北京邮电大学

操作系统实验三

多线程编程

陈朴炎　2021211138

2023-10-26

目录

实验三、多线程编程. 2
1 实验内容 2
1.1 实验 1 内容 2
1.2 实验 2 内容 2
2 实验环境 3
3 实验 1--多线程计算序列有效值 3
3.1 实验 1 内容. 3
3.2 程序设计. 4
3.3 测试报告. 14
4 实验 2--实现多线程矩阵乘法 24
4.1 实验 2 内容. 24
4.2 程序设计. 24
4.3 测试报告. 42

实验三、多线程编程

1 实验内容

1.1 实验 1 内容

编写一个多线程程序来计算数字列表的各种统计值。该程序将从键盘接受一系列数字，然后创建三个单独的工作线程。第一个线程将确定数字的平均值，第二个线程将确定最大值，第三个线程将确定最小值。例如，假设您的程序接受整数 90 81 78 95 79 72 85，程序将报告：平均值为 82 最小值为 72 最大值为 95 代表平均值、最小值和最大值的变量将被全局存储。工作线程将设置这些值，一旦工作线程退出，父线程将输出这些值。

1.2 实验 2 内容

编写多线程程序，实现矩阵相乘。给定两个矩阵 A 和 B，其中 A 是 M 行 K 列的矩阵，矩阵 B 包含 K 行 N 列，A 和 B 的矩阵乘积是矩阵 C，其中 C 包含 M 行和 N 列。矩阵 C 中第 i 行第 j 列 C(i,j) 的项是矩阵 A 中第 i 行元素与矩阵 B 中第 j 列元素的乘积之和。即：

$$C_{i,j} = \sum_{n=1}^{k} A_{n,i} * B_{n,j}$$

例如，如果 A 是 3×2 矩阵，B 是 2×3 矩阵，则元素 C(3,1)将是 A(3,1) x B(1,1)和 A(3,2) x B(2,1)之和。对于此项目，在单独的工作线程中计算每个元素 C(i,j)。这将涉及创建 M x N 工作线程。主线程或父线程将初始化矩阵 A 和 B，并为矩阵 C 分配足够的内存，矩阵 C 将保存矩阵 A 和 B 的乘积。这些矩阵将被声明为全局数据，以便每个工作线程都可以访问 A、B 和 C。

2 实验环境

Win11 下 WSL windows for linux 子系统

Ubuntu 22.04.2 LTS

gcc (Ubuntu 11.3.0-1ubuntu1~22.04.1) 11.3.0

VS Code 版本: 1.83.1 (user setup)

Electron: 25.8.4

ElectronBuildId: 24154031

Chromium: 114.0.5735.289

Node.js: 18.15.0

V8: 11.4.183.29-electron.0

OS: Windows_NT x64 10.0.22000

3 实验 1--多线程计算序列有效值

3.1 实验 1 内容

编写一个多线程程序来计算数字列表的各种统计值。该程序将从键盘接受一系列数字，然后创建三个单独的工作线程。第一个线程将确定数字的平均值，第二个线程将确定最大值，第三个线程将确定最小值。例如，假设您的程序接受整数 90 81 78 95 79 72 85，程序将报告：平均值为 82 最小值为 72 最大值为 95 代表平均值、最小值和最大值的变量将被全局存储。工作线程将设置这些值，一旦工作线程退出，父线程将输出这些值。

3.2 程序设计

3.2.1 主要的 API

1、pthread_create()

API 定义：int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void *(*start_routine)(void*), void *arg);

pthread_create 函数是 POSIX 线程库（Pthreads）提供的函数，用于创建新的线程。

参数说明：

thread：一个指向 pthread_t 类型的指针，用于存储新线程的标识符。

attr：线程属性，通常可以设置为 NULL，表示使用默认线程属性。

start_routine：一个指向函数的指针，这个函数是新线程的入口点。新线程将从这个函数开始执行。

arg：传递给 start_routine 函数的参数。

当使用 pthread_create 函数时，它的作用是创建一个新的线程。这个新线程会执行指定的函数（入口点）并开始执行。pthread_create 允许我们并行执行多个线程，以便同时完成多个任务。它的主要参数包括线程标识符、线程属性、入口点函数和传递给入口点函数的参数。一旦创建新线程，它将在后台执行，直到完成其任务。

2、pthread_join()

API 定义：int pthread_join(pthread_t thread, void **retval)；

pthread_join()函数用于等待一个线程的结束，它会让一个线程等待另一个线程

的结束，主要用于同步线程之间的执行顺序。

参数说明：

thread：要等待的线程的标识符。

retval：一个指向指针的指针，用于获取线程的返回值，通常可以设置为NULL。

pthread_join 函数用于等待一个特定线程的结束。主要用于同步线程的执行顺序。它会阻塞调用线程，直到指定的线程结束。这可以确保在主线程中等待其他线程完成后再继续执行。通过 pthread_join，我们可以获取目标线程的返回值，以便在主线程中获取线程的执行结果。

3、pthread_exit()

API 定义：void pthread_exit(void *retval);

参数说明：

这个函数的参数 retval 是一个指向线程返回值的指针。线程的返回值可以用于在父线程中获取线程的执行结果。

当使用 pthread_exit 函数时，它的作用是终止当前线程的执行。这意味着线程会立即退出，不会再执行其他指令。你可以在需要时调用 pthread_exit 来结束线程的执行，通常是在线程任务完成后进行调用。这个函数的目的是使线程正常退出，而不是终止线程的执行。

当调用 pthread_exit 时，你可以传递一个指针作为参数，这个指针可以用于传递线程的返回值。这个返回值可以在其他线程中使用 pthread_join 函数来获取。这允许线程在退出时提供一个结果或状态信息。

总之，pthread_exit 的主要作用是结束当前线程的执行，而不是终止整个程序。它可以带一个返回值，以便在其他线程中获取线程的执行结果。

3.2.2 程序设计说明

定义全局变量：

maxNum 是存储序列最大值的变量，minNum 是存储序列最小值的变量，averageNum 是存储序列中平均值的变量，number 用来接收用户输入的序列长度值，numberSeq 用来存储用户输入的长度为 number 的序列。

```c
int maxNum, minNum;
double averageNum;
int number;
int * numberSeq;
```

定义工作函数：

void averageCal() 函数是用来计算序列中平均值的函数，它读取了全局变量中的 numberSeq，遍历其中的所有值，来算出序列的平均值。

```c
void averageCal(){
    double sum = 0.0;
    for(int i = 0; i < number; i++){
        sum += numberSeq[i];
    }
```

```
    averageNum = sum / number;
    printf("\tthread1 calculate the average: %.2lf\n", averageNum);
    pthread_exit(NULL);
}
```

void minimum() 函数首先把 minNum 值设置为非常大，然后遍历序列进行比较，找出序列中的最小值，存放到 minNum 中。

```
c
void minimum(){
    minNum = 65535;
    for(int i = 0; i < number; i++){
        if(minNum > numberSeq[i]){
            minNum = numberSeq[i];
        }
    }
    printf("\tthread2 find the minimum: %d\n", minNum);
    pthread_exit(NULL);
}
```

void maximum() 函数首先把 maxNum 设置为非常小，然后遍历序列进行比较，找出序列中的最大值，存放到 maxNum 中。

```
c
void maximum(){
```

```
    maxNum = -65535;
    for(int i = 0;i < number; i++){
        if(maxNum < numberSeq[i]){
            maxNum = numberSeq[i];
        }
    }
    printf("\tthread3 find the maxmum: %d\n", maxNum);
    pthread_exit(NULL);
}
```

提示用户输入、开辟空间：

提示用户输入序列信息，然后开辟动态的序列空间，再提示用户输入完整序列，存储到指定的空间中。

c

```
    printf("Please enter the number: ");
    scanf("%d",&number);
    numberSeq = (int*)malloc(sizeof(int)*number);
    memset(numberSeq, 0, sizeof(numberSeq));
    for(int i =0; i < number; i++){
        printf("please input the %d number: ",i+1);
        scanf("%d",&numberSeq[i]);
    }
```

创建线程：

之后需要把线程定义好、创建好，分别传入工作函数参数，使它们并行执行。同时为了防止线程创建失败，需要判断线程是否创建成功，如果创建失败，将输出线程创建失败信息。

c

```
    pthread_t thread1, thread2, thread3;
    int ret;
    ret = pthread_create(&thread1, NULL, (void *)averageCal, NULL);
    if(ret != 0){
        printf("Create thread1 error.\n");
        return -1;
    }
    ret = pthread_create(&thread2, NULL, (void *)minimum, NULL);
    if(ret != 0){
        printf("Create thread1 error.\n");
        return -1;
    }
    ret = pthread_create(&thread3, NULL, (void *)maximum, NULL);
    if(ret != 0){
        printf("Create thread1 error.\n");
        return -1;
```

```
    }
```

释放资源：

```
c
    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);
    pthread_join(thread3, NULL);
    printf("The parent get the result\n\taverage: %.2lf\n\tminimum: %d\n\tmaxmum:%d\n",averageNum,minNum,maxNum);

    free(numberSeq);
```

3.2.3 源程序代码

```
c
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
int maxNum, minNum;
double averageNum;
```

```
int number;
int * numberSeq;


void averageCal(){
	double sum = 0.0;
	for(int i = 0; i < number; i++){
		sum += numberSeq[i];
	}
	averageNum = sum / number;
	printf("\tthread1 calculate the average: %.2lf\n", averageNum);
	pthread_exit(NULL);
}


void minimum(){
	minNum = 65535;
	for(int i = 0; i < number; i++){
		if(minNum > numberSeq[i]){
			minNum = numberSeq[i];
		}
	}
	printf("\tthread2 find the minimum: %d\n", minNum);
```

```
	pthread_exit(NULL);
}

void maximum(){
	maxNum = -65535;
	for(int i = 0;i < number; i++){
		if(maxNum < numberSeq[i]){
			maxNum = numberSeq[i];
		}
	}
	printf("\tthread3 find the maxmum: %d\n", maxNum);
	pthread_exit(NULL);
}

int main(){
	pthread_t thread1, thread2, thread3;
	int ret;
	printf("Please enter the number: ");
	scanf("%d",&number);
	numberSeq = (int*)malloc(sizeof(int)*number);
	memset(numberSeq, 0, sizeof(numberSeq));
```

```
for(int i =0; i < number; i++){
        printf("please input the %d number: ",i+1);
        scanf("%d",&numberSeq[i]);
}

// 创建线程
ret = pthread_create(&thread1, NULL, (void *)averageCal, NULL);
if(ret != 0){
        printf("Create thread1 error.\n");
        return -1;
}
ret = pthread_create(&thread2, NULL, (void *)minimum, NULL);
if(ret != 0){
        printf("Create thread1 error.\n");
        return -1;
}
ret = pthread_create(&thread3, NULL, (void *)maximum, NULL);
if(ret != 0){
        printf("Create thread1 error.\n");
        return -1;
}
```

```
    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);
    pthread_join(thread3, NULL);
    printf("The parent get the result\n\taverage: %.2lf\n\tminimum: %d\n\tmaxmum:%d\n",averageNum,minNum,maxNum);

    free(numberSeq);
    return 0;
}
```

3.3 测试报告

3.3.1 测试用例 1

输入数据

input
Please enter the number: 7 please input the 1 number: 90 please input the 2 number: 81 please input the 3 number: 78 please input the 4 number: 95 please input the 5 number: 79

```
please input the 6 number: 72

please input the 7 number: 85
```

输入序列长度 7，再输入序列：90、81、78、95、79、72、85

运行结果截图

```
allpfirestorm@LAPTOP-O0JFQIE5:~/projects/bupt-homework/os/threads$ ./threads
Please enter the number: 7
please input the 1 number: 90
please input the 2 number: 81
please input the 3 number: 78
please input the 4 number: 95
please input the 5 number: 79
please input the 6 number: 72
please input the 7 number: 85
        thread1 calculate the average: 82.86
        thread2 find the minimum: 72
        thread3 find the maxmum: 95
The parent get the result
        average: 82.86
        minimum: 72
        maxmum:95
```

图 3-1 测试结果 1

测试结果分析

线程 1 计算出序列的平均值为 82.86，线程 2 计算出最小值为 72，线程 3 计算出最大值为 95。并且在父进程中，等待完各个子线程计算结束后准确得到了这些进程计算出的结果。

程序创建了三个线程，分别用于计算平均值、查找最小值和查找最大值。这三个线程在操作系统的管理下并行执行，每个线程执行特定的计算任务。这展示了操作系统如何管理多线程应用程序的执行。

程序使用 pthread_join 函数来等待每个线程完成其任务。主线程在等待期间会被阻塞，直到所有三个线程完成。这确保了线程的同步执行，主线程只在所有计算任务完成后才继续执行。

每个线程计算了特定的结果（平均值、最小值、最大值），并将这些结果打印

出来。这展示了线程在并行执行时如何协同工作，各自独立计算任务的结果。

程序在堆上分配了内存来存储输入的数字序列，并在计算完成后释放了这些内存。这是操作系统的内存管理的一个示例，确保程序正确管理和释放资源，以避免内存泄漏。

3.3.2 测试用例 2

输入数据

input
Please enter the number: 9 please input the 1 number: 1 please input the 2 number: 2 please input the 3 number: 3 please input the 4 number: 4 please input the 5 number: 5 please input the 6 number: 6 please input the 7 number: 7 please input the 8 number: 8 please input the 9 number: 9

首先输入序列长度 9，再依次输入序列的各个值：1、2、3、4、5、6、7、8、9

运行结果截图

图 3-2 测试结果 2

运行结果分析

子线程正确计算出了平均值、最小值和最大值，父进程等待所有子线程退出后获得了正确的平均值、最小值、最大值。

程序创建了三个线程，分别用于计算平均值、查找最小值和查找最大值。这三个线程在操作系统的管理下并行执行，每个线程执行特定的计算任务。这展示了操作系统如何管理多线程应用程序的执行。

程序使用 pthread_join 函数来等待每个线程完成其任务。主线程在等待期间会被阻塞，直到所有三个线程完成。这确保了线程的同步执行，主线程只在所有计算任务完成后才继续执行。

每个线程计算了特定的结果（平均值、最小值、最大值），并将这些结果打印出来。这展示了线程在并行执行时如何协同工作，各自独立计算任务的结果。

程序在堆上分配了内存来存储输入的数字序列，并在计算完成后释放了这些内存。这是操作系统的内存管理的一个示例，确保程序正确管理和释放资源，以避免内存泄漏。

3.3.3 测试用例 3

输入数据

Input
Please enter the number: 6 please input the 1 number: 52 please input the 2 number: -78 please input the 3 number: 65 please input the 4 number: 91 please input the 5 number: -256 please input the 6 number: 654

首先输入序列长度 6，再输入序列：52、-78、65、91、-256、654

运行结果截图

```
allpfirestorm@LAPTOP-O0JFQIE5:~/projects/bupt-homework/os/threads$ ./threads
Please enter the number: 6
please input the 1 number: 52
please input the 2 number: -78
please input the 3 number: 65
please input the 4 number: 91
please input the 5 number: -256
please input the 6 number: 654
        thread1 calculate the average: 88.00
        thread2 find the minimum: -256
        thread3 find the maxmum: 654
The parent get the result
        average: 88.00
        minimum: -256
        maxmum:654
```

图 3-3 测试结果 3

运行结果分析

程序创建了三个线程，分别用于计算平均值、查找最小值和查找最大值。这三个线程在操作系统的管理下并行执行，每个线程执行特定的计算任务。这展示

了操作系统如何管理多线程应用程序的执行。

程序使用 pthread_join 函数来等待每个线程完成其任务。主线程在等待期间会被阻塞，直到所有三个线程完成。这确保了线程的同步执行，主线程只在所有计算任务完成后才继续执行。

每个线程计算了特定的结果（平均值、最小值、最大值），并将这些结果打印出来。这展示了线程在并行执行时如何协同工作，各自独立计算任务的结果。

程序在堆上分配了内存来存储输入的数字序列，并在计算完成后释放了这些内存。这是操作系统的内存管理的一个示例，确保程序正确管理和释放资源，以避免内存泄漏。

3.3.4 测试用例 4

Input
Please enter the number: 75 please input the 1 number: 8888 please input the 2 number: 7777 please input the 3 number:8888 please input the 4 number: 8888 please input the 5 number: 9999 please input the 6 number: 6666 please input the 7 number: 4444 please input the 8 number: 55555 please input the 9 number: 33333

```
please input the 10 number: 22222
please input the 11 number: 11111
please input the 12 number: 22222
please input the 13 number: 0
please input the 14 number: -1
please input the 15 number: 5
please input the 16 number: 2
please input the 17 number: 2
please input the 18 number: 2
please input the 19 number: 2
please input the 20 number: 7
please input the 21 number: 4
please input the 22 number: 5
please input the 23 number: 6
please input the 24 number: 5
please input the 25 number: 4
please input the 26 number:2
please input the 27 number: 1
please input the 28 number:5
please input the 29 number: 4
please input the 30 number: 4
```

please input the 31 number: 2

please input the 32 number: 2

please input the 33 number: 3

please input the 34 number: 2

please input the 35 number: 1

please input the 36 number: 5

please input the 37 number:7

please input the 38 number: 8

please input the 39 number: 9

please input the 40 number: 5

please input the 41 number: 4

please input the 42 number: 6

please input the 43 number: 2

please input the 44 number: 1

please input the 45 number: 2

please input the 46 number: 3

please input the 47 number:5

please input the 48 number: 4

please input the 49 number: 5

please input the 50 number: 6

please input the 51 number: 5

```
please input the 52 number: 5

please input the 53 number: 5

please input the 54 number:1

please input the 55 number: 2

please input the 56 number: 3

please input the 57 number: 2

please input the 58 number: 5

please input the 59 number: 4

please input the 60 number:8

please input the 61 number: 7

please input the 62 number: 8

please input the 63 number: 5

please input the 64 number: 6

please input the 65 number: 5

please input the 66 number: 1

please input the 67 number: 2

please input the 68 number: 3

please input the 69 number: 2

please input the 70 number: 1

please input the 71 number: 2

please input the 72 number: 3
```

```
please input the 73 number: 2

please input the 74 number: 1

please input the 75 number: 5
```

输入了长度为 75 的序列，序列如上。

运行结果截图

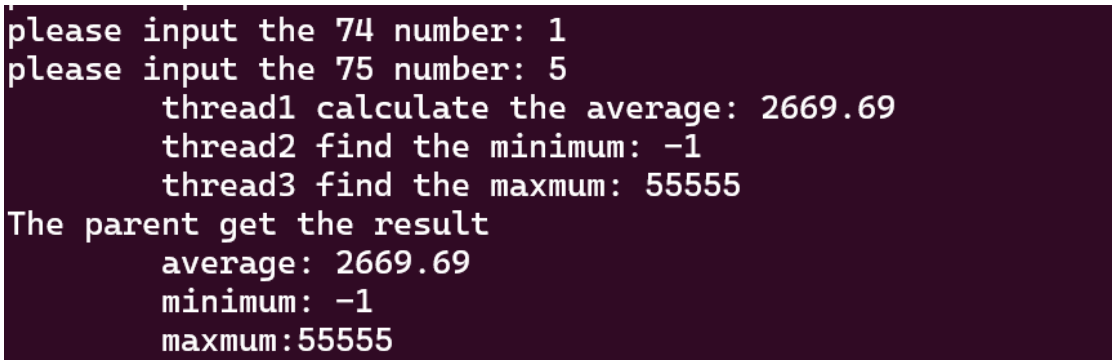

图 3-4 测试结果 4

测试结果分析

用户输入了一个包含 75 个数字的序列，这些数字代表不同的数据点。序列包含了正数、负数、零和多位数。这个数字序列反映了真实世界中可能的多样性。

程序创建了三个线程，分别计算平均值、查找最小值和查找最大值。这三个线程并行执行，每个线程独立处理不同的计算任务。这展示了操作系统如何同时管理多个线程的执行，以提高计算效率。

程序使用 pthread_join 函数来等待每个线程完成其任务。主线程会在等待期间被阻塞，直到所有三个线程都完成了它们的任务。这确保了线程的同步执行，主线程只在所有计算任务完成后才继续执行。

每个线程计算了特定的结果（平均值、最小值、最大值），并将这些结果打印出来。这些结果基于输入数字序列的计算。最大值是 55555，最小值是-1，平均

值是 2669.69。

4 实验 2--实现多线程矩阵乘法

4.1 实验 2 内容

编写多线程程序，实现矩阵相乘。给定两个矩阵 A 和 B，其中 A 是 M 行 K 列的矩阵，矩阵 B 包含 K 行 N 列，A 和 B 的矩阵乘积是矩阵 C，其中 C 包含 M 行和 N 列。矩阵 C 中第 i 行第 j 列 C(i,j) 的项是矩阵 A 中第 i 行元素与矩阵 B 中第 j 列元素的乘积之和。即：

$$C_{i,j} = \sum_{n=1}^{k} A_{n,i} * B_{n,j}$$

例如，如果 A 是 3×2 矩阵，B 是 2×3 矩阵，则元素 C(3,1)将是 A(3,1) x B(1,1)和 A(3,2) x B(2,1)之和。对于此项目，在单独的工作线程中计算每个元素 C(i,j)。这将涉及创建 M x N 工作线程。主线程或父线程将初始化矩阵 A 和 B，并为矩阵 C 分配足够的内存，矩阵 C 将保存矩阵 A 和 B 的乘积。这些矩阵将被声明为全局数据，以便每个工作线程都可以访问 A、B 和 C。

4.2 程序设计

4.2.1 主要的 API

1、pthread_create()

API 定义：int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void *(*start_routine)(void*), void *arg);

pthread_create 函数是 POSIX 线程库（Pthreads）提供的函数，用于创建新的线程。

参数说明：

thread：一个指向 pthread_t 类型的指针，用于存储新线程的标识符。

attr：线程属性，通常可以设置为 NULL，表示使用默认线程属性。

start_routine：一个指向函数的指针，这个函数是新线程的入口点。新线程将从这个函数开始执行。

arg：传递给 start_routine 函数的参数。

当使用 pthread_create 函数时，它的作用是创建一个新的线程。这个新线程会执行指定的函数（入口点）并开始执行。pthread_create 允许我们并行执行多个线程，以便同时完成多个任务。它的主要参数包括线程标识符、线程属性、入口点函数和传递给入口点函数的参数。一旦创建新线程，它将在后台执行，直到完成其任务。

2、void * work(void * arg)

功能：

这个函数的主要功能是计算矩阵相乘的一部分，具体来说，它计算结果矩阵 C 的一个特定元素 C[row][column] 的值。

计算方法是将矩阵 A 中的第 "row" 行和矩阵 B 中的第 "column" 列的对应元素相乘，然后将它们相加以得到结果矩阵 C 的元素。

参数：

arg：这是一个指向结构体 v 的指针，它包含了线程需要的数据，其中 data->row 表示要计算的结果矩阵 C 的行索引，data->column 表示列索引。

N：这是一个全局变量，表示矩阵的维度。

计算过程：

函数使用一个循环遍历矩阵 A 的第 "row" 行和矩阵 B 的第 "column" 列的所有元素。对于每个元素，它将 A 和 B 矩阵中的对应元素相乘，然后将结果累积到变量 sum 中。函数还打印了一些计算过程的信息，如相乘的元素和相加的结果，这有助于调试和理解线程的执行过程。

结果存储：

计算完成后，函数将计算得到的结果 sum 存储在结果矩阵 C 的相应位置，即 C[row][column]。

线程终止：

最后，函数释放了参数 arg 所指向的内存，以避免内存泄漏，并通过 pthread_exit(NULL) 终止了线程的执行。

4.2.2 程序设计说明

定义全局变量

c

int M,K,N; // C 的行数、中间数、C 的列数
int **A, **B, **C; //矩阵，A、B 为乘数，C 为结果
pthread_t ** threads; // worker threads

M、K、N 分别表示 A[M][K]，B[K][M]，M 为 A 的行数、B 的列数，K 为 A 的列数、B 的行数，M 为 C 的行数，N 为 C 的列数。

A，B 是二维数组，用来存储矩阵信息。

threads 是工作线程数组，是一个二维的线程数组，用来存放工作线程。

定义传入线程参数的结构体

```c
struct v
{
    int row;
    int column;
};
```

v 是一个结构体，存储了当前的行列信息，用来传递给各个线程，代表它们要计算的是哪一行和哪一列相乘相加的值。

定义工作函数

```c
void * work(void * arg){
    struct v * data = (struct v *)arg;
    int sum = 0;
    for(int i = 0; i < N; i++){
        printf("(%d,%d): ",data->row,data->column);
        printf("sum = %d, A[%d][%d]=%d, B[%d][%d]=%d\n", sum, data->row,i,A[data->row][i],i,data->column,B[i][data->column]);
        sum += A[data->row][i] * B[i][data->column];
    }

    C[data->row][data->column] = sum;
```

```
    printf("%d,%d: %d\n",data->row,data->column,sum);
    free(data);
    pthread_exit(NULL);
}
```

void * work(void * arg)：这是一个线程函数，接受一个指向 void 类型参数的指针 arg 作为输入。线程将使用这个参数来确定它需要计算矩阵乘法的哪个部分。

struct v * data = (struct v *)arg：这一行将 arg 参数转换为 struct v 类型的指针，以便访问行和列信息。

int sum = 0：初始化一个变量 sum 用于存储矩阵乘法的结果。

for(int i = 0; i < N; i++)：这是一个循环，用于遍历一个矩阵的行或另一个矩阵的列，从而执行矩阵乘法的每一步。

printf("(%d,%d): ",data->row,data->column)：这行代码用于打印当前线程正在计算的元素的行和列。

sum += A[data->row][i] * B[i][data->column]：这是矩阵乘法的一部分，计算两个矩阵中特定元素的乘积，并将结果累加到 sum 变量中。

C[data->row][data->column] = sum：将计算得到的结果存储在结果矩阵 C 的适当位置。

printf("%d,%d: %d\n",data->row,data->column,sum)：这行代码用于打印当前线程计算的元素的行和列以及计算结果。

free(data)：释放线程函数中分配的内存，以避免内存泄漏。

pthread_exit(NULL)：通知线程结束执行，这是线程函数的最后一行。

这个函数是用于并行计算矩阵乘法的一部分，它计算矩阵 A 的特定行与矩阵 B 的特定列的乘积，并将结果存储在结果矩阵 C 中。每个线程负责计算一个元素，通过循环迭代来完成整个矩阵的计算。

定义检查输入格式的函数

c

```
int checkMatrixFormat(FILE *file) {
    int M, N, num;
    if (fscanf(file, "%d %d %d", &M, &K, &N) != 3) {
        fprintf(stderr, "Error: Unable to read matrix dimensions from the input file.\n");
        return 0;
    }

    int expectedNumbers = K*(M + N);
    int actualNumbers = 0;

    while (fscanf(file, "%d", &num) == 1) {
        actualNumbers++;
    }
```

```
    if (actualNumbers != expectedNumbers) {
        fprintf(stderr, "Error: The number of elements in the matrix does not match the specified dimensions (M=%d, N=%d).\n", M, N);
        printf("%d\n",actualNumbers);
        return 0;
    }
    printf("File format right.\n");
    return 1;
}
```

int M, N, num：这些变量用于存储矩阵的行数（M）、列数（N）以及当前正在读取的数字（num）。

if (fscanf(file, "%d %d %d", &M, &K, &N) != 3)：从文件中读取三个整数，即矩阵的行数（M）、列数（K），和另一个矩阵的列数（N）。如果成功读取的整数个数不等于 3，表示文件中的格式不正确，会输出错误信息并返回 0。

int expectedNumbers = K*(M + N)：根据矩阵的维度信息，计算了文件中应该包含的元素个数。

int actualNumbers = 0：初始化一个变量来存储实际读取的元素个数。

while (fscanf(file, "%d", &num) == 1)：这是一个循环，它尝试从文件中逐个读取整数，并每次成功读取后将 actualNumbers 加 1。

if (actualNumbers != expectedNumbers)：在循环结束后，将实际读取的元素个数与预期的元素个数进行比较。如果它们不相等，表示文件中的元素个数与

指定的矩阵维度不匹配，会输出错误信息并返回 0。

设计文件读入格式

```c
int main(int argc, char *argv[]){
    // 从文件读入 M、N 以及矩阵 A、B 的值
    FILE * file = fopen(argv[1], "r");
    if(file == NULL){
        perror("Error opening input file.");
        return -1;
    }
    printf("open file success.\n");
    if(!checkMatrixFormat(file)){
        return -1;
    }
```

通过命令行读入文件名，并读取文件的内容，检查矩阵的格式是否正确。

分配内存并导入到数组中

```c
// 分配内存，并读入矩阵 A 和矩阵 B
    A = (int **)malloc(M * sizeof(int *));
    B = (int **)malloc(K * sizeof(int *));
    C = (int **)malloc(M * sizeof(int *));
```

```
threads = (pthread_t **)malloc(M * sizeof(pthread_t *));


printf("malloc success.\n");
for(int i = 0; i < M; i++){
    A[i] = (int *)malloc(K * sizeof(int));
    C[i] = (int *)malloc(N * sizeof(int));
    threads[i] = (pthread_t *)malloc(N * sizeof(pthread_t));


    for(int j = 0; j < K; j++){
        fscanf(file, "%d", &A[i][j]);
        printf("%d\t",A[i][j]);
    }
    printf("\n");
}
printf("A load success.\n");


for(int i =0; i < K; i++){
    B[i] = (int *)malloc(N * sizeof(int));
    for(int j = 0; j < N; j++){
        fscanf(file, "%d", &B[i][j]);
        printf("%d\t",B[i][j]);
```

```
        }
        printf("\n");
    }
    printf("B load success.\n");
    fclose(file);
```

给 A、B、C、threads 动态分配内存，并将读取到的格式正确的文件导入到数组 A、B 中。

线程并发，创建多个同步线程，计算矩阵的行列式

```c
    for(int i = 0; i < M; i++){
        for(int j = 0; j < N; j++){
            struct v * data = (struct v *) malloc (sizeof(struct v));
            data->row = i;
            data->column = j;
            // 通过 data 创建工作线程
            pthread_create(&threads[i][j], NULL, work, data);
        }
    }
```

等待各个线程执行完毕，输出 C 结果，并释放空间

```c
    // pthread_join 函数等待所有线程完成
```

```
    for (int i = 0; i < M; i++){
        for(int j = 0; j < N; j++){
            pthread_join(threads[i][j], NULL);
        }
    }

    // 打印结果矩阵 C
    printf("==============All threads has done, Result C :==================\n");
    for(int i = 0; i < M; i++){
        for(int j = 0; j < N; j ++){
            printf("%d\t",C[i][j]);
        }
        printf("\n");
    }

    // 擦屁股
    for(int i = 0; i < M; i++){
        free(A[i]);
        free(C[i]);
        free(threads[i]);
```

```
    }
    for(int j = 0; j < K; j++){
        free(B[j]);
    }
    free(A);
    free(B);
    free(C);
    free(threads);
    return 0;
}
```

4.2.3 源程序代码

C 程序源码

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

int M,K,N;     // C 的行数、中间数、C 的列数
int **A, **B, **C;   //矩阵，A、B 为乘数，C 为结果
pthread_t ** threads;     // worker threads

struct v
```

```
{
    int row;
    int column;
};

void * work(void * arg){
    struct v * data = (struct v *)arg;
    int sum = 0;
    for(int i = 0; i < N; i++){
        printf("(%d,%d): ",data->row,data->column);
        printf("sum = %d, A[%d][%d]=%d, B[%d][%d]=%d\n", sum,
data->row,i,A[data->row][i],i,data->column,B[i][data->column]);
        sum += A[data->row][i] * B[i][data->column];
    }

    C[data->row][data->column] = sum;
    printf("%d,%d: %d\n",data->row,data->column,sum);
    free(data);
    pthread_exit(NULL);
}
// 检查矩阵格式是否正确
```

```
int checkMatrixFormat(FILE *file) {
    int M, N, num;
    if (fscanf(file, "%d %d %d", &M, &K, &N) != 3) {
        fprintf(stderr, "Error: Unable to read matrix dimensions from the input file.\n");
        return 0;
    }

    int expectedNumbers = K*(M + N);
    int actualNumbers = 0;

    while (fscanf(file, "%d", &num) == 1) {
        actualNumbers++;
    }

    if (actualNumbers != expectedNumbers) {
        fprintf(stderr, "Error: The number of elements in the matrix does not match the specified dimensions (M=%d, N=%d).\n", M, N);
        printf("%d\n",actualNumbers);
        return 0;
    }
```

```
    printf("File format right.\n");
    return 1;
}

int main(int argc, char *argv[]){
    // 从文件读入 M、N 以及矩阵 A、B 的值
    FILE * file = fopen(argv[1], "r");
    if(file == NULL){
        perror("Error opening input file.");
        return -1;
    }
    printf("open file success.\n");
    if(!checkMatrixFormat(file)){
        return -1;
    }
    // 检查完重定位
    fseek(file, 0, SEEK_SET);
    fscanf(file, "%d %d %d", &M, &K, &N);

    // 分配内存，并读入矩阵 A 和矩阵 B
    A = (int **)malloc(M * sizeof(int *));
```

```
B = (int **)malloc(K * sizeof(int *));
C = (int **)malloc(M * sizeof(int *));
threads = (pthread_t **)malloc(M * sizeof(pthread_t *));

printf("malloc success.\n");
for(int i = 0; i < M; i++){
    A[i] = (int *)malloc(K * sizeof(int));
    C[i] = (int *)malloc(N * sizeof(int));
    threads[i] = (pthread_t *)malloc(N * sizeof(pthread_t));

    for(int j = 0; j < K; j++){
        fscanf(file, "%d", &A[i][j]);
        printf("%d\t",A[i][j]);
    }
    printf("\n");
}
printf("A load success.\n");

for(int i =0; i < K; i++){
    B[i] = (int *)malloc(N * sizeof(int));
    for(int j = 0; j < N; j++){
```

```
            fscanf(file, "%d", &B[i][j]);
            printf("%d\t",B[i][j]);
        }
        printf("\n");
    }
    printf("B load success.\n");
    fclose(file);
    // 创建 M * N 个工作线程
    for(int i = 0; i < M; i++){
        for(int j = 0; j < N; j++){
            struct v * data = (struct v *) malloc (sizeof(struct v));
            data->row = i;
            data->column = j;
            // 通过 data 创建工作线程
            pthread_create(&threads[i][j], NULL, work, data);
        }
    }

    // pthread_join 函数等待所有线程完成
    for (int i = 0; i < M; i++){
        for(int j = 0; j < N; j++){
```

```
            pthread_join(threads[i][j], NULL);
        }
    }

    // 打印结果矩阵 C
    printf("==============All threads has done, Result C :==================\n");
    for(int i = 0; i < M; i++){
        for(int j = 0; j < N; j ++){
            printf("%d\t",C[i][j]);
        }
        printf("\n");
    }

    // 擦屁股
    for(int i = 0; i < M; i++){
        free(A[i]);
        free(C[i]);
        free(threads[i]);
    }
    for(int j = 0; j < K; j++){
```

```
            free(B[j]);
        }
        free(A);
        free(B);
        free(C);
        free(threads);
        return 0;
}
```

4.3 测试报告

4.3.1 测试用例 1

输入数据

Input
3 3 3 1 2 3 4 5 6 7 8 9 9 8 7 6 5 4 3 2 1

运行结果截图

```
allpfirestorm@LAPTOP-O0JFQIE5:~/projects/bupt-homework/os/threads$ ./matrix test1.txt
open file success.
File format right.
malloc success.
1       2       3
4       5       6
7       8       9
A load success.
9       8       7
6       5       4
3       2       1
B load success.
```

图 4-1 测试用例 1 导入数据

```
(0,0): sum = 0, A[0][0]=1, B[0][0]=9
(0,0): sum = 9, A[0][1]=2, B[1][0]=6
(0,0): sum = 21, A[0][2]=3, B[2][0]=3
0,0: 30
(0,1): sum = 0, A[0][0]=1, B[0][1]=8
(0,1): sum = 8, A[0][1]=2, B[1][1]=5
(0,1): sum = 18, A[0][2]=3, B[2][1]=2
0,1: 24
(0,2): (1,1): sum = 0, A[1][0]=4, B[0][1]=8
(1,1): sum = 32, A[1][1]=5, B[1][1]=5
(1,1): sum = 57, A[1][2]=6, B[2][1]=2
1,1: 69
sum = 0, A[0][0]=1, B[0][2]=7
(0,2): sum = 7, A[0][1]=2, B[1][2]=4
(0,2): sum = 15, A[0][2]=3, B[2][2]=1
0,2: 18
(1,0): sum = 0, A[1][0]=4, B[0][0]=9
(1,0): sum = 36, A[1][1]=5, B[1][0]=6
(1,0): sum = 66, A[1][2]=6, B[2][0]=3
1,0: 84
(1,2): sum = 0, A[1][0]=4, B[0][2]=7
(1,2): sum = 28, A[1][1]=5, B[1][2]=4
(1,2): sum = 48, A[1][2]=6, B[2][2]=1
1,2: 54
(2,1): sum = 0, A[2][0]=7, B[0][1]=8
(2,1): sum = 56, A[2][1]=8, B[1][1]=5
(2,1): sum = 96, A[2][2]=9, B[2][1]=2
2,1: 114
(2,0): sum = 0, A[2][0]=7, B[0][0]=9
(2,0): sum = 63, A[2][1]=8, B[1][0]=6
(2,0): sum = 111, A[2][2]=9, B[2][0]=3
2,0: 138
(2,2): sum = 0, A[2][0]=7, B[0][2]=7
(2,2): sum = 49, A[2][1]=8, B[1][2]=4
(2,2): sum = 81, A[2][2]=9, B[2][2]=1
```

图 4-2 测试用例 1 工作过程

```
=============All threads has done, Result C :=================
30      24      18
84      69      54
138     114     90
```

图 4-3 测试用例 1 执行结果

测试结果分析

执行结果分析：

程序首先成功打开了一个文件，这是通过 "open file success." 提示来表明的。

然后，它通过 "File format right." 提示表明成功验证了文件的格式，包括正确

的维度和元素数量。接下来，程序成功地为矩阵 A 和矩阵 B 分配了内存，分别用 "A load success." 和 "B load success." 提示表明。然后，程序创建了一些线程并分配任务给它们。每个线程都执行矩阵相乘的一部分，并且输出了一些调试信息，包括正在计算的部分矩阵、累积的和，以及元素的值。最后，程序输出了结果矩阵 C 的值，并使用 "All threads have done, Result C:" 提示。

操作系统角度分析：

程序使用了多线程来实现并发计算。在创建线程时，操作系统会为每个线程分配处理器时间片，这使得线程可以交替运行，从而在多个处理器上共享计算负载。每个线程执行不同的任务，对输入矩阵 A 和 B 进行部分乘法，然后将结果存储在输出矩阵 C 中。程序通过 pthread 库来管理线程，包括线程的创建和等待线程完成。最终，操作系统协调线程的执行，确保它们正确地完成其任务。

4.3.2 测试用例 2

输入数据

Input
4 5 4 11 22 33 44 55 66 77 88 99 0 1 2 3 4 5 5 4 3 2 1 1 2 3 4 5 6 7 8

```
9   0   11   22

33   44   55   66

77   55   99   100
```

运行结果截图

```
allpfirestorm@LAPTOP-O0JFQIE5:~/projects/bupt-homework/os/threads$ ./matrix test2.txt
open file success.
File format right.
malloc success.
11      22      33      44      55
66      77      88      99      0
1       2       3       4       5
5       4       3       2       1
A load success.
1       2       3       4
5       6       7       8
9       0       11      22
33      44      55      66
77      55      99      100
B load success.
```

图 4-4 测试用例 2 导入数据

```
(0,0): sum = 0, A[0][0]=11, B[0][0]=1
(0,0): sum = 11, A[0][1]=22, B[1][0]=5
(0,0): sum = 121, A[0][2]=33, B[2][0]=9
(0,0): sum = 418, A[0][3]=44, B[3][0]=33
0,0: 1870
(0,1): sum = 0, A[0][0]=11, B[0][1]=2
(0,1): sum = 22, A[0][1]=22, B[1][1]=6
(0,1): sum = 154, A[0][2]=33, B[2][1]=0
(0,1): sum = 154, A[0][3]=44, B[3][1]=44
0,1: 2090
(0,2): sum = 0, A[0][0]=11, B[0][2]=3
(0,2): sum = 33, A[0][1]=22, B[1][2]=7
(0,2): sum = 187, A[0][2]=33, B[2][2]=11
(0,2): sum = 550, A[0][3]=44, B[3][2]=55
0,2: 2970
(0,3): sum = 0, A[0][0]=11, B[0][3]=4
(0,3): sum = 44, A[0][1]=22, B[1][3]=8
(0,3): sum = 220, A[0][2]=33, B[2][3]=22
(0,3): sum = 946, A[0][3]=44, B[3][3]=66
0,3: 3850
(1,0): sum = 0, A[1][0]=66, B[0][0]=1
(1,0): sum = 66, A[1][1]=77, B[1][0]=5
(1,0): sum = 451, A[1][2]=88, B[2][0]=9
(1,0): sum = 1243, A[1][3]=99, B[3][0]=33
1,0: 4510
```

图 4-5 测试用例 2 运行过程 1

```
(1,1): sum = 0, A[1][0]=66, B[0][1]=2
(1,1): sum = 132, A[1][1]=77, B[1][1]=6
(1,3): sum = 0, A[1][0]=66, B[0][3]=4
(2,0): sum = 0, A[2][0]=1, B[0][0]=1
(2,1): sum = 0, A[2][0]=1, B[0][1]=2
(2,1): sum = 2, A[2][1]=2, B[1][1]=6
(2,1): sum = 14, A[2][2]=3, B[2][1]=0
(2,1): sum = 14, A[2][3]=4, B[3][1]=44
2,1: 190
(1,3): sum = 264, A[1][1]=77, B[1][3]=8
(1,3): sum = 880, A[1][2]=88, B[2][3]=22
(1,3): sum = 2816, A[1][3]=99, B[3][3]=66
1,3: 9350
(1,2): sum = 0, A[1][0]=66, B[0][2]=3
(1,2): sum = 198, A[1][1]=77, B[1][2]=7
(1,2): sum = 737, A[1][2]=88, B[2][2]=11
(1,2): sum = 1705, A[1][3]=99, B[3][2]=55
1,2: 7150
(1,1): sum = 594, A[1][2]=88, B[2][1]=0
(1,1): sum = 594, A[1][3]=99, B[3][1]=44
(2,2): (2,0): sum = 1, A[2][1]=2, B[1][0]=5
(2,3): (2,0): sum = 11, A[2][2]=3, B[2][0]=9
1,1: 4950
(3,0): sum = 0, A[3][0]=5, B[0][0]=1
(3,0): sum = 5, A[3][1]=4, B[1][0]=5
(3,0): sum = 25, A[3][2]=3, B[2][0]=9
(3,1): (3,0): sum = 52, A[3][3]=2, B[3][0]=33
sum = 0, A[3][0]=5, B[0][1]=2
```

图 4-6 测试用例 2 运行过程 2

```
(3,1): sum = 10, A[3][1]=4, B[1][1]=6
(3,1): sum = 34, A[3][2]=3, B[2][1]=0
(3,1): sum = 34, A[3][3]=2, B[3][1]=44
3,1: 122
3,0: 118
(3,2): sum = 0, A[3][0]=5, B[0][2]=3
(3,2): sum = 15, A[3][1]=4, B[1][2]=7
(3,2): sum = 43, A[3][2]=3, B[2][2]=11
(3,2): sum = 76, A[3][3]=2, B[3][2]=55
3,2: 186
sum = 0, A[2][0]=1, B[0][2]=3
(2,2): sum = 3, A[2][1]=2, B[1][2]=7
(2,2): sum = 17, A[2][2]=3, B[2][2]=11
(2,2): sum = 50, A[2][3]=4, B[3][2]=55
2,2: 270
(2,0): sum = 38, A[2][3]=4, B[3][0]=33
2,0: 170
(3,3): sum = 0, A[3][0]=5, B[0][3]=4
(3,3): sum = 20, A[3][1]=4, B[1][3]=8
(3,3): sum = 52, A[3][2]=3, B[2][3]=22
(3,3): sum = 118, A[3][3]=2, B[3][3]=66
3,3: 250
sum = 0, A[2][0]=1, B[0][3]=4
(2,3): sum = 4, A[2][1]=2, B[1][3]=8
(2,3): sum = 20, A[2][2]=3, B[2][3]=22
(2,3): sum = 86, A[2][3]=4, B[3][3]=66
2,3: 350
```

图 4-7 测试用例 2 运行过程 3

```
==============All threads has done, Result C :=================
1870    2090    2970    3850
4510    4950    7150    9350
170     190     270     350
118     122     186     250
```

图 4-8 测试用例 2 运行结果

结果分析

执行结果分析：

程序首先成功打开了一个文件，这是通过"open file success."提示来表明的。然后，它通过"File format right."提示表明成功验证了文件的格式，包括正确的维度和元素数量。接下来，程序成功地为矩阵 A 和矩阵 B 分配了内存，分别用"A load success."和"B load success."提示表明。然后，程序创建了一些线程并分配任务给它们。每个线程都执行矩阵相乘的一部分，并且输出了一些调试信息，包括正在计算的部分矩阵、累积的和，以及元素的值。最后，程序输出了结果矩阵 C 的值，并使用"All threads have done, Result C:"提示。

操作系统角度分析：

程序使用了多线程来实现并发计算。在创建线程时，操作系统会为每个线程分配处理器时间片，这使得线程可以交替运行，从而在多个处理器上共享计算负载。每个线程执行不同的任务，对输入矩阵 A 和 B 进行部分乘法，然后将结果存储在输出矩阵 C 中。程序通过 pthread 库来管理线程，包括线程的创建和等待线程完成。最终，操作系统协调线程的执行，确保它们正确地完成其任务。

4.3.3 测试用例 3

输入数据

Input
5 6 1 2 3 4 5 6 1 2 3 4 5 6 1 2 3 4 5 6 1 2 3 4 5 6

1 2 3 4 5 6 1 2 3 4 5 0 0 0 0 0

运行结果截图

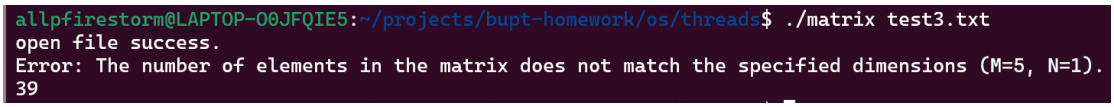

图 4-9 测试用例 3 运行结果

运行结果分析

首先，程序成功打开文件，这是"open file success." 提示表示的。

程序在读取文件中的矩阵数据后，进行了格式检查。然而，格式检查失败了，因为文件中的元素数量与指定的矩阵维度（M=5, N=1）不匹配。这是"Error: The number of elements in the matrix does not match the specified dimensions (M=5, N=1)." 提示表示的。这表明文件中的数据与指定的矩阵维度不一致，可能存在格式错误。

4.3.4 测试用例 4

数据输入

Input
5 6 5 1 2 3 4 5 6 1 2 3 4 5 6 1 2 3 4 5 6

```
1 2 3 4 5 6

1 2 3 4 5 6

1 2 3 4 5

0 0 0 0 0

5 4 3 2 4

5 3 2 1 1

5 4 3 2 1

1 1 1 1 1
```

运行结果截图

```
allpfirestorm@LAPTOP-O0JFQIE5:~/projects/bupt-homework/os/threads$ ./matrix test4.txt
open file success.
File format right.
malloc success.
1       2       3       4       5       6
1       2       3       4       5       6
1       2       3       4       5       6
1       2       3       4       5       6
1       2       3       4       5       6
A load success.
1       2       3       4       5
0       0       0       0       0
5       4       3       2       4
5       3       2       1       1
5       4       3       2       1
1       1       1       1       1
B load success.
```

图 4-10 测试用例 4 数据导入

```
(0,0): sum = 0, A[0][0]=1, B[0][0]=1
(0,0): sum = 1, A[0][1]=2, B[1][0]=0
(0,0): sum = 1, A[0][2]=3, B[2][0]=5
(0,1): (0,0): sum = 0, A[0][0]=1, B[0][1]=2
(0,1): sum = 2, A[0][1]=2, B[1][1]=0
(0,3): sum = 0, A[0][0]=1, B[0][3]=4
(0,4): sum = 0, A[0][0]=1, B[0][4]=5
(0,4): sum = 5, A[0][1]=2, B[1][4]=0
(0,4): sum = 5, A[0][2]=3, B[2][4]=4
(0,4): sum = 17, A[0][3]=4, B[3][4]=1
(0,3): sum = 16, A[0][3]=4, B[3][0]=5
(0,2): (0,1): sum = 2, A[0][2]=3, B[2][1]=4
(1,1): sum = 0, A[1][0]=1, B[0][1]=2
(1,1): sum = 2, A[1][1]=2, B[1][1]=0
(1,3): sum = 0, A[1][0]=1, B[0][3]=4
(2,0): sum = 0, A[2][0]=1, B[0][0]=1
(2,2): sum = 0, A[2][0]=1, B[0][2]=3
(2,3): sum = 0, A[2][0]=1, B[0][3]=4
(2,2): (2,3): sum = 4, A[2][1]=2, B[1][3]=0
(2,3): sum = 4, A[2][2]=3, B[2][3]=2
(1,2): (2,3): (3,2): sum = 0, A[1][0]=1, B[0][2]=3
(1,4): (3,3): sum = 0, A[3][0]=1, B[0][3]=4
(3,3): sum = 4, A[3][1]=2, B[1][3]=0
(4,0): sum = 0, A[4][0]=1, B[0][0]=1
(4,1): sum = 0, A[4][0]=1, B[0][1]=2
(4,1): sum = 2, A[4][1]=2, B[1][1]=0
(4,1): sum = 2, A[4][2]=3, B[2][1]=4
(4,1): sum = 14, A[4][3]=4, B[3][1]=3
(4,1): sum = 26, A[4][4]=5, B[4][1]=4
4,1: 46
(4,0): sum = 1, A[4][1]=2, B[1][0]=0
(4,0): sum = 1, A[4][2]=3, B[2][0]=5
(4,0): sum = 16, A[4][3]=4, B[3][0]=5
(4,0): sum = 36, A[4][4]=5, B[4][0]=5
4,0: 61
```

图 4-11 测试用例 4 运行过程 1

```
(4,0): sum = 1, A[4][1]=2, B[1][0]=0
(4,0): sum = 1, A[4][2]=3, B[2][0]=5
(4,0): sum = 16, A[4][3]=4, B[3][0]=5
(4,0): sum = 36, A[4][4]=5, B[4][0]=5
4,0: 61
(2,4): sum = 0, A[2][0]=1, B[0][4]=5
(2,4): sum = 5, A[2][1]=2, B[1][4]=0
(2,4): sum = 5, A[2][2]=3, B[2][4]=4
(2,4): sum = 17, A[2][3]=4, B[3][4]=1
(2,4): sum = 21, A[2][4]=5, B[4][4]=1
2,4: 26
sum = 3, A[2][1]=2, B[1][2]=0
(2,2): sum = 3, A[2][2]=3, B[2][2]=3
(2,2): sum = 12, A[2][3]=4, B[3][2]=2
(2,2): sum = 20, A[2][4]=5, B[4][2]=3
2,2: 35
(1,3): sum = 4, A[1][1]=2, B[1][3]=0
(1,3): sum = 4, A[1][2]=3, B[2][3]=2
(4,3): sum = 0, A[4][0]=1, B[0][3]=4
(4,3): sum = 4, A[4][1]=2, B[1][3]=0
(4,3): sum = 4, A[4][2]=3, B[2][3]=2
(4,3): sum = 10, A[4][3]=4, B[3][3]=1
(4,3): sum = 14, A[4][4]=5, B[4][3]=2
4,3: 24
(3,4): sum = 0, A[3][0]=1, B[0][4]=5
(3,4): sum = 5, A[3][1]=2, B[1][4]=0
(3,4): sum = 5, A[3][2]=3, B[2][4]=4
(3,4): sum = 17, A[3][3]=4, B[3][4]=1
(3,4): sum = 21, A[3][4]=5, B[4][4]=1
3,4: 26
(3,3): sum = 4, A[3][2]=3, B[2][3]=2
(3,3): sum = 10, A[3][3]=4, B[3][3]=1
(3,3): sum = 14, A[3][4]=5, B[4][3]=2
3,3: 24
```

图 4-12 测试用例 4 运行过程 2

```
3,0: 61
(1,1): sum = 2, A[1][2]=3, B[2][1]=4
(1,1): sum = 14, A[1][3]=4, B[3][1]=3
(1,1): sum = 26, A[1][4]=5, B[4][1]=4
1,1: 46
(1,2): sum = 3, A[1][1]=2, B[1][2]=0
(1,2): sum = 3, A[1][2]=3, B[2][2]=3
(1,2): sum = 12, A[1][3]=4, B[3][2]=2
(1,2): sum = 20, A[1][4]=5, B[4][2]=3
1,2: 35
(1,3): sum = 10, A[1][3]=4, B[3][3]=1
(1,3): sum = 14, A[1][4]=5, B[4][3]=2
1,3: 24
(0,4): sum = 21, A[0][4]=5, B[4][4]=1
0,4: 26
(1,0): sum = 0, A[1][0]=1, B[0][0]=1
(1,0): sum = 1, A[1][1]=2, B[1][0]=0
(1,0): sum = 1, A[1][2]=3, B[2][0]=5
(1,0): sum = 16, A[1][3]=4, B[3][0]=5
(1,0): sum = 36, A[1][4]=5, B[4][0]=5
1,0: 61
sum = 0, A[0][0]=1, B[0][2]=3
(0,2): sum = 3, A[0][1]=2, B[1][2]=0
(0,2): sum = 3, A[0][2]=3, B[2][2]=3
(0,2): sum = 12, A[0][3]=4, B[3][2]=2
(0,2): sum = 20, A[0][4]=5, B[4][2]=3
0,2: 35
```

图 4-13 测试用例 4 运行过程 3

```
(2,0): sum = 1, A[2][1]=2, B[1][0]=0
(2,0): sum = 1, A[2][2]=3, B[2][0]=5
(2,0): sum = 16, A[2][3]=4, B[3][0]=5
(2,0): sum = 36, A[2][4]=5, B[4][0]=5
2,0: 61
(4,4): sum = 0, A[4][0]=1, B[0][4]=5
(4,4): sum = 5, A[4][1]=2, B[1][4]=0
(4,4): sum = 5, A[4][2]=3, B[2][4]=4
(4,4): sum = 17, A[4][3]=4, B[3][4]=1
(4,4): sum = 21, A[4][4]=5, B[4][4]=1
4,4: 26
(4,2): sum = 0, A[4][0]=1, B[0][2]=3
(4,2): sum = 3, A[4][1]=2, B[1][2]=0
(4,2): sum = 3, A[4][2]=3, B[2][2]=3
(4,2): sum = 12, A[4][3]=4, B[3][2]=2
(4,2): sum = 20, A[4][4]=5, B[4][2]=3
4,2: 35
sum = 0, A[1][0]=1, B[0][4]=5
(1,4): sum = 5, A[1][1]=2, B[1][4]=0
(1,4): sum = 5, A[1][2]=3, B[2][4]=4
(1,4): sum = 17, A[1][3]=4, B[3][4]=1
(1,4): sum = 21, A[1][4]=5, B[4][4]=1
1,4: 26
sum = 10, A[2][3]=4, B[3][3]=1
(2,3): sum = 14, A[2][4]=5, B[4][3]=2
2,3: 24
sum = 0, A[3][0]=1, B[0][2]=3
(3,2): sum = 3, A[3][1]=2, B[1][2]=0
(3,2): sum = 3, A[3][2]=3, B[2][2]=3
(3,2): sum = 12, A[3][3]=4, B[3][2]=2
(3,2): sum = 20, A[3][4]=5, B[4][2]=3
3,2: 35
```

图 4-14 测试用例 4 运行过程 4

```
sum = 4, A[0][1]=2, B[1][3]=0
(0,3): sum = 4, A[0][2]=3, B[2][3]=2
(0,3): sum = 10, A[0][3]=4, B[3][3]=1
(0,3): sum = 14, A[0][4]=5, B[4][3]=2
0,3: 24
(0,0): sum = 36, A[0][4]=5, B[4][0]=5
0,0: 61
(0,1): sum = 14, A[0][3]=4, B[3][1]=3
(0,1): sum = 26, A[0][4]=5, B[4][1]=4
0,1: 46
(3,1): sum = 0, A[3][0]=1, B[0][1]=2
(3,1): sum = 2, A[3][1]=2, B[1][1]=0
(3,1): sum = 2, A[3][2]=3, B[2][1]=4
(3,1): sum = 14, A[3][3]=4, B[3][1]=3
(3,1): sum = 26, A[3][4]=5, B[4][1]=4
3,1: 46
(2,1): sum = 0, A[2][0]=1, B[0][1]=2
(2,1): sum = 2, A[2][1]=2, B[1][1]=0
(2,1): sum = 2, A[2][2]=3, B[2][1]=4
(2,1): sum = 14, A[2][3]=4, B[3][1]=3
(2,1): sum = 26, A[2][4]=5, B[4][1]=4
2,1: 46
```

图 4-15 测试用例 4 运行过程 5

```
=============All threads has done, Result C :=================
61      46      35      24      26
61      46      35      24      26
61      46      35      24      26
61      46      35      24      26
61      46      35      24      26
```

图 4-16 测试用例 4 运行结果

运行结果分析

执行结果分析：

程序首先成功打开了一个文件，这是通过 "open file success." 提示来表明的。然后，它通过 "File format right." 提示表明成功验证了文件的格式，包括正确的维度和元素数量。接下来，程序成功地为矩阵 A 和矩阵 B 分配了内存，分别用 "A load success." 和 "B load success." 提示表明。然后，程序创建了一些线程并分配任务给它们。每个线程都执行矩阵相乘的一部分，并且输出了一些调试信息，包括正在计算的部分矩阵、累积的和，以及元素的值。最后，程序输出了结果矩阵 C 的值，并使用 "All threads have done, Result C:" 提示。

操作系统角度分析：

程序使用了多线程来实现并发计算。在创建线程时，操作系统会为每个线程分配处理器时间片，这使得线程可以交替运行，从而在多个处理器上共享计算负载。每个线程执行不同的任务，对输入矩阵 A 和 B 进行部分乘法，然后将结果存储在输出矩阵 C 中。程序通过 pthread 库来管理线程，包括线程的创建和等待线程完成。最终，操作系统协调线程的执行，确保它们正确地完成其任务。