类型的情况，我们将讨论如何写 Makefile.am？对于编译 apple/core 目录下的文件，我们写成的 Makefile.am 如下所示：

```
noinst_PROGRAMS=test
test_SOURCES=test.c
test_LDADD=$(top_srcdir)/src/ModuleA/apple/shell/apple.o $(top_srcdir)/src/lib/libswap.a
test_LDFLAGS=-D_GNU_SOURCE
DEFS+=-D_GNU_SOURCE
#LIBS=-lpthread
```

由于我们的 test.c 文件在链接时，需要 apple.o 和 libswap.a 文件，所以我们需要在 test_LDADD 中包含这两个文件。对于 Linux 下的信号量/读写锁文件进行编译，需要在编译选项中指明-D_GNU_SOURCE。所以在 test_LDFLAGS 中指明。而 test_LDFLAGS 只是链接时的选项，编译时同样需要指明该选项，所以需要 DEFS 来指明编译选项，由于 DEFS 已经有初始值，所以这里用+=的形式指明。从这里可以看出，Makefile.am 中的语法与 Makefile 的语法一致，也可以采用条件表达式。如果你的程序还包含其他的库，除了用 AC_CHECK_LIB 宏来指明外，还可以用 LIBS 来指明。

如果你只想编译某一个文件，那么 Makefile.am 如何写呢？这个文件也很简单，写法跟可执行文件的差不多，如下例所示：

```
noinst_PROGRAMS=apple
apple_SOURCES=apple.c
DEFS+=-D_GNU_SOURCE
```

我们这里只是欺骗 automake，假装要生成 apple 文件，让它为我们生成依赖关系和执行命令。所以当你运行完 automake 命令后，然后修改 apple/shell/下的 Makefile.in 文件，直接将 LINK 语句删除，即：

```
.......
clean-noinstPROGRAMS:
 -test -z "$(noinst_PROGRAMS)" || rm -f $(noinst_PROGRAMS)
apple$(EXEEXT): $(apple_OBJECTS) $(apple_DEPENDENCIES)
 @rm -f apple$(EXEEXT)
#$(LINK) $(apple_LDFLAGS) $(apple_OBJECTS) $(apple_LDADD) $(LIBS)
.......
```

通过上述处理，就可以达到我们的目的。从图 12 中不难看出为什么要修改 Makefile.in 的原因，而不是修改其他的文件。

调试篇