习题四

1

> 用户级线程和内核级线程的映射模式有哪些？各有什么特点。

- 多对一模型：不支持内核线程的操作系统，内核只能看到一个进程，即多个线程不能并行地运行在多个处理器上；进程内线程的切换不会导致进程的切换；一个线程的系统调用会导致整个进程阻塞；可运行在不支持线程的操作系统中；允许进程定制调度算法，线程管理灵活；线程调度不需要内核参与，控制简单
- 一对一模型：用于支持线程的操作系统，用户线程映射到内核线程，由操作系统管理这些系统，即需要操作系统支持；并发性好，多个线程可以并行运行在多个处理器上；但是其内核开销大，线程调度可能会引起进程调度；多处理器系统中，同一进程中的线程能够并行在多个处理器上运行；进程中一个线程被阻塞，能切换同一进程内的其他线程继续执行。
- 多对多模型：多个用户线程映射为相等或更小数量的内核线程，并发性和效率兼顾，但是增加了模型的复杂度。
- 双层模型：仍然多路复用多个用户级线程到同样数量或更少数量的内核线程，但也允许绑定某个用户线程到一个内核线程。

2

> 线程库有什么作用？请举一个例子说明利用线程库创建线程的过程。

线程库的作用主要包括：

- 管理线程:线程库帮助程序员创建、管理和控制线程。它提供了一套API，使得程序员能够更容易地处理多线程编程的复杂性。
- 提高性能和响应速度通过允许多个操作并行执行，线程库可以帮助提高应用程序的性能和响应速度。
- 资源共享:线程共享相同的地址空间，这意味着它们可以更容易地共享资源，如内存和文件。
- 提高程序的可扩展性:线程库允许程序更好地利用多核处理器的能力，从而提高程序的可扩展性。

例子：使用Python的threading库创建线程

```
import threading

def print_hello_world():
    print("Hello, world!")

thread1 = threading.Thread(target=print_hello_world)
thread2 = threading.Thread(target=print_hello_world)

thread1.start()
thread2.start()

thread1.join()
thread2.join()
```

```
print("Both threads have finished execution!")
```

3

> 请举例说明为什么线程技术适合多处理器架构的计算机。

- 并发执行:在多处理器系统中，多线程技术可以允许不同的线程在不同的处理器上并发执行。这种并发可以显著提高程序的执行速度，因为它允许多个操作同时进行，而不是顺序执行。
- 负载均衡:线程技术可以帮助实现负载均衡，因为它可以分配不同的线程到不同的处理器上执行。通过合理的负载分配，系统能够更有效地利用所有可用的处理资源，从而提高整体性能。
- 资源共享:线程共享同一个进程空间，这意味着它们可以共享内存和其他资源。这种资源共享在多处理器系统中是有益的，因为它可以减少数据在不同处理器之间的传输，从而提高效率

4

> 一个多处理器系统中某个应用程序采用多对多线程模式编写。假如该程序的用户线程数量多于系统的处理器数量，讨论下列情况下的性能：
>
> 1. 该程序分配得到的内核线程的数量比处理器数量少
> 2. 该程序分配得到的内核线程的数量和处理器相同
> 3. 该程序分配得到的内核线程的数量大于处理器数量，但少于用户线程的数量

情况	性能
该程序分配得到的内核线程的数量比处理器数量少	差
该程序分配得到的内核线程的数量和处理器相同	一般
该程序分配得到的内核线程的数量大于处理器数量，但少于用户线程的数量	好

5

> 有两个 512*512 的整数矩阵，请用 *Pthreads* 库写一个程序，该程序利用4个线程来计算这两个矩阵的乘积。

```
#include <pthread.h>
#include <iostream>
#include <vector>
#include <cstdlib>
#include <ctime>
#include <fstream>

#define MAT_SIZE 512 // 题目要求的矩阵行列数
#define NUM_THREADS 4 // 题目要求使用的线程数

std::vector<std::vector<int>> A(MAT_SIZE, std::vector<int>(MAT_SIZE));
std::vector<std::vector<int>> B(MAT_SIZE, std::vector<int>(MAT_SIZE));
std::vector<std::vector<int>> C(MAT_SIZE, std::vector<int>(MAT_SIZE));
```

```
struct thread_data {
    int start_row;
    int end_row;
};

void* multiply(void* arg) {
    auto data = static_cast<thread_data*>(arg);
    int start = data->start_row;
    int end = data->end_row;

    for (int i = start; i < end; i++) {
        for (int j = 0; j < MAT_SIZE; j++) {
            C[i][j] = 0;
            for (int k = 0; k < MAT_SIZE; k++) {
                C[i][j] += A[i][k] * B[k][j];
            }
        }
    }
    return nullptr;
}

void printMatrixToCSVFile(const std::vector<std::vector<int>>& matrix,
const std::string& filename) {
    std::ofstream file(filename);

    if (!file) {
        std::cerr << "Unable to open the file " << filename << " for
writing." << std::endl;
        return;
    }

    for (const auto& row : matrix) {
        for (size_t i = 0; i < row.size(); ++i) {
            file << row[i];
            if (i < row.size() - 1) {
                file << ',';
            }
        }
        file << '\n';
    }

    file.close();
    std::cout << "Matrix has been written to " << filename << std::endl;
}

int main() {
    pthread_t threads[NUM_THREADS];
    thread_data thread_data_array[NUM_THREADS];

    std::srand(std::time(nullptr));

    for (int i = 0; i < MAT_SIZE; i++) {
        for (int j = 0; j < MAT_SIZE; j++) {
            A[i][j] = std::rand() % 10;
```

```
            B[i][j] = std::rand() % 10;
        }
    }

    int rows_per_thread = MAT_SIZE / NUM_THREADS;

    for (int t = 0; t < NUM_THREADS; t++) {
        thread_data_array[t].start_row = t * rows_per_thread;
        thread_data_array[t].end_row = (t+1) * rows_per_thread;
        pthread_create(&threads[t], nullptr, multiply,
&thread_data_array[t]);
    }

    for (auto & thread : threads) {
        pthread_join(thread, nullptr);
    }

    printMatrixToCSVFile(A, "matrix_A.csv");
    printMatrixToCSVFile(B, "matrix_B.csv");
    printMatrixToCSVFile(C, "matrix_C.csv");

    return 0;
}
```

其中,三个矩阵的内容被分别存入了三个CSV文件以便阅读

矩阵A与B的值均为0-9之间的随机数

三个矩阵的内容分别被存入了

- matrix_A.csv
- matrix_B.csv
- matrix_C.csv