操 作 系 统

实 验 报 告

实验名称：实验三　多线程程序实验

姓名：陈亚楠

学号：16340041

实验名称：**多线程程序实验**

一、实验目的：

理解与掌握线程的相关概念。

二、实验要求：

1. 用线程生成 Fibonacci 数列：

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．多线程矩阵乘法。

三、实验过程：

实验 1.用线程生成 Fibonacci 数列：

1. 实验设计：

该程序通过一个独立线程计算 Fibonacci 数列。由于对于 Pthread 程序，独立线程是通过特定函数执行的，在本实验中，这个特定函数是 fibonacciFun() 函数。当程序开始时，单个控制线程在 main() 中开始。在初始化之后, main() 创建了第二个线程并在 fibonacciFun() 中开始控制。两个线程共享全局数据 fibonacci 数组。

现在对该程序做一个更为详细的描述。所有 Pthread 程序都需要包括 pthread.h 头文件。语句 pthread _t tid 声明了所创建线程的标识符。每个线程都有一组属性，包括栈大小和调度信息。pthread_attr_t attr 表示线程的属性，通过函数调用 pthread _attr_init(attr) 来设置这些属性。由于该程序不需要明确地设置任何属性，故使用提供的默认属性。通过函数调用 pthread_create() 创建一个独立线程。除了传递线程标识符和线程属性外，还要传递函数名称 fibonacciFun 以便新线程可以开始执行。最后传递由命令行参数 argv[1]所提供的整数参数。

程序此时有两个线程: main()的初始(父)线程和通过 fibonacciFun()函数执行计算 Fibonacci 数列线程。在创建了计算 Fibonacci 数列线程之后，父线程通过调用 pthread_join()函数，以等待计算 Fibonacci 数列线程的完成。计算 Fibonacci 数列线程在调用了函数 pthread_exit()之后就完成了。一旦计算 Fibonacci 数列线程返回，父线程将输出累加和的值。

2. 程序代码：

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

int fibonacci[100] = {0};

void* fibonacciFun(void* param);
int fun(int n);
```

```c
int main(int argc, char *argv[])
{
    if (argc != 2) {
        fprintf(stderr, "usage: a.out <integer value>\n");
        return -1;
    }
    if (atoi(argv[1]) < 0) {
        fprintf(stderr, "%d must be >= 0\n", atoi(argv[1]));
        return -1;
    }

    pthread_t tid;
    pthread_attr_t attr;

    pthread_attr_init(&attr);
    pthread_create(&tid, &attr, fibonacciFun, argv[1]);
    pthread_join(tid, NULL);

    for (int i = 0; i < atoi(argv[1]); i++)
    {
        printf("%d ", fibonacci[i]);
    }

    printf("\n");
    return 0;
}
```

```c
void* fibonacciFun(void* param)
{
    int n = atoi(param);
    fun(n);
    ptread_exit(0);
}

int fun(int n) {
    if (n <= 1) {
        fibonacci[n] = n;
        return fibonacci[n];
    }
    else {
        fibonacci[n] = fun(n - 1) + fun(n - 2);
        return fibonacci[n];
    }
}
```

3. 实验结果：

```
chen@ChenYanan:~/文档$ gcc pthread_fibonacci.c -o pthread_fibona
cci -pthread
chen@ChenYanan:~/文档$ ./pthread_fibonacci 5
0 1 1 2 3
chen@ChenYanan:~/文档$ ./pthread_fibonacci 20
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181
```

实验 2. 项目乘法：

1. 实验思路：

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

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

。

由于每个 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。

①静态初始化矩阵 A、B、C：

```
#define M 3
#define K 2
#define N 3
int A[M][K] = {{1, 4}, {2, 5}, {3, 6}};
int B[K][N] = {{8, 7, 6}, {5, 4, 3}};
int C[M][N] = {{0, 0, 0}, {0, 0, 0}, {0, 0, 0}};
```

②向每个线程传递参数：

父线程将生成 MxN 个工作线程，给每个线程传递行 i 和列 j 的值，工作线程利用行和列的值来计算矩阵积。这需要向每个线程传递两个参数。在本实验中利用 Pthread 中的 struct 生成一个数据结构：

```
struct value {
    int row;
    int col;
};
void* multiply(void* param);
```

并采用如下算法生成工作线程：

```
for(int i = 0; i < M; ++i) {
    for(int j = 0; j < N; ++j) {
        struct value *data = (struct value *)malloc(sizeof(struct value));
        data -> row = i;
        data -> col = j;
        pthread_create(&tid[i][j], &attr, multiply, (void*)data);
    }
}
```

数据指针将被传递给 pthread_create()函数，然后又将它作为参数传递给作为独立线程运行的函数。

③等待线程结束：

一旦所有的工作线程结束，主线程将输出包含在矩阵 C 中的积。这需要主线程等待所有的工作线程完成其工作，然后才能输出矩阵积的值。本实验用使用下面的方式来使主线程等待其他线程的结束：

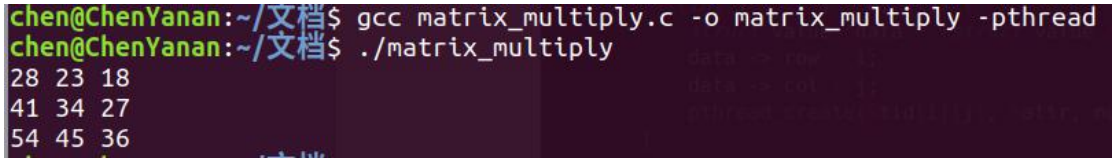
```
for(int i = 0; i < M; i++) {
    for(int j = 0; j < N; j++) {
        pthread_join(tid[i][j], NULL);
    }
}
```

2. 程序代码：

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

#define M 3
#define K 2
#define N 3
int A[M][K] = {{1, 4}, {2, 5}, {3, 6}};
int B[K][N] = {{8, 7, 6}, {5, 4, 3}};
int C[M][N] = {{0, 0, 0}, {0, 0, 0}, {0, 0, 0}};

struct value {
    int row;
    int col;
};
void* multiply(void* param);
```

```c
int main(int argc, char const *argv[]) {
    pthread_t tid[M][N];
    pthread_attr_t attr;

    pthread_attr_init(&attr);

    for(int i = 0; i < M; ++i) {
        for(int j = 0; j < N; ++j) {
            struct value *data = (struct value *)malloc(sizeof(struct value));
            data -> row = i;
            data -> col = j;
            pthread_create(&tid[i][j], &attr, multiply, (void*)data);
        }
    }

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

    for(int i = 0; i < M; i++) {
        for(int j = 0; j < N; j++) {
            printf("%d ", C[i][j]);
        }
        printf("\n");
    }
    return 0;
}
void *multiply(void *param) {
    struct value *mul = (struct value*)param;
    for(int i = 0; i < M; i++) {
        C[mul -> row][mul -> col] +=
                A[mul -> row][i] * B[i][mul -> col];
    }
}
```

3. 实验结果：


chen@ChenYanan:~/文档$ gcc matrix_multiply.c -o matrix_multiply -pthread
chen@ChenYanan:~/文档$ ./matrix_multiply
28 23 18
41 34 27
54 45 36