实验 2 多线程程序实验

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

我的代码如下：

先声明全局变量：

```
#define SIZE 1024
char fibonacci_string[SIZE];
```

判断输入参数，获得要生成的斐波那契数列的个数 num：

```
if (argv != 2) {
    print_error_message();
}

for (int i = 0; i < strlen(args[1]); i++) {
    if (!isdigit(args[1][i]))
        print_error_message();
}

int num = atoi(args[1]);
if (num == 0)
    print_error_message();
```

创建线程计算斐波那契数列,并把参数 num 传递过去：

```
pthread_t thread;
if (pthread_create(&thread, NULL, (void *)&get_fibonacci, (void *)&num) != 0) {
    printf("create thread fail.\n");
    exit(1);
}
```

等待线程完成斐波那契数列的计算：

```
void *retival;
if (pthread_join(thread, &retival) != 0)
    printf("cannot join with thread\n");
else
    printf("thread is end!\n");
```

最后等待线程结束后 main 函数输出斐波那契数列：

```
printf("%s\n", fibonacci_string);
printf("Done!\n");

exit(0);
```

关于斐波那契数列的计算：
先转换参数为 int 变量：

```
void get_fibonacci(void *n) {
    int num = *((int *)n);
    printf("thread begins!\n");
```

根据 num 值计算斐波那契数列：

```
if (num == 1) {
    sprintf(fibonacci_string, "%d", 0);
}
else {
    sprintf(fibonacci_string, "%d %d", 0, 1);
    int fib0 = 0, fib1 = 1;
    for (int i = 0; i < num; i++) {
        int fibonacci_string_length = strlen(fibonacci_string);
        if (i % 2 == 0) {
            fib0 += fib1;
            sprintf(fibonacci_string + fibonacci_string_length, " %d", fib0);
        }
        else {
            fib1 += fib0;
            sprintf(fibonacci_string + fibonacci_string_length, " %d", fib1);
        }
    }
}
```

最后加上个换行符

```
int fibonacci_string_length = strlen(fibonacci_string);
sprintf(fibonacci_string + fibonacci_string_length, "\n");
```

编译的时候一开始是使用 ppt 给的命令，但是有问题，后来在 stackoverflow 上找到 linux 的编译命令：

```
~/Desktop/os2$ gcc -pthread -o  part1 part1.c
```

运行结果如下：

```
nanxuan@nanxuan-Lenovo-G40-70m:~/Desktop/os2$ ./part1
Please input a positive number
nanxuan@nanxuan-Lenovo-G40-70m:~/Desktop/os2$ ./part1 0
Please input a positive number
nanxuan@nanxuan-Lenovo-G40-70m:~/Desktop/os2$ ./part1 1
thread begins!
thread is end!
0

Done!
nanxuan@nanxuan-Lenovo-G40-70m:~/Desktop/os2$ ./part1 10
thread begins!
thread is end!
0 1 1 2 3 5 8 13 21 34 55 89

Done!
nanxuan@nanxuan-Lenovo-G40-70m:~/Desktop/os2$ ./part1 45
thread begins!
thread is end!
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181 6765 10946 17711
28657 46368 75025 121393 196418 317811 514229 832040 1346269 2178309 3524578 5702
887 9227465 14930352 24157817 39088169 63245986 102334155 165580141 267914296 433
494437 701408733 1134903170 1836311903

Done!
```

2.多线程矩阵乘法

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

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。

我的代码参考了教材第四章结尾的实验提示，具体如下：
先声明全局变量：

```
#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];
```

声明一个结构体用来传递多参数：

```
struct para {
    int i;
    int j;
};
```

计算矩阵乘法的函数：

```
void calcultate_new_matrix(void *pa) {
    struct para *p = (struct para *)pa;
    int i = p->i;
    int j = p->j;
    C[i][j] = 0;
    for (int k = 0; k < K; k++) {
        C[i][j] += A[i][k] * B[k][j];
    }
}
```

在 main 函数里声明线程和结构体的二维数组

```
int main() {
    pthread_t thread[M][N];
    struct para pa[M][N];
```

创建线程，每个线程计算矩阵 C 的一个元素：

```
for (int i = 0; i < M; i++) {
    for (int j = 0; j < N; j++) {
        pa[i][j].i = i;
        pa[i][j].j = j;
        if (pthread_create(&thread[i][j], NULL, (void *)&calcultate_new_matrix, (void *)&pa[i][j]) != 0) {
            printf("create thread[%d][%d] fail.\n", i, j);
            exit(1);
        }
    }
}
```

等待所有线程结束：

```
void *retival;
for (int i = 0; i < M; i++) {
    for (int j = 0; j < N; j++) {
        if (pthread_join(thread[i][j], &retival) != 0)
            printf("cannot join with thread\n");
    }
}
```

最后输出矩阵：

```
for (int i = 0; i < M; i++) {
    for (int j = 0; j < N; j++) {
        printf("%d ", C[i][j]);
    }
    putchar('\n');
}
```

运行结果如下：

```
nanxuan@nanxuan-Lenovo-G40-70m:~/Desktop/os2$ gcc -pthread -o  part2 part2.c
nanxuan@nanxuan-Lenovo-G40-70m:~/Desktop/os2$ ./part2
28 23 18
41 34 27
54 45 36
```

经过验证，运行结果正确。

3.实验遇到的问题：

在进行第二个实验的时候，我的代码一开始是这样声明在循环中声明结构体的

```
for (int i = 0; i < M; i++) {
    for (int j = 0; j < N; j++) {
        struct para p;
        p.i = i;
        p.j = j;
        printf("p's add = %p\n", &p);
        if (pthread_create(&thread[i][j], NULL, (void *)&calcultate_new_matrix, (void *)&p) != 0) {
            printf("create thread[%d][%d] fail.\n", i, j);
            exit(1);
        }
    }
}
```

这样做，创建的线程无法正确获得参数，后来问了同学才知道是因为在循环中声明的结构体是局部变量，在那一次循环结束后，该变量生命周期结束，该指针指向的值会被销毁，该指针会指向别的结构体

为了证明是不是因为这样，我在创建进程前多加了一句打印出指针地址的语句，运行结果如下：

```
p 's add = 0x7ffdbcca75bf
p 's add = 0x7ffdbcca75bf
p 's add = 0x7ffdbcca75bf
p 's add = 0x7ffdbcca75bf
p 's add = 0x7ffdbcca75bf
p 's add = 0x7ffdbcca75bf
p 's add = 0x7ffdbcca75bf
p 's add = 0x7ffdbcca75bf
p 's add = 0x7ffdbcca75bf
```

因此在循环中同一个地址指向不同的值，前面循环结束后，地址会指向第二个循环的值，因此会出现不同线程通过参数获取的值是同一个的现象。

4.实验总结：

通过这次实验，以及在网上查的资料，学习了一些创建多线程的相关知识，在实验中遇到问题应该要多思考，多联系已经学过的知识。