操 作 系 统
实 验 报 告

实验名称：实验三 多线程程序实验

姓名： 王迎旭

学号： 16340226

实验名称：**多线程程序实验**

一、实验目的：

1. 用线程生成Fibonacci 数列

2. 完成多线程矩阵乘法程序设计

二、实验要求：

1. 用pthread 线程库，按照第四章习题4.11 的要求生成并输出Fibonacci 数列

2. 矩阵乘法

给定两个矩阵A 和B，其中A 是具有M 行、K 列的矩阵， B 为K 行、N 列的矩阵， A 和B 的矩阵积为矩阵C， C 为M 行、N 列。矩阵C 中第i 行、第j 列的元素Cij 就是矩阵A 第i 行每个元素和矩阵B 第j 列每个元素乘积的和，即

$$C_{i,j} = \sum_{n=1}^{K} A_{i,n} \times B_{n,j}$$

要求：每个Ci j 的计算用一个独立的工作线程，因此它将会涉及生成M ×N 个工作线程。主线程(或称为父线程)将初始化矩阵A 和B，并分配足够的内存给矩阵C，它将容纳矩阵A 和B 的积。这些矩阵将声明为全局数据，以使每个工作线程都能访问矩阵A、B 和C。

三、实验过程：

（1）用线程生成Fibonacci 数列

4.11 Fibonacci 序列为0,1,1,2,3,5,8,…，通常，这可表达为：

$fib_0 = 0$

$fib_1 = 1$

$fib_n = fib_{n-1} + fib_{n-2}$

使用Java、Pthread 或Win32 线程库编写一个多线程程序来生成Fibonacci 序列。程序应这样工作：用户运行程序时在命令行输入要产生Fibonacci 序列的数，然后程序创建一个新的线程来产生Fibonacci 数，把这个序列放到线程共享的数据中（数组可能是一种最方便的数据结构）。当线程执行完成后，父线程将输出由子线程产生的序列。由于在子线程结束前，父线程不能开始输出Fibonacci 序列，因此父线程必须等待子线程的结束，这可采用4.3 节所述的技术。

1.明确目标程序的需求：

① 使用线程完成斐波那契数列的设计

② 理清线程之间的关系

2.搜索题目所涉及相关资料：

Ⅰ、线程问题相关函数介绍

① pthread_create()：创建线程。

② pthread_join()：阻塞调用线程，直到 threadid 所指定的线程终止。 每个线程只能用 pthread_join()一次。若多次调用就会发生逻辑错误。

③ pthread_exit()：终止调用线程。

④ pthread_attr_init ()：初始化线程属性为默认属性

⑤ pthread_attr_getscope ()：获得线程竞争范围

⑥ pthread_attr_setscope ()：设置线程竞争范围

Ⅱ、pthread_create()函数

功能：

用于线程的创建

例：

```
int pthread_create(pthread_t *restrict tid,
            const pthread_attr_t *restrict attr,
            void *(*start_rtn)(void),
            void *restrict arg);
```

参数详解：

① 第一个参数为指向线程标识符的指针，用于输出线程标识符。

② 第二个参数用来设置线程属性。

③ 第三个参数是线程运行函数的起始地址。

④ 最后一个参数是运行函数的参数。

返回值：

线程创建成功返回 0，否则返回错误编码

注：

在编译时注意加上-lpthread 参数，以调用静态链接库。因为 pthread 并非 Linux 系统的默认库。

Ⅲ、pthread_join()函数

功能：

这个函数是一个线程阻塞的函数，调用它的函数将一直等待到被等待的线程结束为止，当函数返回时，被等待线程的资源被收回

例：

```
int pthread_join __P (pthread_t __th,
                      void **__thread_return);
```

参数详解：

① 第一个参数为被等待的线程标识符

② 第二个参数为一个用户定义的指针，它可以用来存储被等待线程的返回值。

返回值：

成功调用返回 0，否则返回错误编码

3. 斐波拉契数列的程序设计与运行

Ⅰ、针对斐波拉契数列数列理清设计思路

① 创建斐波那契数列求值函数

```c
void * func(void *data)
{
    int *a = (int *) data ;
    for( int i = 2 ; i < number ; i ++ )
    {
        a[i] = a[i - 1] + a [i - 2] ;
    }
    pthread_exit(NULL);
}
```

② 限制斐波那契数列项数必须大于 2

```c
while(1)
{
    printf("Number:");
    scanf("%d",&number) ;
    if( number > 2)
    {
        break ;
    }
    else
    {
        printf("Number error\n");
    }
}
```

③ 为斐波那契数列动态分配空间

```c
int *a = (int*)malloc(sizeof(int) * 100) ;
a[0] = 0 ;
a[1] = 1 ;
```

④ 创建线程并完成非阻塞调用

```c
//创建线程指针
pthread_t th ;
//创建线程
pthread_create(&th,NULL,func,(void*)a);
//完成非阻塞调用
pthread_join(th,NULL);
```

⑤ 完成输出

```c
//完成输出
printf("Fibonacci:");
for(int i = 0 ; i < number ; i ++ )
{
    printf("%d ",a[i]);
}
```

Ⅱ、编译运行并输出结果

① 正常编译运行

```
[dell@localhost Desktop]$ gcc question.c -o question -pthread
[dell@localhost Desktop]$ ./question
Number:5
Fibonacci:0 1 1 2 3
[dell@localhost Desktop]$
```

② 输入数字错误情况

```
[dell@localhost Desktop]$ ./question
Number:2
Number error
Number:4
Fibonacci:0 1 1 2
[dell@localhost Desktop]$
```

分析：

A、编译时加上-pthread 指令，静态调用链接库，否则会出现编译错误的提示.

B、创当输入的参数小于等于 2 的时候，程序提示错误，要求用户重新输入，直到输入的参数大于 2.

（2） 多线程矩阵乘法

1、明确目标程序的需求：

① 通过使用多线程，完成矩阵乘法的运算

② 主线程完成程序初始化功能，子线程完成乘法运算

（注：由于两个题之间均使用线程相关知识，（2）不再展示相关资料，直接完成程序的设计）

2、完成矩阵乘法的程序设计：

① 申请三个矩阵 A,B,C 最大规格为 100*100

```c
//申请全局矩阵
int A[100][100];
int B[100][100];
int C[100][100];
```

② 使用结构体记录某次需要计算 C 矩阵的参数位置

```c
struct v
{
    int i, j;
};
```

③ 完成 A,B 矩阵的数据输入，C 的初始化

```c
//输入A矩阵
printf("The first Matrix, size(Number_one * Number_two)\n");
for (int i = 0; i < Number_one; i++) {
    for (int j = 0; j < Number_two; j++) {
        scanf("%d", &A[i][j]);
    }
}
//输入B矩阵
printf("The first Matrix, size(Number_two * Number_three)\n");
for (int i = 0; i < Number_two; i++) {
    for (int j = 0; j < Number_three; j++) {
        scanf("%d",&B[i][j]);
    }
}
//初始化C矩阵
for (int i = 0; i < Number_one; i++) {
    for (int j = 0; j < Number_three; j++) {
        C[i][j] = 0;
    }
}
```

④ 设计矩阵乘法函数

```c
//计算矩阵的乘积
void *calculate(void *data) {
    struct v *a = (struct v*)data;
    int i = a->i;
    int j = a->j;
    for (int k = 0; k < Number_two; k++) {
        C[i][j] += A[i][k] * B[k][j];
    }
    pthread_exit(NULL);
}
```

⑤ 完成子线程的创建

```c
//创建M*N个线程指针
pthread_t tid[Number_one * Number_three];
//设定线程属性
pthread_attr_t attr;
// 初始化线程属性为默认值
pthread_attr_init(&attr);
//为M*N个线程指针分配M*N个线程
for (int i = 0; i < Number_one; i++) {
    for (int j = 0; j < Number_three; j++) {
        struct v *a = (struct v*)malloc(sizeof(struct v));
        a -> i = i;
        a -> j = j;
        pthread_create(&tid[i * Number_three + j], &attr, calculate, (void*)a);
    }
}
```

注：

[

这里由于需要使用子线程完成对乘法的运算，所以 C 矩阵（规格 Number_one * Number_three）需要 Number_one * Number_three 个线程来完成相应的计算。

使用结构体也方便我们找出需要计算的位置，以及需要调用的线程在数组中的位置。

]

⑥ 非阻塞调用线程

```
//调用阻塞线程函数
for (int i = 0; i < Number_one * Number_three; i++) {
    pthread_join(tid[i], NULL);
}
```

⑦ 输出计算结果

```
//完成输出
printf("C matrix is:\n");
for (int i = 0; i < Number_one; i++) {
    for (int j = 0; j < Number_three; j++) {
        printf("%d ", C[i][j]);
    }
    printf("\n");
}
return 0;
```

3、编译与运行：

```
[dell@localhost Desktop]$ gcc question2.c -o question2 -pthread
[dell@localhost Desktop]$ ./question2
Number_one:3
Number_one:2
Number_one:3
The first Matrix, size(Number_one * Number_two)
1 2
3 4
5 6
The second Matrix, size(Number_two * Number_three)
1 2 3
4 5 6
C matrix is:
9 12 15
19 26 33
29 40 51
```

4. 验证运算结果正确性：

四、实验心得：

1. 单线程序设计比较简单，只需要使用一次线程调用就可以解决问题，但是在完成任务之前，还是需要对线程函数进行学习，比如 create 线程不成功会出现在什么情况下以及如何判断线程的终止，同时也是对线程函数的参数进行了了解，随后完成了第一部分的任务。

2. 进行矩阵乘法程序设计，在单线程的程序设计基础上需要再多考虑几个问题，比如：

①、如何使用子线程指针调用多子线程进行计算

②、如何准确记录需要运算的位置

③、如何为线程指针分配对应的线程

等问题。为了处理多子线程指针的问题，我创建了一个一维数组用于对应 C 矩阵中的某个元素，对应格式为数组[i*Number_three + j];在设计这个程序的过程中，也遇到了段错误的问题，仔细研究之后，发现是访问目标矩阵越界；所以，在访问数组元素时候，一定要先判断自己的访问是否是非法的。

3. 同时我也看到了多线程程序的 4 个优点：响应度高、资源共享、经济、多处理器体系结构的利用。