**《操作系统》实验报告**

徽标, 公司名称

描述已自动生成

**题目: 实验五 多核多线程编程**

**学 院 计算机学院**

**班 级 2021211304**

**学 号 2021212171**

**姓 名 杨晨、张梓良**

**2023年 11月**

## 目 录

第一章 实验概述 1

1.1 实验目的 1

1.2 实验内容 1

1.2.1 多核多线程编程及性能分析 1

1.2.2 运行时间对比总结 1

1.3 实验设计原理 1

第二章 实验步骤 2

2.1 观察实验平台物理CPU 、CPU 核和逻辑CPU 的数目 2

2.1.1 基础知识 2

2.1.2 实验过程 2

2.2 单线程/进程串行vs 双线程并行vs 三线程加锁并行程序 2

2.2.1 程序功能 2

2.2.2 K-best测量方法 2

2.2.3 单线程/进程串行 6

2.2.4 双线程并行 7

2.2.5 三线程加锁并行 8

2.3 三线程加锁 vs 三线程不加锁 10

2.3.1 三线程不加锁并行 10

2.4 针对Cache的优化 12

2.4.1 Cache优化三线程不加锁 12

2.4.2 Cache优化三线程加锁 16

2.5 CPU亲和力对并行程序影响 18

2.5.1 双线程设置硬亲和力 18

2.5.2 三线程设置硬亲和力 20

第三章 实验结果分析 22

3.1 观察实验平台物理CPU 、CPU 核和逻辑CPU 的数目 22

3.1.1 执行结果 22

3.1.2 结果分析 22

3.2 对比分析单线程/进程、2线程、3线程加锁程序的运行时间差异 22

3.2.1 执行结果 22

3.2.2 结果分析 24

3.3 对比分析三线程加锁、三线程不加锁的运行时间差异 24

3.3.1 执行结果 24

3.3.2 结果分析 26

3.4 对比分析Cache 优化后运行时间的差异 26

3.4.1 不加锁优化后的执行结果 26

3.4.2 不加锁优化后的结果分析 29

3.4.3 加锁优化后的执行结果 29

3.4.4 加锁优化后的结果分析 31

3.5 对比分析CPU亲和力对并行程序影响 31

3.5.1 单线程/进程程序、双线程（无CPU亲和）和双线程（有CPU亲和）的结果 31

3.5.2 结果分析 33

3.5.3 三线程Cache优化（无CPU亲和）和三线程Cache优化（有CPU亲和）的结果 33

3.5.4 结果分析 35

第四章 总结 36

4.1八种实现方案运行时间对比总结 36

# **第一章** **实验概述**

##### 1.1 实验目的

掌握多CPU、多核硬件环境下基本的多线程并行编程技术。

##### 1.2 实验内容

# **1.2.1 多核多线程编程及性能分析**

参照参考文献“利用多核多线程进行程序优化”，在Linux 环境下，编写多线程程序，依次完成下述实验：

* 实验5.1观察实验平台物理CPU 、CPU 核和逻辑CPU的数目
* 实验5.2单线程/进程串行 vs 两线程并行 vs 三线程加锁并行程序对比
* 实验5.3三线程加锁 vs 三线程不加锁 对比
* 实验5.4针对 Cache 的优化
* 实验5.5 CPU 亲和力对并行程序影响

# **1.2.2 运行时间对比总结**

* 以柱状图形式表示上述5 种情况下程序运行时间的定量测试结果（运行时间）
* 分析对比程序并行化、线程数目、共享资源加、CPU 亲和、cache 优化对程序运行时间的影响，结合程序/进程自身业务逻辑、相互间同步互斥关系等分析解释运行时间产生差异的原因

##### 1.3 实验设计原理

参照参考文献“利用多核多线程进行程序优化”，在Linux 环境下，编写多线程程序，分析以下几个因素对程序运行时间的影响：

* 程序并行化
* 线程数目
* 共享资源加锁
* CPU 亲和
* cache 优化

# **第二章 实验步骤**

##### 2.1 观察实验平台物理CPU 、CPU 核和逻辑CPU 的数目

# **2.1.1** **基础知识**

物理CPU 数目：主板上实际插入的CPU 数量，可以数不重复的 physical id 有几个（physical id）。

多路服务器、大型主机系统、集群系统一般可以配置多个物理CPU；常规微机、笔记本电脑一般只配备1 个物理CPU；

CPU 核（cpu cores）的数目：单块CPU 上面能处理数据的芯片组的数量，如双核、四核等；

逻辑CPU 数目：

对不支持超线程HT 的CPU，逻辑CPU数目=物理CPU个数×每颗CPU核数，

对支持超线程HT 的CPU，逻辑CPU数目=物理CPU个数×每颗CPU核数\*2

# **2.1.2 实验过程**

查看物理CPU数目：

[XXXX@server ~]$ grep 'physical id' /proc/cpuinfo|sort|uniq|wc -l

运行结果



查看CPU核数：

[XXXX@server ~]$ grep 'cpu cores' /proc/cpuinfo|uniq|awk -F ':' '{print $2}'

运行结果



查看逻辑CPU数：

[XXXX@server ~]$ cat /proc/cpuinfo| grep "processor"|wc -l

运行结果



##### 2.2 单线程/进程串行vs 双线程并行vs 三线程加锁并行程序

# **2.2.1 程序功能**

求从1一直到 APPLE\_MAX\_VALUE(100000000)相加累计的和，并赋值给apple的a和b；求orange数据结构中的a[i]+b[i]的和，循环ORANGE\_MAX\_VALUE(1000000)次。

# **2.2.2 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。

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。

代码如下：

#define MICRO\_SECOND 1000000      // 1s = 10^6us  
// 用于qsort的比较函数  
int cmp(const void \*a, const void \*b)  
{  
    return \*(long \*)a - \*(long \*)b;  
}  
   
/\*  
 \* k\_best: 用于获取最佳时间的函数  
 \* k: 最佳时间的个数  
 \* epsilon: 误差百分比  
 \* m: 循环次数  
 \* fuc: 调用的函数  
 \* return: 返回一个数组，数组中存储了最佳时间  
 \*/  
long \*k\_best(int k, int epsilon, int m, void (\*fuc)(void))  
{  
    // 分配存储最佳时间的数组  
    long \*result = (long \*)malloc(sizeof(long) \* k);  
    // 将数组初始化为0  
    memset(result, 0, sizeof(long) \* k);  
   
    struct timeval start, end;  
    int i;  
    long time\_cost;  
   
    // 执行循环m次  
    for (i = 0; i < m; i++)  
    {  
        // 获取起始时间  
        if (gettimeofday(&start, NULL) == -1)  
        {  
            perror("gettimeofday error\n");  
        }  
   
        // 调用传入的函数  
        fuc();  
   
        // 获取结束时间  
        if (gettimeofday(&end, NULL) == -1)  
        {  
            perror("gettimeofday error\n");  
        }  
   
        // 计算时间消耗  
        time\_cost = (end.tv\_sec - start.tv\_sec) \* MICRO\_SECOND + (end.tv\_usec - start.tv\_usec);  
        printf("%d: %ld us\n", i, time\_cost);  
   
        // 如果是前k次循环  
        if (i < k)  
        {  
            // 将时间消耗存入结果数组  
            result[i] = time\_cost;  
   
            // 如果已经存满k个时间  
            if (i == k - 1)  
            {  
                // 对结果数组进行排序  
                qsort(result, i + 1, sizeof(long), cmp);  
   
                // 如果最后一个时间小于等于第一个时间乘以(1 + epsilon / 100)  
                // 则满足要求，结束循环  
                if (result[k - 1] <= result[0] \* (1.0 + epsilon / 100.0))  
                {  
                    printf("break when i = %d\n", i);  
                    break;  
                }  
            }  
        }  
        // 如果是第k次循环之后  
        else  
        {  
            // 如果当前时间消耗小于结果数组中的最大值  
            if (time\_cost < result[k - 1])  
            {  
                // 将当前时间消耗存入结果数组，并重新排序  
                result[k - 1] = time\_cost;  
                qsort(result, k, sizeof(long), cmp);  
            }  
   
            // 如果最后一个时间小于等于第一个时间乘以(1 + epsilon / 100)  
            // 则满足要求，结束循环  
            if (result[k - 1] <= result[0] \* (1.0 + epsilon / 100.0))  
            {  
                printf("break when i = %d\n", i);  
                break;  
            }  
        }  
    }  
   
    printf("Done\n");  
    return result;  
}  
   
int main(int argc, char \*argv[])  
{  
    // gcc -o code code.c -lpthread  
    long \*arr;  
    int k = 10, epsilon = 2, m = 200;  
    int i;  
    long ave, sum = 0;  
    int fun\_num;  
   
    cpu\_nums = sysconf(\_SC\_NPROCESSORS\_CONF);  
   
    if (argc < 2)  
    {  
        printf("Usage: ./code [function number]\n");  
        exit(1);  
    }  
   
    fun\_num = atoi(argv[1]);  
    switch (fun\_num)  
    {  
    case 1:  
        arr = k\_best(k, epsilon, m, single\_thread);  
        break;  
   
    default:  
        printf("No such function\n");  
        break;  
    }  
   
    for (i = 0; i < k; i++)  
    {  
        printf("%ld\n", arr[i]);  
        sum += arr[i];  
    }  
    ave = sum / k;  
    printf("ave: %ld\n", ave);  
    return 0;  
}

# **2.2.3 单线程/进程串行**

依照附件，单线程/进程的代码如下

#define ORANGE\_MAX\_VALUE 1000000  // 1M  
#define APPLE\_MAX\_VALUE 100000000 // 100M  
struct apple  
{  
    unsigned long long a;  
    unsigned long long b;  
};  
   
struct orange  
{  
    int a[ORANGE\_MAX\_VALUE];  
    int b[ORANGE\_MAX\_VALUE];  
};  
   
void single\_thread()  
{  
    int sum, index;  
    struct apple test;  
    struct orange test1;  
   
    for (sum = 0; sum < APPLE\_MAX\_VALUE; sum++)  
    {  
        test.a += sum;  
        test.b += sum;  
    }  
   
    sum = 0;  
    for (index = 0; index < ORANGE\_MAX\_VALUE; index++)  
    {  
        sum += test1.a[index] + test1.b[index];  
    }  
}

测试结果如下：

文本

描述已自动生成

该结果的含义为，在运行到i = 13的次数时，按照误差e = 2%的要求，结果已经收敛。此时输出最短时间的十次结果，与这十次结果的平均时间(微秒)。

# **2.2.4 双线程并行**

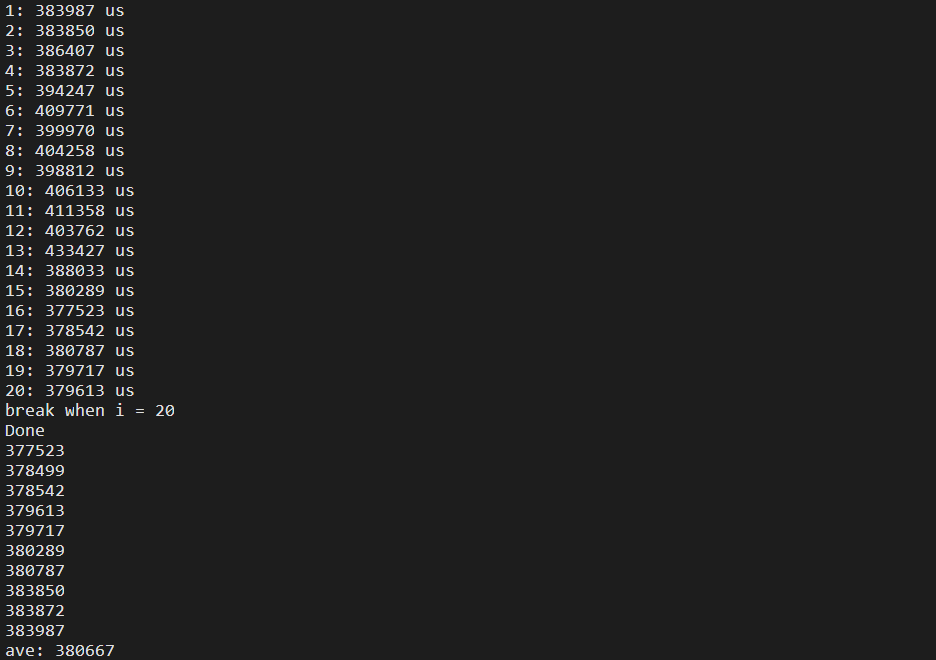
仔细分析样例程序，运用任务分解的方法 ，不难发现计算 apple 的值和计算 orange 的值，属于完全不相关的两个操作，因此可以并行。

采用任务分解的方法，将互不相关的计算 apple值和计算 orange值的2部分代码分解为2个线程，实现线程级并行执行。

代码如下：

void \*add(void \*x)  
{  
    int sum;  
    for (sum = 0; sum < APPLE\_MAX\_VALUE; sum++)  
    {  
        ((struct apple \*)x)->a += sum;  
        ((struct apple \*)x)->b += sum;  
    }  
    return NULL;  
}  
// 双线程执行的函数  
void double\_thread()  
{  
    int sum, index;  
    struct apple test;  
    struct orange test1 = {{0}, {0}};  
    pthread\_t ThreadA;  
   
    pthread\_create(&ThreadA, NULL, add, &test);  
   
    for (index = 0; index < ORANGE\_MAX\_VALUE; index++)  
    {  
        sum += test1.a[index] + test1.b[index];  
    }  
    pthread\_join(ThreadA, NULL);  
}

运行结果如下：



该结果的含义为，在运行到i =20的次数时，按照误差e = 2%的要求，结果已经收敛。此时输出最短时间的十次结果，与这十次结果的平均时间(微秒)。

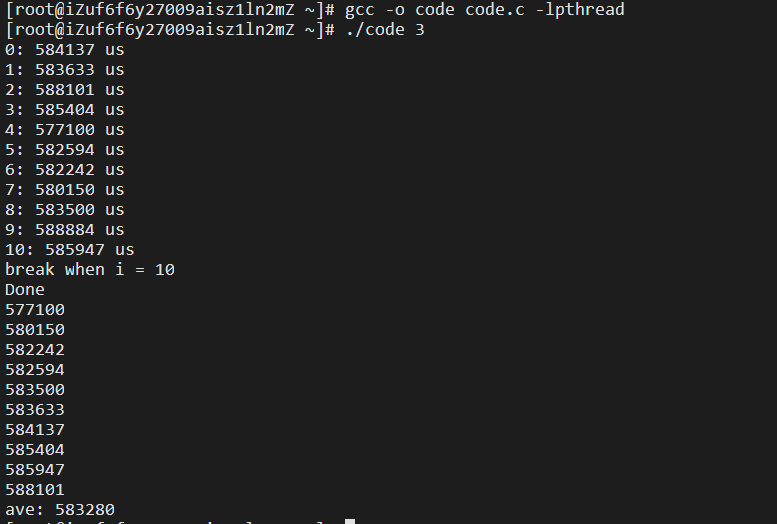
# **2.2.5 三线程加锁并行**

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。

代码如下：

struct apple\_lock  
{  
    unsigned long long a;  
    unsigned long long b;  
    pthread\_rwlock\_t rwLock;  
};  
void \*addx(void \*x)  
{  
    // 加锁  
    pthread\_rwlock\_wrlock(&((struct apple\_lock \*)x)->rwLock);  
    int sum;  
    for (sum = 0; sum < APPLE\_MAX\_VALUE; sum++)  
    {  
        ((struct apple \*)x)->a += sum;  
    }  
    // 解锁  
    pthread\_rwlock\_unlock(&((struct apple\_lock \*)x)->rwLock);  
}  
// 用于三线程执行的函数  
void \*addy(void \*y)  
{  
    // 加锁  
    pthread\_rwlock\_wrlock(&((struct apple\_lock \*)y)->rwLock);  
    int sum;  
    for (sum = 0; sum < APPLE\_MAX\_VALUE; sum++)  
    {  
        ((struct apple \*)y)->b += sum;  
    }  
    // 解锁  
    pthread\_rwlock\_unlock(&((struct apple\_lock \*)y)->rwLock);  
}  
// 三线程执行的函数  
void three\_thread()  
{  
    struct apple\_lock test;  
    struct orange test1 = {{0}, {0}};  
    pthread\_t ThreadA, ThreadB;  
    pthread\_rwlock\_init(&test.rwLock, NULL);  
    pthread\_create(&ThreadA, NULL, addx, &test);  
    pthread\_create(&ThreadB, NULL, addy, &test);  
   
    int sum, index;  
    for (index = 0; index < ORANGE\_MAX\_VALUE; index++)  
    {  
        sum += test1.a[index] + test1.b[index];  
    }  
   
    pthread\_join(ThreadA, NULL);  
    pthread\_join(ThreadB, NULL);  
    pthread\_rwlock\_destroy(&test.rwLock);  
}

运行结果如下：



该结果的含义为，在运行到i = 10的次数时，按照误差e = 2%的要求，结果已经收敛。此时输出最短时间的十次结果，与这十次结果的平均时间(微秒)。

##### 2.3 三线程加锁 vs 三线程不加锁

# **2.3.1 三线程不加锁并行**

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（删除读写锁代码），通过不加锁来提高性能。

代码如下：

void \*addx\_noLock(void \*x)  
{  
    int sum;  
    for (sum = 0; sum < APPLE\_MAX\_VALUE; sum++)  
    {  
        ((struct apple \*)x)->a += sum;  
    }  
}  
// 用于三线程不加锁执行的函数  
void \*addy\_noLock(void \*y)  
{  
    int sum;  
    for (sum = 0; sum < APPLE\_MAX\_VALUE; sum++)  
    {  
        ((struct apple \*)y)->b += sum;  
    }  
}  
// 三线程不加锁执行的函数  
void three\_thread\_noLock()  
{  
    struct apple test;  
    struct orange test1 = {{0}, {0}};  
    pthread\_t ThreadA, ThreadB;  
    pthread\_create(&ThreadA, NULL, addx\_noLock, &test);  
    pthread\_create(&ThreadB, NULL, addy\_noLock, &test);  
   
    int sum, index;  
    for (index = 0; index < ORANGE\_MAX\_VALUE; index++)  
    {  
        sum += test1.a[index] + test1.b[index];  
    }  
   
    pthread\_join(ThreadA, NULL);  
    pthread\_join(ThreadB, NULL);  
}

运行结果如下：

形状

低可信度描述已自动生成

可以看出，在3线程不加锁的方案非常不稳定，测量时间有时相差两倍。使用k-best测量方法，实验达到200次后，最快十次的误差也没有小于2%，该方案不收敛。最后统计使用最快十次的测量结果，但实际上由于该方案的不稳定，每一次的运行结果都有差异。

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。

##### 2.4 针对Cache的优化

# **2.4.1 Cache优化三线程不加锁**

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。

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。从前面的实验结果也可以看出，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。

这里，尝试32字节、64字节和128字节的字符数组大小，让apple的两个成员a和b位于不同的Cache行中，观察增加Cache后的运行时间，分析是否达到预期要求。这里只展示设置32字节的代码，64字节、128字节同理

**32字节的代码如下**：

// 32字节cache优化的apple  
struct apple\_cache\_32  
{  
    unsigned long long a;  
    char c[32];  
    unsigned long long b;  
};  
void \*addx\_cache\_32(void \*x)  
{  
    int sum;  
    for (sum = 0; sum < APPLE\_MAX\_VALUE; sum++)  
    {  
        ((struct apple\_cache\_32 \*)x)->a += sum;  
    }  
}  
// 用于三线程32字节cache优化执行的函数  
void \*addy\_cache\_32(void \*y)  
{  
    int sum;  
    for (sum = 0; sum < APPLE\_MAX\_VALUE; sum++)  
    {  
        ((struct apple\_cache\_32 \*)y)->b += sum;  
    }  
}  
// 三线程32字节cache优化执行的函数  
void three\_thread\_cache\_32()  
{  
    struct apple\_cache\_32 test;  
    struct orange test1 = {{0}, {0}};  
    pthread\_t ThreadA, ThreadB;  
    pthread\_create(&ThreadA, NULL, addx\_cache\_32, &test);  
    pthread\_create(&ThreadB, NULL, addy\_cache\_32, &test);  
   
    int sum, index;  
    for (index = 0; index < ORANGE\_MAX\_VALUE; index++)  
    {  
        sum += test1.a[index] + test1.b[index];  
    }  
   
    pthread\_join(ThreadA, NULL);  
    pthread\_join(ThreadB, NULL);  
}

运行结果1：

文本

低可信度描述已自动生成

运行结果2：

文本

描述已自动生成

多次运行改程序，发现运行结果相差甚远。有时可以快速收敛；有时则时分不稳定，不能收敛。

同理，64字节的运行结果如下：

文本

描述已自动生成

最后，验证设置128字节的情况，运行结果如下：

文本

描述已自动生成

# **2.4.2 Cache优化三线程加锁**

针对加锁三线程程序，对apple数据结构也增加一行类似功能的代码。

预期结论：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。这里以设置64字节的Cache优化为例

代码如下：

// 64字节cache优化的带有锁的apple  
struct apple\_lock\_cache\_64  
{  
    unsigned long long a;  
    char c[64];  
    unsigned long long b;  
    pthread\_rwlock\_t rwLock;  
};  
void \*addx\_lock\_cache\_64(void \*x)  
{  
    int sum;  
    // 加锁  
    pthread\_rwlock\_wrlock(&((struct apple\_lock\_cache\_64 \*)x)->rwLock);  
    for (sum = 0; sum < APPLE\_MAX\_VALUE; sum++)  
    {  
        ((struct apple\_lock\_cache\_64 \*)x)->a += sum;  
    }  
    // 解锁  
    pthread\_rwlock\_unlock(&((struct apple\_lock\_cache\_64 \*)x)->rwLock);  
}  
// 用于三线程64字节cache优化加锁执行的函数  
void \*addy\_lock\_cache\_64(void \*y)  
{  
    int sum;  
    // 加锁  
    pthread\_rwlock\_wrlock(&((struct apple\_lock\_cache\_64 \*)y)->rwLock);  
    for (sum = 0; sum < APPLE\_MAX\_VALUE; sum++)  
    {  
        ((struct apple\_lock\_cache\_64 \*)y)->b += sum;  
    }  
    // 解锁  
    pthread\_rwlock\_unlock(&((struct apple\_lock\_cache\_64 \*)y)->rwLock);  
}  
// 三线程64字节cache优化加锁执行的函数  
void three\_threed\_lock\_cache\_64()  
{  
    // 3线程加锁cache优化3  
    struct apple\_lock\_cache\_64 test;  
    struct orange test1 = {{0}, {0}};  
    pthread\_t ThreadA, ThreadB;  
    pthread\_rwlock\_init(&test.rwLock, NULL);  
    pthread\_create(&ThreadA, NULL, addx\_lock\_cache\_64, &test);  
    pthread\_create(&ThreadB, NULL, addy\_lock\_cache\_64, &test);  
   
    int sum, index;  
    for (index = 0; index < ORANGE\_MAX\_VALUE; index++)  
    {  
        sum += test1.a[index] + test1.b[index];  
    }  
   
    pthread\_join(ThreadA, NULL);  
    pthread\_join(ThreadB, NULL);  
    pthread\_rwlock\_destroy(&test.rwLock);  
}

运行结果如下：

文本

描述已自动生成

##### 2.5 CPU亲和力对并行程序影响

# **2.5.1 双线程设置硬亲和力**

在多核机器上，针对两线程的方案，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，分析对比与单线程/进程程序、2线程（无CPU亲和）程序的运行时间。

代码如下：

inline int set\_cpu(int i)  
{  
    cpu\_set\_t mask;  
    CPU\_ZERO(&mask);   // 将cpu\_set\_t结构清零  
    if (cpu\_nums >= 2) // 如果有两个及以上cpu  
    {  
        CPU\_SET(i, &mask); // 设置cpu\_set\_t结构的i位为1，即将cpu i加入其中  
   
        if (sched\_setaffinity(gettid(), sizeof(mask), &mask) == -1)  
        {  
            perror("sched\_setaffinity error\n");  
            return -1;  
        }  
    }  
    return 0;  
}  
// 用于双线程绑定执行的函数  
void \*addx\_bind(void \*x)  
{  
    if (set\_cpu(1) == -1)  
    {  
        return NULL;  
    }  
    int sum;  
    for (sum = 0; sum < APPLE\_MAX\_VALUE; sum++)  
    {  
        ((struct apple \*)x)->a += sum;  
        ((struct apple \*)x)->b += sum;  
    }  
}  
// 双线程绑定执行的函数  
void double\_thread\_bind()  
{  
    struct orange test1;  
    struct apple test;  
    if (set\_cpu(0) == -1)  
    {  
        return;  
    }  
    pthread\_t ThreadA;  
    pthread\_create(&ThreadA, NULL, addx\_bind, &test);  
    int sum, index;  
    for (index = 0; index < ORANGE\_MAX\_VALUE; index++)  
    {  
        sum += test1.a[index] + test1.b[index];  
    }  
    pthread\_join(ThreadA, NULL);  
}

运行结果如下：

文本

描述已自动生成

# **2.5.2 三线程设置硬亲和力**

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。为此，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。观察程序运行时间，并与采用Cache的三线程方案进行对比。

此处使用64字节的Cache优化，代码如下：

void \*addx\_bind\_3(void \*x)  
{  
    // 设置cpu亲和性,且考虑Cache对性能的影响  
    if (set\_cpu(1) == -1)  
    {  
        return NULL;  
    }  
   
    int sum;  
    for (sum = 0; sum < APPLE\_MAX\_VALUE; sum++)  
    {  
        ((struct apple\_cache\_64 \*)x)->a += sum;  
    }  
}  
// 用于三线程绑定执行的函数  
void \*addy\_bind\_3(void \*y)  
{  
    // 设置cpu亲和性,且考虑Cache对性能的影响  
    if (set\_cpu(2) == -1)  
    {  
        return NULL;  
    }  
   
    int sum;  
    for (sum = 0; sum < APPLE\_MAX\_VALUE; sum++)  
    {  
        ((struct apple\_cache\_64 \*)y)->b += sum;  
    }  
}  
// 三线程绑定执行的函数  
void three\_thread\_bind()  
{  
    struct apple\_cache\_64 test;  
    struct orange test1 = {{0}, {0}};  
    if (set\_cpu(0) == -1)  
    {  
        return;  
    }  
   
    pthread\_t ThreadA, ThreadB;  
    pthread\_create(&ThreadA, NULL, addx\_bind\_3, &test);  
    pthread\_create(&ThreadB, NULL, addy\_bind\_3, &test);  
   
    int sum, index;  
    for (index = 0; index < ORANGE\_MAX\_VALUE; index++)  
    {  
        sum += test1.a[index] + test1.b[index];  
    }  
   
    pthread\_join(ThreadA, NULL);  
    pthread\_join(ThreadB, NULL);  
}

运行结果如下：

文本

描述已自动生成

# **第三章 实验结果分析**

##### 3.1 观察实验平台物理CPU 、CPU 核和逻辑CPU 的数目

# **3.1.1 执行结果**

查看物理CPU数目：

[XXXX@server ~]$ grep 'physical id' /proc/cpuinfo|sort|uniq|wc -l

运行结果



查看CPU核数：

[XXXX@server ~]$ grep 'cpu cores' /proc/cpuinfo|uniq|awk -F ':' '{print $2}'

运行结果



查看逻辑CPU数：

[XXXX@server ~]$ cat /proc/cpuinfo| grep "processor"|wc -l

运行结果



# **3.1.2 结果分析**

根据提供的命令和结果，可以对系统的CPU配置进行分析：

物理CPU数目：运行结果1

这个结果表示系统中存在1个物理CPU。物理CPU是指实际的硬件处理器单元，一个物理CPU可能包含多个核心。

CPU核数：运行结果2

这个结果表示每个CPU拥有2个核心。CPU核心是指可以独立执行指令的处理单元，一个物理CPU可以包含多个核心。

逻辑CPU数目：运行结果4

这个结果表示系统中存在4个逻辑CPU。逻辑CPU是通过超线程技术实现的虚拟CPU，可以提升处理器的并发执行能力。

综上所述，该系统有1个物理CPU，每个CPU有2个核心，以及4个逻辑CPU。这意味着系统具备一定的多线程并行处理能力，可以同时执行多个任务或线程。

##### 3.2 对比分析单线程/进程、2线程、3线程加锁程序的运行时间差异

# **3.2.1 执行结果**

单线程

文本

描述已自动生成

双线程

图片包含 文本

描述已自动生成

三线程加锁

文本

描述已自动生成

# **3.2.2 结果分析**

图表, 条形图

描述已自动生成

首先观察单线程、双线程和三线程的结果。结果显示，双线程的耗时大于单线程，这是由于线程启停以及线程上下文切换都会引起额外的开销，所以消耗的时间比单线程多。另外主要耗时的线程是apple线程，orrange线程的开销小，所以由于多线程带来的收益没有开销大。从中可以吸取到的经验是，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。

而三线程的耗时巨大的原因是，addx线程和addy线程，由于要竞争同一把锁，所以实际上是串行运行的。比双线程多出的时间则是三线程产生的系统开销。由于访问apple的两个线程访问的是 apple 的不同元素，根本没有加锁的必要，所以可以删除读写锁代码。

##### 3.3 对比分析三线程加锁、三线程不加锁的运行时间差异

# **3.3.1 执行结果**

三线程加锁

文本

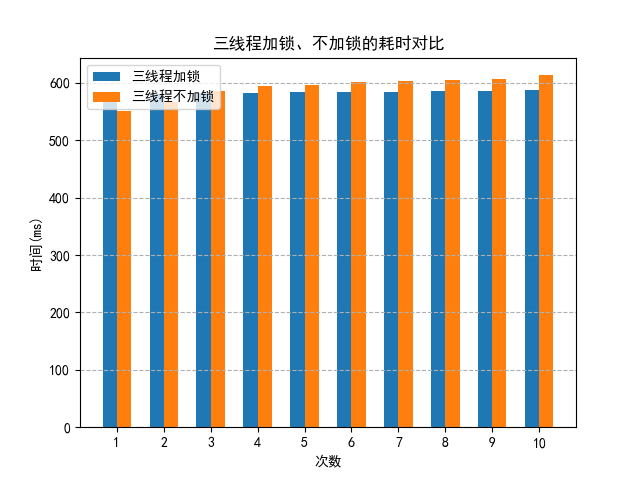
描述已自动生成

三线程不加锁

形状

低可信度描述已自动生成

# **3.3.2 结果分析**



当多个线程同时读写位于同一个缓存行的不同变量时，会发生伪共享问题，这导致CPU缓存的失效。具体而言，在本问题中，尽管addx\_noLock线程操作的变量a与addy\_noLock线程操作的变量b之间没有直接关联，但它们在主内存中相邻，并共享同一个缓存行。由于缓存行是CPU缓存的最小单位，当任意一个线程修改其中一个变量时，整个缓存行都会失效。这意味着频繁的多线程操作将导致CPU缓存完全失效，从而导致CPU内核和主内存之间的直接交互，降低了性能。

为了解决伪共享问题，常见的方法是通过字节填充（Padding）来确保不同变量位于不同的缓存行。通过在变量之间插入额外的字节，我们可以确保它们位于不同的缓存行，从而避免多个线程同时修改同一缓存行的情况。

##### 3.4 对比分析Cache 优化后运行时间的差异

# **3.4.1 不加锁优化后的执行结果**

不加锁优化前的结果：

形状

低可信度描述已自动生成

32字节结果1和结果2：

文本

低可信度描述已自动生成

文本

描述已自动生成

64字节结果

文本

描述已自动生成

128字节的结果

文本

描述已自动生成

# **3.4.2 不加锁优化后的结果分析**

图表, 条形图

描述已自动生成

前文中提到，填充32字节的方案每次运行结果大不相同。这是因为在实验平台上的处理器，Cache 行的大小为64bytes，可以用如下命令查看cache行大小：

cat /sys/devices/system/cpu/cpu0/cache/index0/coherency\_line\_size

得到的结果如下



因此，如果使用32字节做填充，a与b有时会被分配到同一个cache行，从而产生cache伪共享的问题，有时则不会。

因为使用32字节填充的不稳定性，之后再考虑字节填充时不会采用这种方案。

# **3.4.3 加锁优化后的执行结果**

加锁不优化的结果：

文本

描述已自动生成

64字节优化后结果：

文本

描述已自动生成

# **3.4.4 加锁优化后的结果分析**

图表, 条形图

描述已自动生成

结果是没有什么变化。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。

三线程不加锁并行程序带给我们的启示是，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。

##### 3.5 对比分析CPU亲和力对并行程序影响

# **3.5.1 单线程/进程程序、双线程（无CPU亲和）和双线程（有CPU亲和）的结果**

单线程/进程

文本

描述已自动生成

双线程（无亲和）

图片包含 文本

描述已自动生成

双线程（有亲和）

文本

描述已自动生成

# **3.5.2 结果分析**

图表, 条形图

描述已自动生成

其测量结果与预期相符合。设置CPU硬亲和力比不设置的效率略有提升。这是因为在同一个CPU上执行，则cache中缓存的数据在下一次运行时仍然存在，而进程在CPU之间迁移会提升消耗。Linux 内核进程调度器天生就具有被称为CPU软亲和力的特性，这意味着进程通常不会在处理器之间频繁迁移，但是不频繁迁移不代表不迁移。而设置硬亲和力之后，避免这种迁移造成的消耗，在某种程度上硬亲和力比软亲和力具有一定的优势，同时还提高了程序的可靠性。

但花费的时间还是比单线程的多，其原因之前分析的类似。

# **3.5.3 三线程Cache优化（无CPU亲和）和三线程Cache优化（有CPU亲和）的结果**

三线程64字节Cache优化，无锁的结果

文本

描述已自动生成

三线程64字节Cache优化，无锁，绑定的结果

文本

描述已自动生成

# **3.5.4 结果分析**

图表, 条形图

描述已自动生成

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



文本

描述已自动生成



文本

描述已自动生成

其中，./code 52是方案五中，用64BCache优化（不设置CPU亲和），./code 8是方案八，用64B优化（设置CPU亲和）

在实际问题中，我们需要综合考虑复杂均衡、数据竞争、系统开销的问题，以免事倍功半。

# **第四章 总结**

##### 4.1 八种实现方案运行时间对比总结

根据前面实验内容，总结对比单线程/进程串行程序和其它7 种多线程并行程序的运行时间，并以图表方式给出具体结果

图表, 条形图

描述已自动生成

最终的实验结果与实验指导书中的不同之处在于，单线程仍然是最优的方案。推测这是由于多线程造成的系统开销的缘故。其他的结果均与实验指导书一致。