总 评 成 绩	

《高性能计算》实验报告册

学年学期：2023 - 2024 学年第 2 学期

学生姓名：高星杰

学生学号：202130722071 2

专业年级：计算机科学与技术 21 级

任课教师：郑 芳

华中农业大学信息学院

2024 年 3 月

实 验 报 告 册

实验名称	实验二　pthread 并行程序设计	实 验 成 绩	
实验日期	2024 年 3 月 27 日星期三 第 9-12 节		

一、**实验目的**（描述本实验的学习目的及你对本实验的学习预期。）

1、掌握 pthread 程序的设计思想、设计原则和设计方法。

2、掌握 pthread 并行程序的编程方法。

3、掌握并行程序的评估方法

4、掌握忙等待、互斥量及条件变量等同步互斥的使用方法

二、**实验环境**（请描述本实验教学活动所使用的实际环境。）

1、笔记本一台

2、Mac 系统

3、ITC Linux pthread 实验环境

三、**实验任务**（本实验要求的实验任务完成情况，未完成注明原因。）

1、完成编写完整的 pthread 矩阵-向量程序。

2、完成利用忙等待（实现了改进的 Peterson's Algorithm）、互斥量和条件变量来编写求∏值的 pthread 程序。

3、完成用信号量和条件变量编写发送消息的 pthread 程序

四、实验内容

要求：采用 pthread 多线程编程实现以下三个题目。

(一）实验题目

（1）编写完整的 Pthreads 矩阵-向量程序。

（2）利用忙等待、互斥量及条件变量来编写求∏值的 Pthreads 程序。

（3）用信号量或者条件变量编写发送消息的 Pthreads 程序（可参看教材程序 4-7、4-8 改写）。

(二）实验过程

（1） 集群使用 （以一题为示例即可）

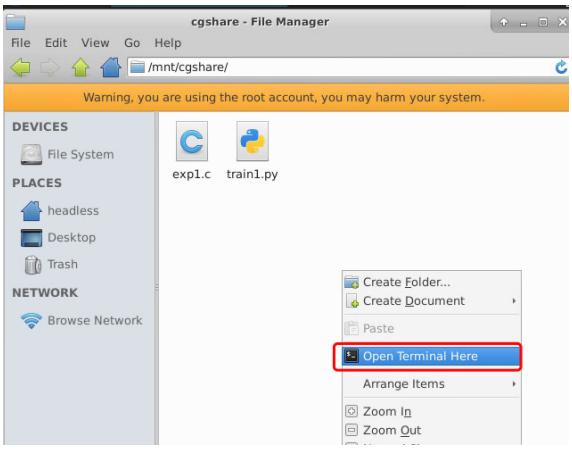
1）在本地编写好 pthread c/c++语言程序

2）在本地编译运行测试程序可行性和正确性

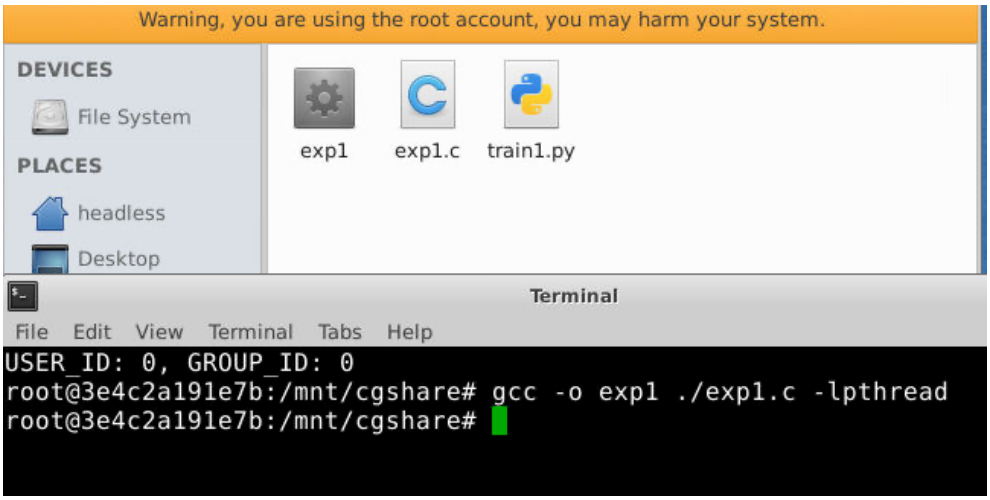
3）编写测试 python 程序（可选）

4）将代码上传至 itc pthread 环境

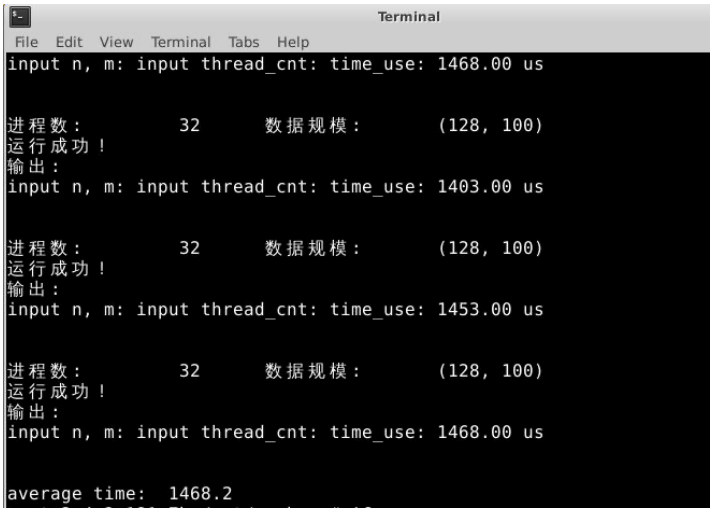
5）在/mnt/cgshare 目录下打开终端



6）使用命令行 gcc matrix.c -o 程序名 -lpthread 来编译源代码，生成可执行程序



7）使用命令行./程序名 进程数 来运行程序（或者使用 python3 ./测试程序.py 直接循环测试所有数据规模和进程数）。



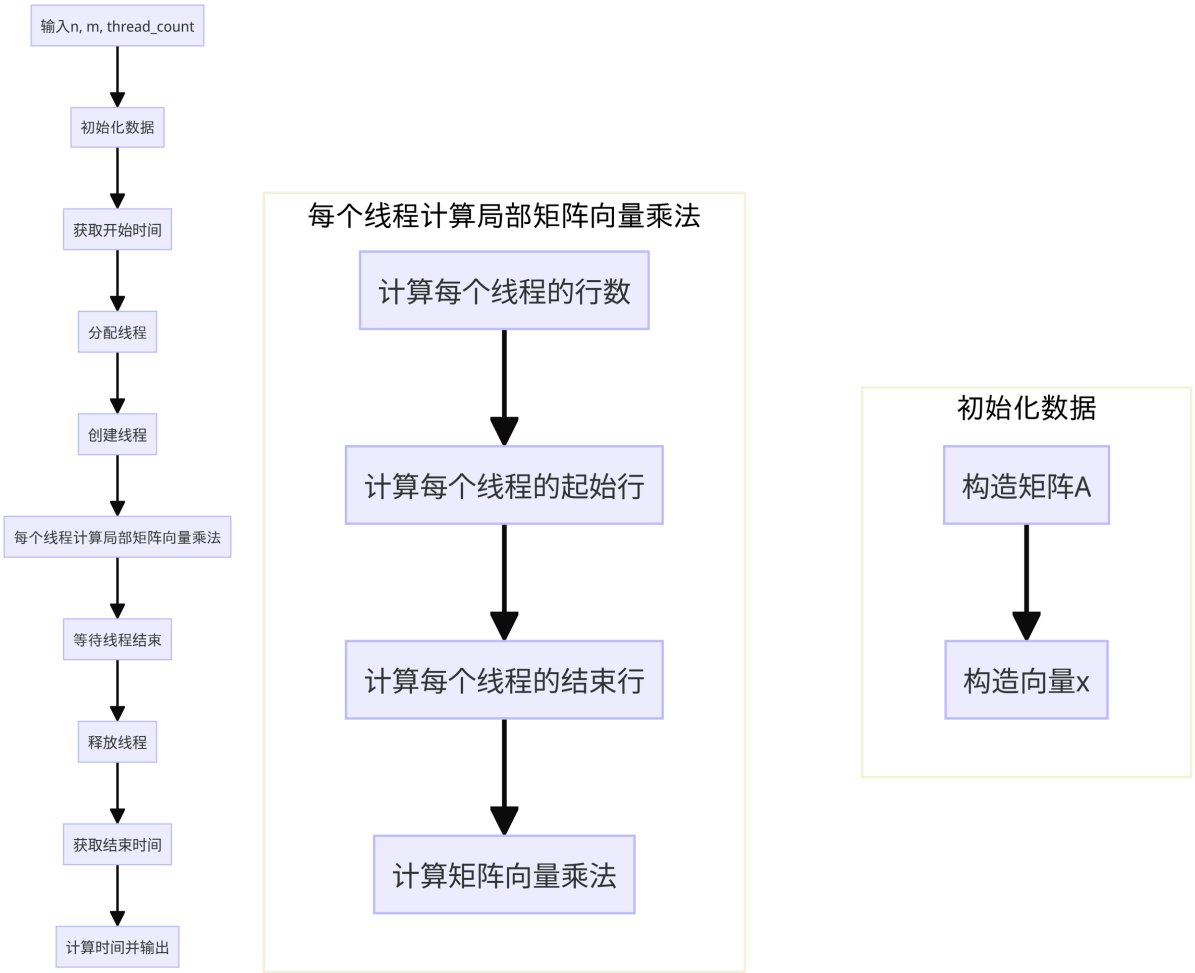
（2） 源码及解析（每题一个源码）

1) 矩阵-向量乘法

原理：

矩阵向量乘法是矩阵的每一行中每个元素乘向量中列的对应元素，每行计算互不影响，所以我们就可以把每一行的计算作为一个原子问题分配给线程，以实现并行计算。但是这里相较于 MPI 比较简单，因为 MPI 需要考虑到进程之间的通信和数据交换，而 pthread 可以直接使用共享变量来实现。

流程图：



代码解析：

1. init_data()

- 功能：初始化矩阵`A`和向量`x`。矩阵`A`的每个元素被设置为其行索引，向量`x`的每个元素被设置为其索引值。
- 参数：无。
- 返回值：无。

2. pthread_mat_vect(void *rank)

- 功能：线程执行的函数，用于计算矩阵`A`和向量`x`的乘积的一部分。每个线程负责计算结果向量`y`的一部分。
- 参数：
  - ✧ void *rank：线程的排名或编号，用于计算该线程应处理的矩阵行的范围。
- 返回值：`NULL`。

3. main(int argc, char *argv[])

- 功能：
  a) 读取用户输入的矩阵尺寸（`n`行，`m`列）和线程数量（`thread_count`）。
  b) 调用`init_data`函数初始化矩阵和向量。
  c) 使用`gettimeofday`函数获取开始时间。
  d) 分配线程句柄数组 thread_handles 并创建线程，每个线程执行 pthread_mat_vect 函数。
  e) 等待所有线程完成计算。
  f) 释放线程句柄数组。
  g) 使用`gettimeofday`函数获取结束时间，并计算总的运行时间。

h) 打印运行时间。

- 参数：
  - `int argc`：命令行参数的数量。
  - `char *argv[]`：命令行参数的字符串数组。
- 返回值：程序的退出状态。`0`表示成功。

实现了使用多线程技术来加速矩阵与向量乘法的计算过程。通过将矩阵分割成多个部分，每个线程负责计算一部分的结果，从而实现并行计算。

代码详细解析

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <sys/time.h>

#define MAX 10000

int thread_count; // number of threads
int n;          // row
int m;          // column

double A[MAX][MAX]; // matrix input
double x[MAX];      // vector input
double y[MAX];      // result
```

```
void *pthread_mat_vect(void *rank);
void init_data();

int main(int argc, char *argv[])
{
    long thread;
    pthread_t *thread_handles;
    struct timeval start, end;
    double time_use;

    printf("input n, m: ");
    scanf("%d%d", &n, &m);
    printf("input thread_cnt: ");
    scanf("%d", &thread_count);
    // 初始化数据 Initialize data
    init_data();

    // 获取开始时间 Get start time
    gettimeofday(&start, NULL);
    // 分配线程 Allocate threads
    thread_handles     =     (pthread_t     *)malloc(thread_count     *
sizeof(pthread_t));

    // 创建线程 Create threads
```

```
    for (thread = 0; thread < thread_count; thread++)
    {
        pthread_create(&thread_handles[thread], NULL, pthread_mat_vect,
(void *)thread);
    }
    // 等待线程结束 Wait for the thread to end
    for (thread = 0; thread < thread_count; thread++)
    {
        pthread_join(thread_handles[thread], NULL);
    }
    // 释放线程 Free thread
    free(thread_handles);
    gettimeofday(&end, NULL);

    // 计算时间 Calculate time consumption
    time_use = (end.tv_sec - start.tv_sec) * 1000000 + (end.tv_usec -
start.tv_usec);
    printf("time_use: %.2lf us\n", (double)time_use);
    return 0;
}
// 初始化矩阵和向量 Initialize data
void init_data()
{
    int i, j;
    // 构造矩阵
```

```
    for (i = 0; i < n; i++)
    {
        for (j = 0; j < m; j++)
        {
            A[i][j] = i;
        }
    }
    // 构造向量
    for (i = 0; i < m; i++)
    {
        x[i] = i;
    }
}
// 矩阵向量乘法 Matrix vector multiplication
void *pthread_mat_vect(void *rank)
{
    long my_rank = (long)rank;
    int i, j;
    // 计算每个线程的行数 Calculate the number of rows for each thread
    int local_m = (m % thread_count != 0) ? ((int)(m / thread_count) + 1) :
(m / thread_count);
    // 计算每个线程的起始行 Calculate the starting row of each thread
    int my_first_row = my_rank * local_m;
    // 计算每个线程的结束行 Calculate the end row of each thread
    int my_last_row = ((my_rank + 1) * local_m - 1) > m - 1 ? (m - 1) : ((my_rank
```

```
+ 1) * local_m - 1);
    // 计算矩阵向量乘法 Calculate matrix vector multiplication
    for (i = my_first_row; i <= my_last_row; i++)
    {
        y[i] = 0.0;
        for (j = 0; j < n; j++)
        {
            y[i] += A[i][j] * x[j];
        }
    }
    return NULL;
}
```

2) 利用忙等待、互斥量及条件变量来编写求Π值

原理：

并行计算 π 的近似值可以使用莱布尼茨级数来计算 π。莱布尼茨级数的公式如下：

$$x = 4\sum_{k=0}^{\infty}\frac{(-1)^k}{2k+1}$$

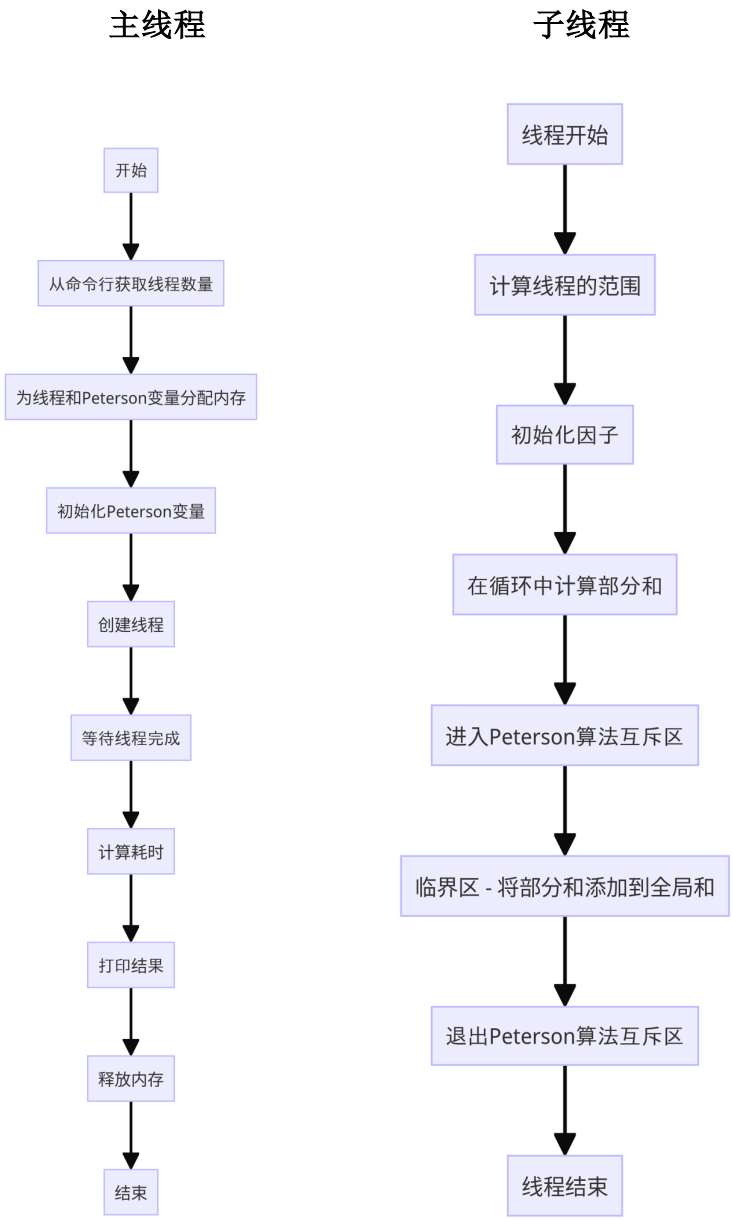
该公式将 π 的计算转化为一个无穷级数求和。程序将这个求和过程分配给多个线程并行执行，以加速计算。但是我们在求和的时候计算完毕，需要将结果告知，而这里不同于 MPI 我们是使用的共享变量实现的通信，**所以就涉及到了互斥，在访问 sum 的时候其他线程不能访问。互斥的解决办法有：基本满足忙等待的办法（扩展的 Peterson's Algorithm）、互斥量和条件变量。**

下面就逐个编码运行

a. 扩展的 Peterson’s Algorithm(满足忙等待的互斥算法)

Peterson 算法是一种经典的解决两个进程或线程互斥访问共享资源问题的算法，通过使用两个布尔标志变量和一个“转弯”变量来确保在任何时刻只有一个进程能够进入临界区，从而防止竞争条件的发生，并确保公平性和避免饥饿。

算法流程图：



代码+逐行注释

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <time.h>
#include <sys/time.h>

int thread_count;  // 线程数量
int n = 100000000; // 计算的项数，这里是10^8
double sum = 0.0;  // 全局求和变量

// Peterson 算法使用的变量
int *flag;
int turn;

void *thread_calulate_to_sum(void *rank); // 线程函数声明

int main(int argc, char *argv[])
{
    struct timeval start;
    struct timeval end; // 结构体用于存储时间，用于计算程序运行时间
    long thread;
    pthread_t *thread_handles = NULL; // 线程句柄数组
```

```
    gettimeofday(&start, NULL); // 获取开始时间

    thread_count = strtol(argv[1], NULL, 10);                                    //
从命令行获取线程数量
    thread_handles      =      (pthread_t      *)malloc(thread_count      *
sizeof(pthread_t)); // 为线程分配内存

    // 分配并初始化Peterson算法所需的变量
    flag = (int *)malloc(thread_count * sizeof(int));
    for (thread = 0; thread < thread_count; thread++)
    {
        flag[thread] = 0;
    }
    turn = 0;

    for (thread = 0; thread < thread_count; thread++)
    { // 创建线程
        pthread_create(&thread_handles[thread],                          NULL,
thread_calulate_to_sum, (void *)thread);
    }

    printf("Hello from the main thread\n");

    for (thread = 0; thread < thread_count; thread++)
    { // 等待所有线程完成
```

```
        pthread_join(thread_handles[thread], NULL);
    }
    gettimeofday(&end, NULL); // 获取结束时间
    long long startusec = start.tv_sec * 1000000 + start.tv_usec;
    long long endusec = end.tv_sec * 1000000 + end.tv_usec;
    double elapsed = (double)(endusec - startusec); // 计算运行时间
    printf("the result of π took %.2f us\n", elapsed);
    free(thread_handles); // 释放线程句柄内存
    free(flag);           // 释放Peterson算法变量内存
    printf("sum: %lf\n", 4 * sum); // 打印π的近似值
    return 0;
}

void *thread_calulate_to_sum(void *rank)
{
    long my_rank = (long)rank; // 线程标识
    double factor, my_sum = 0.0;
    long long i;
    long long my_n = n / thread_count;      // 每个线程处理的项数
    long long my_first_i = my_n * my_rank;   // 线程处理的第一个项的索引
    long long my_last_i = my_first_i + my_n; // 线程处理的最后一个项的索引
    if (my_first_i % 2 == 0)
        factor = 1.0;
    else
        factor = -1.0;
```

```
    for (i = my_first_i; i < my_last_i; i++, factor = -factor)
    {
        my_sum += factor / (2 * i + 1); // 计算部分和
    }
    // 使用Peterson 算法实现互斥
    flag[my_rank] = 1;
    turn = my_rank;
    for (long j = 0; j < thread_count; j++)
    {
        if (j != my_rank)
        {
            while (flag[j] && turn == my_rank)
                ; // 忙等待
        }
    }
    sum += my_sum; // 将部分和加到全局变量
    flag[my_rank] = 0; // 退出临界区
    return NULL;
}
```

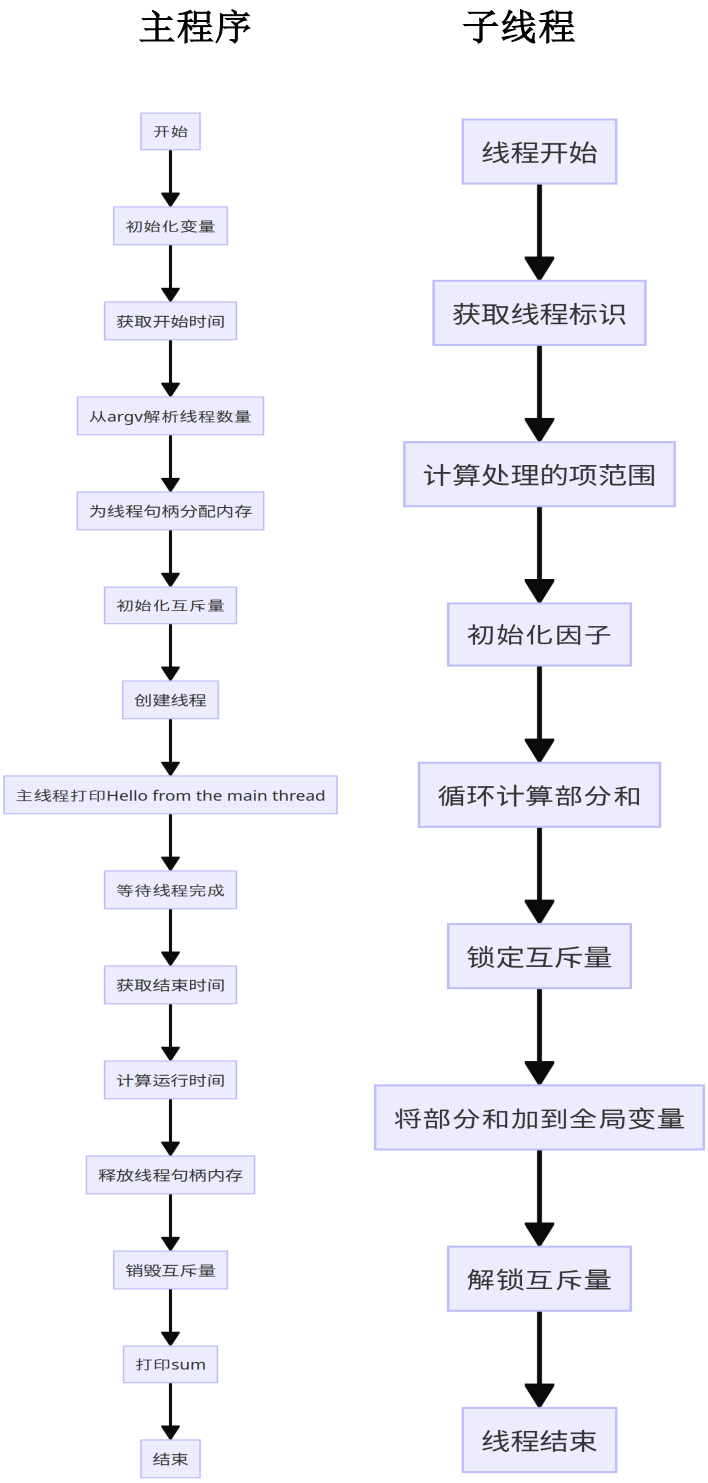
b. 互斥量

互斥量（Mutex）是一种同步机制，用于保护共享资源在多线程环境下的安全访问。通过锁定（lock）和解锁（unlock）操作，互斥量确保在任意时刻只有一个线程能够访问临界区，从而避免数据竞争和不一致的问题。当一个线程锁定互斥量后，其他尝试锁定的线程将被阻塞，直到互斥量被解锁为止。

使用互斥量来实现互斥，其实我们可以直接用互斥量 mutex 来代替 Peterson 算法的互斥机制。互斥量通过 pthread_mutex_lock 和 pthread_mutex_unlock 来控制对共享资

源 sum 的访问，确保只有一个线程可以在临界区中操作。这样就实现了互斥访问全局变量 sum 的目的。

流程图



代码+逐行解释

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <time.h>
#include <sys/time.h>

int thread_count;  // 线程数量
int n = 100000000; // 计算的项数，这里是10^8
double sum = 0.0;  // 全局求和变量

pthread_mutex_t mutex; // 互斥量

void *thread_calculate_to_sum(void *rank); // 线程函数声明

int main(int argc, char *argv[])
{
    struct timeval start;
    struct timeval end; // 结构体用于存储时间，用于计算程序运行时间
    long thread;
    pthread_t *thread_handles = NULL; // 线程句柄数组

    gettimeofday(&start, NULL); // 获取开始时间
```

```
    thread_count = strtol(argv[1], NULL, 10);                                  //
从命令行获取线程数量
    thread_handles      =      (pthread_t      *)malloc(thread_count      *
sizeof(pthread_t)); // 为线程分配内存

    // 初始化互斥量
    pthread_mutex_init(&mutex, NULL);

    for (thread = 0; thread < thread_count; thread++)
    { // 创建线程
        pthread_create(&thread_handles[thread],                        NULL,
thread_calculate_to_sum, (void *)thread);
    }

    printf("Hello from the main thread\n");

    for (thread = 0; thread < thread_count; thread++)
    { // 等待所有线程完成
        pthread_join(thread_handles[thread], NULL);
    }

    gettimeofday(&end, NULL); // 获取结束时间

    long long startusec = start.tv_sec * 1000000 + start.tv_usec;
    long long endusec = end.tv_sec * 1000000 + end.tv_usec;
```

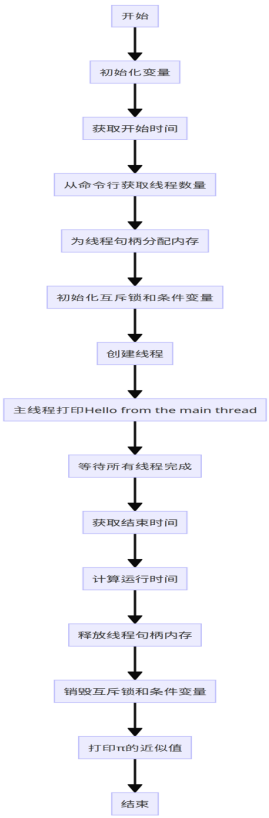
```
    double elapsed = (double)(endusec - startusec); // 计算运行时间
    printf("the result of π took %.2f us\n", elapsed);

    free(thread_handles);          // 释放线程句柄内存
    pthread_mutex_destroy(&mutex); // 销毁互斥量

    printf("sum: %lf\n", 4 * sum); // 打印π的近似值
    return 0;
}

void *thread_calculate_to_sum(void *rank)
{
    long my_rank = (long)rank; // 线程标识
    double factor, my_sum = 0.0;
    long long i;
    long long my_n = n / thread_count;       // 每个线程处理的项数
    long long my_first_i = my_n * my_rank;   // 线程处理的第一个项的索引
    long long my_last_i = my_first_i + my_n; // 线程处理的最后一个项的索引
    if (my_first_i % 2 == 0)
        factor = 1.0;
    else
        factor = -1.0;
    for (i = my_first_i; i < my_last_i; i++, factor = -factor)
    {
        my_sum += factor / (2 * i + 1); // 计算部分和
```

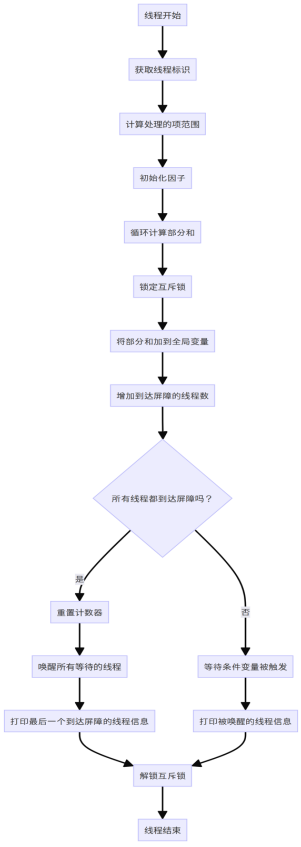
```
    }

    // 使用互斥量实现互斥
    pthread_mutex_lock(&mutex);   // 进入临界区
    sum += my_sum;                // 将部分和加到全局变量
    pthread_mutex_unlock(&mutex); // 退出临界区
    return NULL;
}
```

c. 条件变量

条件变量（Condition Variable）是一种线程同步机制，用于使线程在某些条件满足前进行等待。当某个条件发生变化时，一个线程可以通知其他在条件变量上等待的线程继续执行。条件变量通常与互斥量一起使用，确保对共享资源的安全访问和线程的同步。通过条件变量，可以实现线程间的高效等待和通知机制，避免忙等待，提高程序的并发性能。

流程图





代码+逐行注释

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <time.h>
#include <sys/time.h>

int thread_count;  // 线程数
int n = 100000000; // 10^8 数据规模，用于π计算的项数
double sum = 0.0;  // 全局变量，用于累加π的近似值
int flag = 0;      // 未使用的变量
int count = 0;     // 用于记录到达屏障（barrier）的线程数
pthread_mutex_t mutex; // 互斥锁，用于保护条件变量和全局变量sum
pthread_cond_t cond_var; // 条件变量，用于线程间同步

void *thread_calculate_to_sum(void *rank); // 线程函数声明

int main(int argc, char *argv[])
{
    struct timeval start;
    struct timeval end; // 用于计算程序运行时间的结构体
    long thread;        // 用于循环中的线程索引
    pthread_t *thread_handles = NULL; // 线程句柄数组
```

```
    gettimeofday(&start, NULL); // 获取开始时间

    thread_count = strtol(argv[1], NULL, 10); // 从命令行参数获取线程数
    thread_handles     =     (pthread_t     *)malloc(thread_count     *
sizeof(pthread_t)); // 为线程句柄分配内存

    pthread_mutex_init(&mutex, NULL); // 初始化互斥锁
    pthread_cond_init(&cond_var, NULL); // 初始化条件变量

    for (thread = 0; thread < thread_count; thread++)
    { // 创建线程
        pthread_create(&thread_handles[thread],                    NULL,
thread_calculate_to_sum, (void *)thread);
    }

    printf("Hello from the main thread\n");

    for (thread = 0; thread < thread_count; thread++)
    { // 等待所有线程完成
        pthread_join(thread_handles[thread], NULL);
    }

    gettimeofday(&end, NULL); // 获取结束时间

    long long startusec = start.tv_sec * 1000000 + start.tv_usec;
```

```
    long long endusec = end.tv_sec * 1000000 + end.tv_usec;
    double elapsed = (double)(endusec - startusec); // 计算运行时间
    printf("the result of π took %.2f us\n", elapsed);

    free(thread_handles); // 释放线程句柄内存
    pthread_mutex_destroy(&mutex); // 销毁互斥锁
    pthread_cond_destroy(&cond_var); // 销毁条件变量

    printf("sum: %f\n", 4 * sum); // 打印π的近似值
    return 0;
} // main
void *thread_calculate_to_sum(void *rank)
{
    long my_rank = (long)rank; // 线程标识
    double factor, my_sum = 0.0;
    long long i;
    long long my_n = n / thread_count; // 每个线程处理的项数
    long long my_first_i = my_n * my_rank; // 线程处理的第一个项的索引
    long long my_last_i = my_first_i + my_n; // 线程处理的最后一个项的索引
    if (my_first_i % 2 == 0)
        factor = 1.0; // 根据项的索引决定加减
    else
        factor = -1.0;
    for (i = my_first_i; i < my_last_i; i++, factor = -factor)
    {
```

```
        my_sum += factor / (2 * i + 1); // 计算部分和
    }

    pthread_mutex_lock(&mutex); // 锁定互斥锁
    sum += my_sum; // 将部分和加到全局变量
    count++; // 增加到达屏障的线程数
    if (count == thread_count)
    { // 如果所有线程都到达了屏障
        count = 0; // 重置计数器
        pthread_cond_broadcast(&cond_var); // 唤醒所有等待的线程
        printf("%ld(the last thread) has arrive at barrier\n", my_rank);
    }
    else
    { // 如果还有线程未到达屏障
        while (pthread_cond_wait(&cond_var, &mutex) != 0)
            ; // 等待条件变量被触发
        printf("%ld wake up\n", my_rank);
    }
    pthread_mutex_unlock(&mutex); // 解锁互斥锁
    return NULL;
}
```

3) 用信号量或者条件变量编写发送消息

原理：

使用条件变量的原理：条件变量和互斥锁配合使用实现线程间同步。线程发送消息后，

使用条件变量通知目标线程。接收消息的线程在条件变量上等待，直到收到通知再读取消息。互斥锁保护对共享资源的访问，确保线程安全。

使用信号量的原理：信号量用于线程间同步。发送消息的线程使用 sem_post 通知目标线程消息已到达。接收消息的线程使用 sem_wait 等待信号量，确保在消息到达前阻塞。信号量的增减操作控制线程的执行顺序，保证消息正确传递。

代码+逐行注释

a. 使用信号量实现

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h> // 引入信号量相关的头文件
#include <string.h>

#define MSG_MAX 100 // 定义消息的最大长度

int thread_count;  // 线程数量
char **messages;   // 存储每个线程发送的消息的数组
sem_t *semaphores; // 为每个线程创建的信号量数组

void *Send_msg(void *rank); // 线程函数声明

int main(int argc, char *argv[])
```

```
{
    long thread;                            // 用于循环的线程索引
    pthread_t *thread_handles;              // 存储线程句柄的数组
    thread_count = strtol(argv[1], NULL, 10); // 从命令行参数获取线程数量

    // 为线程句柄、消息数组和信号量数组分配内存
    thread_handles = malloc(thread_count * sizeof(pthread_t));
    messages = malloc(thread_count * sizeof(char *));
    semaphores = malloc(thread_count * sizeof(sem_t));

    for (thread = 0; thread < thread_count; thread++)
    {
        messages[thread] = NULL;            // 初始化消息数组
        sem_init(&semaphores[thread], 0, 0); // 初始化每个线程的信号量
    }

    for (thread = 0; thread < thread_count; thread++)
    {
        // 创建线程，传递线程编号作为参数
        pthread_create(&thread_handles[thread],  NULL,  Send_msg,  (void
*)thread);
    }

    for (thread = 0; thread < thread_count; thread++)
    {
```

```
        // 等待所有线程完成
        pthread_join(thread_handles[thread], NULL);
    }

    // 清理资源：释放消息内存，销毁信号量，释放线程句柄数组
    for (thread = 0; thread < thread_count; thread++)
    {
        free(messages[thread]);
        sem_destroy(&semaphores[thread]);
    }
    free(messages);
    free(semaphores);
    free(thread_handles);
    return 0;
}

void *Send_msg(void *rank)
{
    long my_rank = (long)rank;                          // 线程编号
    long dest = (my_rank + 1) % thread_count;           // 计算消息的
目标线程
    long source = (my_rank + thread_count - 1) % thread_count; // 计算消
息的来源线程
    char *my_msg = malloc(MSG_MAX * sizeof(char));      // 分配消息内
存
```

```
    sprintf(my_msg, "Hello to %ld from %ld", dest, my_rank); // 格式化消息内容

    messages[dest] = my_msg;     // 将消息存储在目标线程的位置
    sem_post(&semaphores[dest]); // 通过信号量通知目标线程有新消息

    sem_wait(&semaphores[my_rank]);                          // 等待自己的信号量，以接收消息
    printf("Thread %ld > %s\n", my_rank, messages[my_rank]); // 打印接收到的消息
    return NULL;
}
```

b. 使用条件变量实现

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

#define MSG_MAX 100 // 定义消息最大长度

int thread_count;         // 线程数量
char **messages;          // 存储消息的字符串数组
pthread_mutex_t mutex;    // 互斥锁，用于同步访问messages 数组
```

```c
pthread_cond_t *cond_vars; // 条件变量数组，用于线程间的同步

void *Send_msg(void *rank); // 线程函数声明

int main(int argc, char *argv[])
{
    long thread;                            // 用于循环的线程索引
    pthread_t *thread_handles;              // 线程句柄数组
    thread_count = strtol(argv[1], NULL, 10); // 从命令行参数获取线程数量

    // 为线程句柄、消息数组和条件变量数组分配内存
    thread_handles     =     (pthread_t     *)malloc(thread_count     *
sizeof(pthread_t));
    messages = (char **)malloc(thread_count * sizeof(char *));
    cond_vars     =     (pthread_cond_t     *)malloc(thread_count     *
sizeof(pthread_cond_t));
    pthread_mutex_init(&mutex, NULL); // 初始化互斥锁

    for (thread = 0; thread < thread_count; thread++)
    {
        messages[thread] = NULL;                    // 初始化消息数组
        pthread_cond_init(&cond_vars[thread], NULL); // 初始化每个线程的条件
变量
    }
```

```
    for (thread = 0; thread < thread_count; thread++)
    {
        // 创建线程，传递线程编号作为参数
        pthread_create(&thread_handles[thread], NULL, Send_msg, (void
*)thread);
    }

    for (thread = 0; thread < thread_count; thread++)
    {
        // 等待所有线程完成
        pthread_join(thread_handles[thread], NULL);
    }

    // 清理资源：释放消息内存，销毁条件变量，互斥锁和线程句柄数组
    for (thread = 0; thread < thread_count; thread++)
    {
        free(messages[thread]);
        pthread_cond_destroy(&cond_vars[thread]);
    }
    free(messages);
    free(cond_vars);
    pthread_mutex_destroy(&mutex);
    free(thread_handles);
    return 0;
}
```

```
void *Send_msg(void *rank)
{
    long my_rank = (long)rank;                                  // 线程编号
    long dest = (my_rank + 1) % thread_count;                   // 计算消息的
目标线程
    long source = (my_rank + thread_count - 1) % thread_count; // 计算消
息的来源线程
    char *my_msg = (char *)malloc(MSG_MAX * sizeof(char));     // 分配消息
内存

    sprintf(my_msg, "Hello to %ld from %ld", dest, my_rank); // 格式化消息
内容

    pthread_mutex_lock(&mutex);           // 锁定互斥锁
    messages[dest] = my_msg;              // 将消息存储在目标位置
    pthread_cond_signal(&cond_vars[dest]); // 通知目标线程有消息
    pthread_mutex_unlock(&mutex);         // 解锁互斥锁

    pthread_mutex_lock(&mutex); // 再次锁定互斥锁以等待自己的消息
    while (messages[my_rank] == NULL)
    {
        // 如果当前线程的消息为空，则等待
        pthread_cond_wait(&cond_vars[my_rank], &mutex);
    }
```

```
    // 打印接收到的消息
    printf("Thread %ld > %s\n", my_rank, messages[my_rank]);
    pthread_mutex_unlock(&mutex); // 解锁互斥锁
    return NULL;
}
```

(三) 执行时间

(1) 实验第 1 题，可以参考以下测试列表进行测试，并试图与 MPI 执行效率进行比较，并说明原因。

测试场景	进程数	数据规模	测试次数
1	1（串行）	4800	10
2	2（并行）	4800	10
3	4（并行）	4800	10
4	16（并行）	4800	10
5	24（并行）	4800	10
6	32（并行）	4800	10
7	1（串行）	2000	10
8	1（串行）	4000	10
9	1（串行）	8000	10
10	1（串行）	16000	10
11	16（并行）	3200	10
12	16（并行）	6400	10
13	16（并行）	12800	10

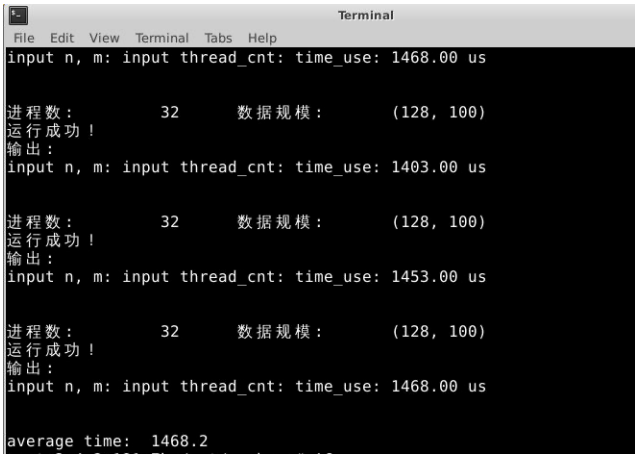
14	16（并行）	16000	10

（2）实验第 2 题，可以参考教材 P113 页表 4-1 进行比较测试。

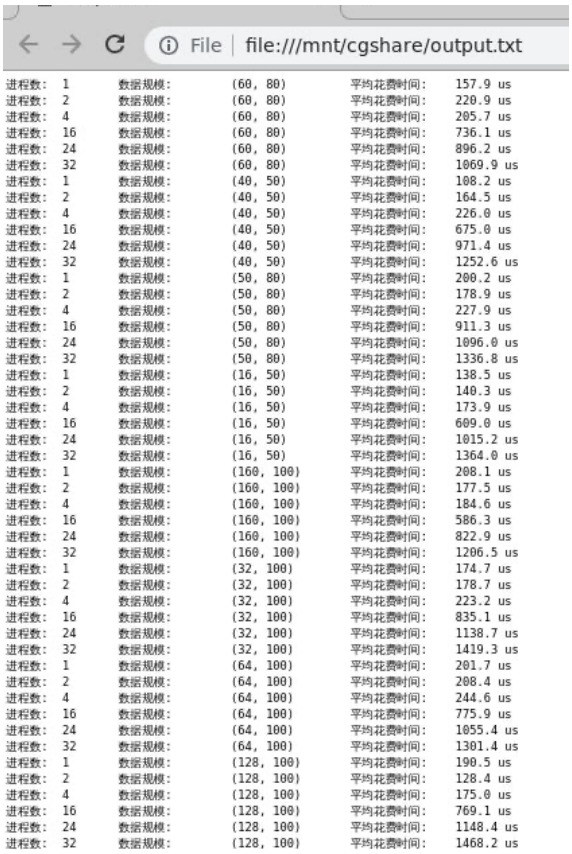
（3）实验第 3 题主要考察对信号量的使用，可以不用测试比较。

（1）执行时间截图

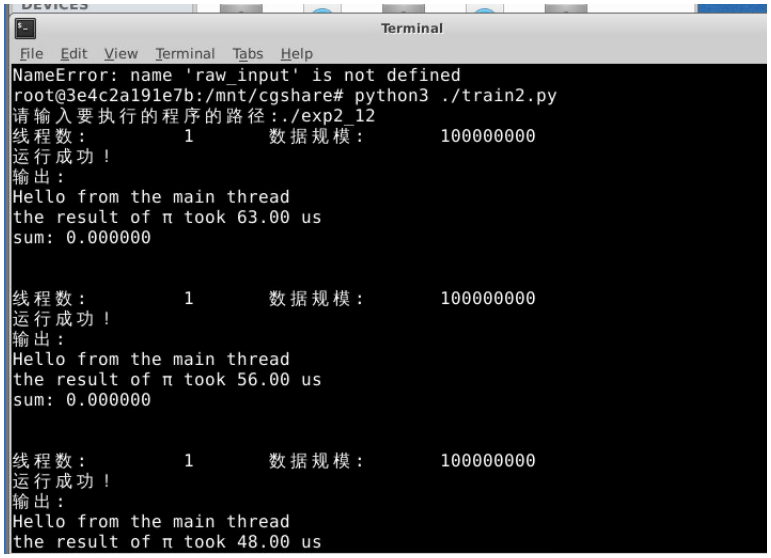
1) 矩阵-向量乘法

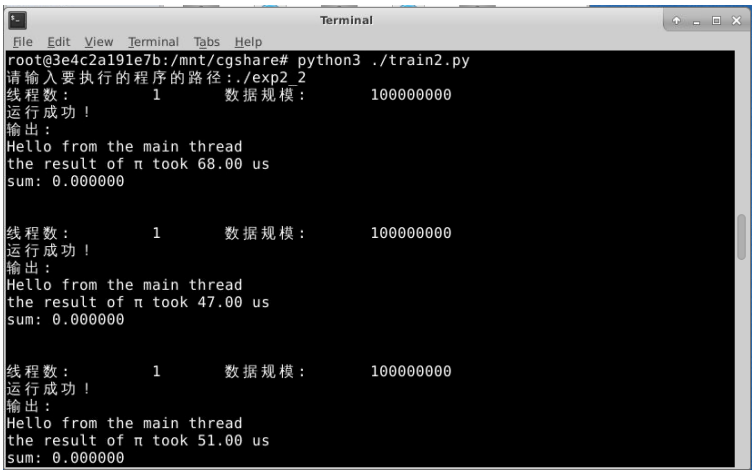


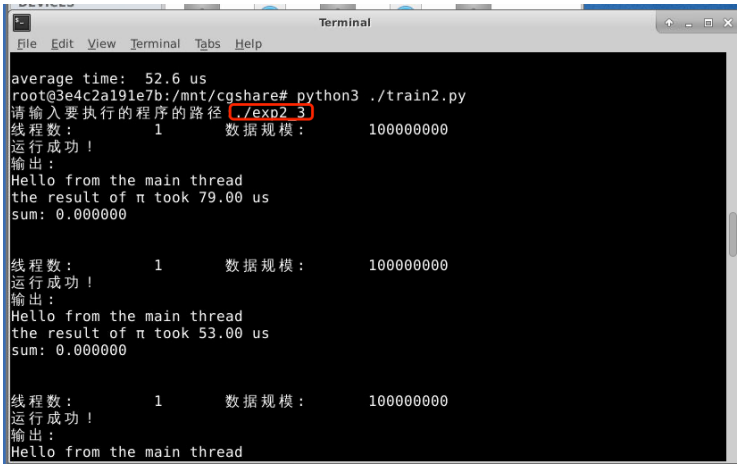
使用 python 程序循环调用后获得的运行时间结果



2) 利用忙等待、互斥量及条件变量来编写求∏值







3) 用信号量或者条件变量编写发送消息

```
root@3e4c2a191e7b:/mnt/cgshare# gcc -o exp3 ./exp3.c -lpthread
root@3e4c2a191e7b:/mnt/cgshare# gcc -o exp3_2 ./exp3_2.c -lpthread
root@3e4c2a191e7b:/mnt/cgshare# ./exp3 10
Thread 1 > Hello to 1 from 0
Thread 2 > Hello to 2 from 1
Thread 3 > Hello to 3 from 2
Thread 4 > Hello to 4 from 3
Thread 5 > Hello to 5 from 4
Thread 6 > Hello to 6 from 5
Thread 7 > Hello to 7 from 6
Thread 8 > Hello to 8 from 7
Thread 0 > Hello to 0 from 9
Thread 9 > Hello to 9 from 8
root@3e4c2a191e7b:/mnt/cgshare#
```

```
root@3e4c2a191e7b:/mnt/cgshare# gcc -o exp3_2 ./exp3_2.c -lpthread
root@3e4c2a191e7b:/mnt/cgshare# ./exp3 10
Thread 1 > Hello to 1 from 0
Thread 2 > Hello to 2 from 1
Thread 3 > Hello to 3 from 2
Thread 4 > Hello to 4 from 3
Thread 5 > Hello to 5 from 4
Thread 6 > Hello to 6 from 5
Thread 7 > Hello to 7 from 6
Thread 8 > Hello to 8 from 7
Thread 0 > Hello to 0 from 9
Thread 9 > Hello to 9 from 8
root@3e4c2a191e7b:/mnt/cgshare# ./exp3_2 10
Thread 1 > Hello to 1 from 0
Thread 2 > Hello to 2 from 1
Thread 3 > Hello to 3 from 2
Thread 4 > Hello to 4 from 3
Thread 5 > Hello to 5 from 4
Thread 6 > Hello to 6 from 5
Thread 9 > Hello to 9 from 8
Thread 0 > Hello to 0 from 9
Thread 7 > Hello to 7 from 6
Thread 8 > Hello to 8 from 7
```

（2）执行时间分析（表格）

1) 矩阵向量乘法

① 执行时间表格分析(单位 us)

	2000	4000	8000	16000
1（串行）	103.5	128.6	119.9	261.9
2（并行）	121.5	167.0	165.7	222.9
4（并行）	152.7	215.8	246.4	220.1
16（并行）	587.9	935.4	578.0	764.9
24（并行）	893.2	986.8	863.8	804.7
32（并行）	1226.0	1169.2	1281.0	1047.6

分析：

1. 串行执行时间：随着数据规模的增加，执行时间也增加。这是预期中的结果，因为处理的数据量越大，所需时间越长。
2. 并行执行时间：
    - 对于小数据规模（2000 和 4000），并行执行时间比串行时间更长。这可能是由于线程创建和管理的开销超过了并行执行的收益。
    - 对于中等数据规模（8000），并行执行时间开始接近或略长于串行执行时间。
    - 对于大数据规模（16000），并行执行时间在 2 和 4 个线程时优于串行执行，但在更多线程时（16、24、32）反而显著增加。这表明并行效率在高线程数时下降，可能因为线程间通信开销和竞争增加。

② 执行加速比分析

	2000	4000	8000	16000
1（串行）	1	1	1	1
2（并行）	0.85	0.77	0.72	1.18
4（并行）	0.68	0.60	0.49	1.19
16（并行）	0.18	0.14	0.21	0.34
24（并行）	0.12	0.13	0.14	0.33
32（并行）	0.08	0.11	0.09	0.25

分析：

1. 低线程数（2 和 4）：
    - 对于小数据规模，加速比小于 1，表明并行执行并没有比串行快。
    - 对于大数据规模，加速比略大于 1，表明并行执行稍微加速。
2. 高线程数（16、24、32）：
    - 无论数据规模大小，加速比都显著小于 1。这表明多线程带来的开销大于并行执行的收益，可能由于线程间通信和竞争增加。

③ 执行效率分析

	2000	4000	8000	16000
1（串行）	1	1	1	1
2（并行）	0.425	0.385	0.360	0.590
4（并行）	0.170	0.150	0.123	0.298
16（并行）	0.011	0.009	0.013	0.021
24（并行）	0.005	0.005	0.006	0.014
32（并行）	0.003	0.003	0.003	0.008

分析：

1. **低线程数（2 和 4）：**
    - 执行效率在小数据规模时较高，但随着数据规模增加，效率有所下降。
2. **高线程数（16、24、32）：**
    - 执行效率非常低，表明随着线程数增加，开销增加导致并行执行效果不佳。
    - 数据规模较大时（16000），执行效率略有提升，但仍远低于低线程数。

总结

1. **并行执行的优势**：在适当的线程数（例如 2 或 4 个线程）和较大的数据规模（16000）时，并行执行能提供一些加速。
2. **并行执行的劣势**：当线程数过多时，并行执行的开销（例如线程管理、通信、同步）超过了其带来的收益，导致整体效率下降。
3. **优化方向**：为了最大化并行执行的效率，应该合理选择线程数。对于不同的数据规模，可能需要不同的最优线程数。一般来说，线程数不宜过多，否则反而会降低性能。

2) 利用忙等待、互斥量及条件变量来编写求∏值

① 执行时间表格分析(us)

	1（串行）	2（并行）	4（并行）	8（并行）	16（并行）	32（并行）	64（并行)
忙等待	53.8	51.0	55.3	51.7	50.9	57.7	56.9

互斥量	57.9	53.5	49.0	49.1	49.9	51.8	48.2
条件变量	55.6	55.7	49.9	55.5	48.6	48.7	54.0

分析：

1. **忙等待**：
   - 随着线程数的增加，执行时间并没有显著减少，反而在某些情况下有所增加。例如，在 32 个线程时执行时间增加到 57.7 毫秒，而串行执行时间为 53.8 毫秒。
   - 最佳执行时间出现在 16 个线程（50.9 毫秒），但整体上并没有显著优势。
2. **互斥量**：
   - 执行时间在 4 个线程时显著减少，为 49.0 毫秒，比串行执行时间 57.9 毫秒减少了 8.9 毫秒。
   - 互斥量的并行化效果在 4 到 8 个线程时较为理想，但在线程数继续增加时效果递减，例如 32 个线程时为 51.8 毫秒。
   - 最佳执行时间出现在 64 个线程（48.2 毫秒），比串行执行时间减少了 9.7 毫秒。
3. **条件变量**：
   - 在 4 个线程时执行时间显著减少，为 49.9 毫秒，比串行执行时间 55.6 毫秒减少了 5.7 毫秒。
   - 执行时间在 16 个线程时达到最佳，为 48.6 毫秒，比串行执行时间减少了 7 毫秒。
   - 但在 64 个线程时，执行时间回升至 54.0 毫秒，接近串行执行时间。

② 执行加速比分析

线程数	**1**（串行）	**2**（并行）	**4**（并行）	**8**（并行）	**16**（并行）	**32**（并行）	**64**（并行）
忙等待	1.00	1.05	0.97	1.04	1.06	0.93	0.95

互斥量	1.00	1.08	1.18	1.18	1.16	1.12	1.20
条件变量	1.00	1.00	1.11	1.00	1.14	1.14	1.03

分析：

- **忙等待**：加速比在 2 和 8 个线程时略大于 1，表示有一些并行化的好处，但在其他线程数时，加速比小于 1 或接近 1，表明并行化并没有显著提高性能。
- **互斥量**：加速比在所有并行线程数下都大于 1，特别是在 4 个线程时达到最高 1.18，表明在适当的线程数下，互斥量并行化效果较好。
- **条件变量**：加速比在 4 个和 16 个线程时显著大于 1，表明在这些线程数下，条件变量的并行化效果较好。

③ 执行效率分析

线程数	**1（串行）**	**2（并行）**	**4（并行）**	**8（并行）**	**16（并行）**	**32（并行）**	**64（并行）**
忙等待	1.00	0.525	0.2425	0.13	0.0663	0.0291	0.0148
互斥量	1.00	0.54	0.295	0.1475	0.0725	0.035	0.0188
条件变量	1.00	0.50	0.2775	0.125	0.0713	0.0356	0.0161

分析：

- **忙等待**：效率随着线程数增加而下降，特别是在高线程数时（32 和 64），效率非常低，表明高线程数下并行化的开销抵消了其带来的好处。
- **互斥量**：效率在 4 个线程时最高，随着线程数增加逐渐下降，显示出在中等线程数时互斥量的并行化效果最佳。
- **条件变量**：效率在 4 个线程时较高，但在高线程数时迅速下降，表明高线程数下条件变量的并行化效果并不显著。

总结：

- **忙等待**：并行化对性能提升有限，甚至在高线程数时会降低效率。
- **互斥量**：适当的并行化（例如 4 至 8 个线程）可以显著提高性能，但高线程数时效果递减。

- **条件变量**：适当的并行化（例如 4 至 16 个线程）可以显著提高性能，但在过高线程数时效果减弱。

五、实验总结与扩展

（一）实验总结

任务一

任务一使我深入了解了 pthread 的基本编程实现，掌握了一些 pthread 中的常用函数。通过这次实验，我熟悉了 Pthread 编程的基本实现流程，能够自如地编写 Pthread 程序。这不仅让我能够创建和管理线程，还让我理解了线程之间的通信和同步。通过实际操作，我清楚了 Pthread 和 MPI 的区别：Pthread 适用于共享内存模型，而 MPI 适用于分布式内存模型。实验让我认识到，在适当的场景下选择合适的并行编程模型是至关重要的。

任务二

任务二让我对同步和互斥有了更深刻的理解。之前在操作系统课程中，我只是理论上学习了一些同步和互斥算法，但是没有机会实际实现和运行它们。这次实验让我有机会亲自动手实现这些算法，并通过测试验证其正确性和性能表现。这种亲身实践的机会非常宝贵，使我在理解这些概念的基础上，真正掌握了如何在实际编程中应用它们。我深刻体会到，同步和互斥不仅仅是概念上的理解，更是实实在在的编程技术。

任务二通过三种不同的方法实现了对临界区资源的访问，使我对这些方法有了全面的认识和理解。最开始使用的是最简单直接的忙等待法（busy-waiting）。这种方法虽然易于理解和实现，但通过实验我发现它会大量浪费系统资源，并且在高并发环境下性能表现很差。

任务三

任务三的目标是用信号量和条件变量编写发送消息的 Pthreads 程序。通过这个任务，我实

现了多线程之间的消息传递机制，并对信号量和条件变量的应用有了更深入的理解。

1. **信号量的实现**：
    - 我使用信号量实现了线程之间的消息传递。通过信号量的初始化、等待（sem_wait）和信号（sem_post）操作，我成功地让每个线程能够安全地发送和接收消息。信号量的使用避免了忙等待的问题，提高了线程同步的效率。
    - 实验过程中，我观察到信号量能够有效地控制线程的执行顺序，确保消息按预期发送和接收。这种机制在资源利用和线程调度上表现出色，特别适合简单的同步场景。
2. **条件变量的实现**：
    - 我使用条件变量实现了另一种线程间消息传递的方法。条件变量结合互斥锁，通过条件等待（pthread_cond_wait）和信号通知（pthread_cond_signal），实现了线程间的高效同步。
    - 条件变量的使用使得线程在等待消息时不会占用 CPU 资源，只有在接收到信号通知后才会被唤醒处理消息。这种机制不仅提高了资源利用率，还使得线程同步更加灵活和高效。
    - 在实验中，我通过条件变量的实现，深刻体会到它在复杂同步场景中的优势，尤其是在需要多个线程协调工作的情况下。

总结

总的来说，这次实验不仅让我熟练掌握了 Pthread 编程，还加深了我对同步和互斥机制的理解。通过实际编程和测试，我体会到不同同步方法的优缺点及其适用场景。实验中的每一步都让我收获颇丰，从理论到实践的转变让我对并行编程有了更加全面和深刻的认识。这次实验经历让我意识到，扎实的理论基础加上丰富的实践经验，是成为一名优秀程序员的

关键。这些宝贵的经验和收获将为我未来的学习和工作打下坚实的基础。

（二）实验中所涉及的算法的改进措施及效果

1. 使用了 peterson 算法改进普通的忙等待算法

用

```
// 使用Peterson 算法实现互斥
flag[my_rank] = 1;
turn = my_rank;
for (long j = 0; j < thread_count; j++)
{
    if (j != my_rank)
    {
        while (flag[j] && turn == my_rank)
            ; // 忙等待
    }
}
```

替换了

```
while (flag != my_rank)
    ;
sum += my_sum;
```

Peterson 算法的优势：

1. **公平性**：确保所有线程都能公平地获得进入临界区的机会，避免了 CPU 资源的浪费，系统性能更高效。
2. **资源高效**：通过合理的等待机制，减少了系统资源占用。

普通忙等待算法的劣势：

1. 只能按固定顺序执行
2. **高 CPU 占用**：持续占用 CPU 时间片，导致系统性能下降。
3. **资源效率低**：不必要的资源竞争，尤其在多线程环境下问题更严重。

2. 使用了自己编写的 python 程序

可以一次性测试完所有情况，并且将结果保存在文件中。

```
#coding=UTF-8
import subprocess


def run_exp1_with_input(executable_path, n, m, thread_num, output_file):
    # 构建mpirun 命令
    command = [executable_path]


    # 将输入转换为字符串，以便通过stdin 传递
    input_str = str(n) + " " + str(m) + "\n" + str(thread_num) + "\n"


    # 使用subprocess 运行命令，并提供输入
    result = subprocess.run(command, input=input_str,
stdout=subprocess.PIPE, stderr=subprocess.PIPE, universal_newlines=True)


    # 打开输出文件
    with open(output_file, "a") as file:
```

```
        # 检查命令是否成功执行
        if result.returncode == 0:
            print("运行成功! ")
            print("输出:\n" + result.stdout + "\n")
        else:
            # file.write("运行失败! \n")
            # file.write("错误信息:\n" + result.stderr + "\n")
            print("运行失败! ")
            print("错误信息:\n" + result.stderr + "\n")
    return str(result.stdout).split("time_use: ")[1].split(" ")[0]


data_size = [4800,2000,4000,8000,16000,3200,6400,12800] #测试的数据规模
matrix_size =
[(60,80),(40,50),(50,80),(16,50),(160,100),(32,100),(64,100),(128,100)
] #测试的矩阵规模
process_num = [1,2,4,16,24,32] #测试的进程数
times = 10 #每个数据规模下的测试次数
#测试不同的数据规模
for size in matrix_size:
    for num in process_num:
        t = 0;
        for i in range(times):
            str_print = "进程数: \t" + str(num) + "\t 数据规模: \t" + str(size)
            print(str_print)
            t += float(run_exp1_with_input("./exp1", size[0], size[1], num,
```

```
"output.txt"))
        t /= times
        with open("output.txt", "a") as file:
            file.write(str_print + "\t 平均花费时间: \t" + str(t) + " us\n")
        print("average time: ", t)
```

六、思考题

（一）什么是程序，什么是进程？什么是线程？请重点阐述进程与线程的区别。

程序、进程与线程的概念

程序 是计算机可执行文件的一种形式，它由一组指令和数据的有序集合组成，描述了计算机在执行时应该按照什么样的顺序执行哪些操作。程序本身是静态的，它不占用系统的任何资源，只是存储在磁盘上的一个文件。

进程 是操作系统中的一个概念，是指一个正在执行中的程序的实例。一个进程可以看做是一个执行中的程序及其相关资源的总称，包括程序代码、数据、打开的文件、系统资源（如内存、CPU 时间片等），以及进程自身的状态信息（如优先级、进程 ID 等）。每个进程都有独立的地址空间，进程之间无法直接访问彼此的内存和资源。一个进程的生命周期包括创建、就绪、运行、等待（阻塞）和终止五个状态。

线程 是操作系统中的另一个概念，是指进程中的一个执行单元，是操作系统分配处理器时间片的基本单位。线程是比进程更轻量级的执行单元，一个进程可以包含一个或多个线程，这些线程可以共享同一个进程的资源，如代码段、数据段、打开的文件、以及进程的状态信息等。线程之间的切换比进程之间的切换要更加高效，因为线程之间共享进程的资源，不需要完全独立的环境。

进程与线程的主要区别

1. **独立性与共享性**:
    - **进程** 之间相互独立，每个进程都有自己独立的地址空间，不同进程之间的内存和资源不能直接访问。进程的独立性意味着一个进程的崩溃不会影响到其他进程。
    - **线程** 是进程中的一个执行单元，多个线程可以共享同一个进程的资源（如代码段、数据段、打开的文件等）。线程共享进程的地址空间，这使得线程之间可以更加高效地通信和数据共享。
2. **内存与资源**:
    - **进程** 拥有自己独立的地址空间，进程之间的内存和资源不能直接访问。进程之间的通信需要通过操作系统提供的机制（如管道、信号量、共享内存等），这些机制通常比较复杂且开销较大。
    - **线程** 共享进程的地址空间和资源，线程之间可以直接通过共享内存等机制进

行通信，这种通信方式更加高效且简单。

3. **切换开销**:
    - **进程** 之间的切换需要保存和恢复整个进程的状态信息，包括所有的寄存器状态、内存映射表等，因此开销比较大。这种切换被称为“上下文切换”。
    - **线程** 之间的切换只需要保存和恢复部分状态信息（如程序计数器、部分寄存器状态），开销比较小。因此，在多线程的程序中，线程的切换会更加频繁，但总体效率更高。
4. **通信机制**:
    - **进程** 之间的通信需要借助操作系统提供的进程间通信（IPC）机制，如管道、消息队列、共享内存和信号量等。这些机制虽然功能强大，但也带来了额外的复杂性和系统开销。
    - **线程** 之间可以通过共享进程的全局变量、数据段等方式进行直接通信，无需额外的操作系统支持，效率更高。

举例说明

程序 是你在计算机上编写并保存的一个文件，例如一个 C 语言源文件 hello.c。当你使用编译器将这个源文件编译成可执行文件 hello.exe 时，这个可执行文件依然是静态的，不占用任何系统资源。

当你运行 hello.exe 时，操作系统会为其创建一个 **进程**。这个进程包括代码段、数据段、程序计数器、堆栈、打开的文件描述符等。当这个进程开始执行时，它进入运行状态，并占用 CPU 时间片和内存资源。如果你在 hello.c 中创建了多个 **线程**，那么这些线程会共享这个进程的地址空间和资源，分别执行不同的任务，但所有的线程都属于同一个进程，并共享进程的全局变量和数据。

总结:

- **程序** 是静态的指令和数据的集合。
- **进程** 是正在执行中的程序实例，具有独立的地址空间。
- **线程** 是进程中的执行单元，多个线程共享同一个进程的资源。

进程与线程的主要区别在于资源的独立性和共享性、切换开销和通信机制。进程之间相互独立，拥有独立的地址空间，切换开销大，通信复杂；线程之间共享进程的资源，切换开销小，通信高效。

（二）什么是并行，什么是并发？多线程系统中如何实现并发？试举例说明并发的应用场景。

并行 是指多个任务在同一时刻同时执行，每个任务都有自己的处理器或处理器核心。并行处理需要硬件支持，例如多个 CPU 或者 CPU 中的多个核心。并行处理的目的是通过同时执行多个任务来提高系统的处理能力和效率。例如，在一个拥有四个核心的 CPU 上，四个任务可以同时运行，每个核心独立执行一个任务，这样可以大大减少总的执行时间。

并发 是指多个任务在同一时间段内交替执行，看起来像是同时执行。并发通常通过时间片轮转的方式实现，即任务依次占用 CPU 并执行一段时间，然后切换到下一个任务。这种方式可以在单个 CPU 或者单个核心上实现。并发处理的目的是通过任务交替执行来提高系统的响应能力和资源利用率。例如，在一个单核心的 CPU 上，操作系统可以通过快速切换不同的任务，使得用户感觉多个任务在“同时”运行。

多线程系统中实现并发

在多线程系统中，实现并发的关键是使用线程。线程是进程中的执行单元，多个线程可以共享同一个进程的资源（如内存、文件句柄等），但是每个线程有自己的调用栈和执行路径。操作系统通过线程调度算法，让多个线程交替执行，从而实现并发。

线程调度算法

1. **时间片轮转**：每个线程分配一个时间片，线程在其时间片内执行任务，时间片结束后切换到下一个线程。这样可以确保每个线程都有机会运行，从而实现公平调度。
2. **优先级调度**：每个线程分配一个优先级，优先级高的线程优先执行，优先级相同的线程按时间片轮转执行。这种调度方式可以确保重要任务优先执行。
3. **多级反馈队列**：综合时间片轮转和优先级调度，根据线程的执行时间和优先级动态调整线程的调度策略。这种调度算法可以根据线程的行为调整其优先级，从而提高系统的整体性能。

通过这些调度算法，操作系统能够在单个 CPU 或者单个核心上实现多个线程的并发执行。例如，在一个多线程的 Web 服务器中，不同的线程可以同时处理不同的客户端请求，虽然这些线程是在单个 CPU 上交替执行的，但用户感觉是同时处理的。

并发的应用场景

并发在许多实际应用中起到了关键作用，以下是一些典型的并发应用场景：

1. **数据库系统**：
   - 数据库是一个典型的并发应用程序。多个用户可以同时读取和写入数据库，而不会互相干扰。例如，一个用户在查询数据的同时，另一个用户可以更新数据。数据库管理系统通过锁机制和事务管理来实现并发控制，确保数据的一致性和完整性。
   - 具体例子：在一个银行系统中，多个用户可以同时进行转账操作，数据库管理系统通过锁机制确保每次转账操作的原子性和一致性。
2. **网络服务**：
   - 网络服务（例如 Web 服务器）需要同时处理多个客户端的请求。使用并发可以提高服务器的响应时间和吞吐量。例如，一个 Web 服务器可以同时处理多个 HTTP 请求，通过多线程或异步 I/O 的方式来实现并发，从而提供更快的响应和更高的服务质量。
   - 具体例子：在一个电子商务网站中，多个用户可以同时浏览商品、添加购物车和下订单，服务器通过并发处理这些请求，提高了用户体验。
3. **游戏开发**：
   - 现代游戏需要同时处理大量的用户输入和复杂的物理计算。使用并发可以提高游戏的流畅性和响应时间。例如，一个游戏可以使用多个线程分别处理用户输入、物理引擎和图形渲染，使游戏运行更加平滑和实时。
   - 具体例子：在一个多人在线游戏中，不同的线程可以分别处理玩家的输入、游戏的物理模拟和网络通信，确保游戏的流畅运行和实时响应。
4. **人工智能**：
   - 许多机器学习和人工智能应用程序需要同时处理大量的数据和计算。使用并发可以加快训练和推理过程。例如，一个深度学习模型的训练过程可以并行化，将训练数据划分为多个批次，分别在多个 GPU 上进行计算，从而加快训练速度。
   - 具体例子：在一个图像识别系统中，多个线程可以同时处理不同的图像数据，加快了图像识别的速度和准确性。
5. **大数据处理**：

- 处理大量数据的任务可以被拆分成多个独立的子任务，并在多个处理器上同时运行以提高处理速度。例如，Hadoop 和 Spark 等大数据处理框架使用并行计算模型，将大规模数据集划分为多个数据块，在多个节点上并行处理，从而实现高效的大数据分析。
- 具体例子：在一个数据分析项目中，数据集被分成多个部分，不同的线程或节点同时处理这些数据部分，从而加快了数据处理和分析的速度。

（三）pthread 可以通过哪些方式对临界区进行访问，分别采用什么方式实现，试写出具体实现的过程函数?

pthread 对临界区的访问方式

在多线程编程中，临界区是指多个线程可能同时访问的共享资源。为了保证数据的一致性和完整性，必须采取适当的同步机制来控制对临界区的访问。pthread 库提供了多种方式来实现对临界区的访问控制，包括互斥锁、条件变量、读写锁和信号量。

1. 互斥锁（mutex）

互斥锁是一种最基本的锁机制，用于保护共享资源，确保同一时间只有一个线程可以访问临界区。互斥锁通过阻塞的方式实现对共享资源的独占访问，避免了数据的不一致性。

互斥锁的实现方法:

- pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *attr): 初始化互斥锁。
- pthread_mutex_lock(pthread_mutex_t *mutex): 获取互斥锁。如果锁已经被其他线程占用，该调用将阻塞，直到锁可用。
- pthread_mutex_trylock(pthread_mutex_t *mutex): 尝试获取互斥锁。如果锁已经被其他线程占用，该调用不会阻塞，返回一个错误代码。

- pthread_mutex_unlock(pthread_mutex_t *mutex): 释放互斥锁，使其他等待该锁的线程可以获取锁。
- pthread_mutex_destroy(pthread_mutex_t *mutex): 销毁互斥锁，释放其占用的资源。

示例代码:

pthread_mutex_lock(&mutex); // 临界区开始

printf("Thread %d is in the critical section.\n", *(int *)arg); // 临界区结束 pthread_mutex_unlock(&mutex); return NULL;

2. 条件变量（condition variable）

条件变量是用于线程之间通信的一种机制，它允许一个线程等待另一个线程发出的信号，从而避免了忙等待。条件变量通常与互斥锁一起使用，以确保在访问共享资源之前获取互斥锁。

条件变量的实现方法:

- pthread_cond_init(pthread_cond_t *cond, const pthread_condattr_t *attr): 初始化条件变量。
- pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex): 等待条件变量。当条件变量满足时，被唤醒的线程会重新获取互斥锁。
- pthread_cond_signal(pthread_cond_t *cond): 唤醒等待该条件变量的一个线程。
- pthread_cond_broadcast(pthread_cond_t *cond): 唤醒等待该条件变量的所有线程。
- pthread_cond_destroy(pthread_cond_t *cond): 销毁条件变量，释放其占用的资源。

示例代码:

```
pthread_mutex_lock(&mutex);

while (!ready) { pthread_cond_wait(&cond, &mutex); }

printf("Thread %d received signal.\n", *(int *)arg);

pthread_mutex_unlock(&mutex);
```

3. 读写锁（read-write lock）

读写锁是一种特殊的锁，它可以同时允许多个线程进行读操作，但只允许一个线程进行写操作。当有线程需要写共享资源时，会阻塞其他所有线程的读写操作。当写操作完成后，所有被阻塞的读写操作都可以继续执行。

读写锁的实现方法:

- pthread_rwlock_init(pthread_rwlock_t *rwlock, const pthread_rwlockattr_t *attr): 初始化读写锁。
- pthread_rwlock_rdlock(pthread_rwlock_t *rwlock): 获取读锁。多个线程可以同时获取读锁。
- pthread_rwlock_wrlock(pthread_rwlock_t *rwlock): 获取写锁。只有一个线程可以获取写锁，且写锁会阻塞其他读写操作。
- pthread_rwlock_unlock(pthread_rwlock_t *rwlock): 释放读写锁。
- pthread_rwlock_destroy(pthread_rwlock_t *rwlock): 销毁读写锁，释放其占用的资源。

示例代码:

```
pthread_rwlock_rdlock(&rwlock);

printf("Reader thread %d is reading.\n", *(int *)arg);

pthread_rwlock_unlock(&rwlock);
```

4. 信号量（semaphore）

信号量用于控制多个线程对共享资源的访问。信号量的值表示可用资源的数量，线程可以通过信号量来同步和互斥地访问资源。

信号量的实现方法:

- sem_init(sem_t *sem, int pshared, unsigned int value): 初始化信号量。value 表示信号量的初始值，pshared 为 0 表示信号量在进程内共享，为非 0 表示在进程间共享。
- **sem_wait(sem_t *sem)**: 等待信号量。如果信号量的值大于 0,则将其减 1 并立即返回。如果信号量的值为 0，则阻塞直到信号量的值大于 0。
- **sem_post(sem_t *sem)**: 释放信号量，将信号量的值加 1。如果有线程正在等待信号量，则唤醒其中一个线程。
- sem_destroy(sem_t *sem): 销毁信号量，释放其占用的资源。

示例代码:

```
sem_wait(&sem);

printf("Thread %d is in the critical section.\n", *(int *)arg);

sem_post(&sem);
```

（四）互斥锁、信号量、条件变量三种方式实现的 pthread 基本函数，并说明它们在线程

同步作用中实现的区别。

在多线程编程中，线程同步是一个重要的概念，用于确保多个线程在访问共享资源时不会发生冲突。以下是三种常见的线程同步机制：互斥锁、信号量和条件变量。

1. **互斥锁（Mutex）**

互斥锁（mutex）是最常用的同步机制之一，用于保护共享资源的访问。互斥锁通过线程获取锁的方式来实现对临界区的保护。线程在访问共享资源之前必须先获取锁，如果锁已经被其他线程获取，则当前线程会被阻塞，直到锁被释放。

pthread 中的互斥锁基本函数：

- `pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *attr)`：初始化互斥锁。`mutex` 是要初始化的互斥锁，`attr` 是互斥锁的属性，可以为 NULL，表示使用默认属性。
- `pthread_mutex_lock(pthread_mutex_t *mutex)`：获取互斥锁。如果锁已经被其他线程获取，则当前线程会被阻塞，直到锁被释放。
- `pthread_mutex_trylock(pthread_mutex_t *mutex)`：尝试获取互斥锁。如果锁已经被其他线程获取，函数会立即返回而不会阻塞。
- `pthread_mutex_unlock(pthread_mutex_t *mutex)`：释放互斥锁，使得其他线程可以获取锁。
- `pthread_mutex_destroy(pthread_mutex_t *mutex)`：销毁互斥锁。

互斥锁的作用：

- 保护临界区：确保同一时刻只有一个线程能够进入临界区，防止数据竞争和不一致性。
- 阻塞线程：当一个线程获取锁时，其他需要访问该锁的线程将会被阻塞，直到锁被释放。

互斥锁的优缺点：

- 优点：实现简单，开销较小，适用于需要互斥访问共享资源的场景。
- 缺点：会导致线程阻塞，可能影响程序的性能；使用不当可能导致死锁。

2. **信号量（Semaphore）**

信号量（semaphore）是一种更加通用的同步机制，用于控制访问某个共享资源的线程数量。信号量是一个计数器，用于表示某个共享资源的可用数量。线程在访问共享资源之前必须先获取信号量，如果信号量的值为 0，则当前线程会被阻塞，直到其他线程释放信号量。

pthread 中的信号量基本函数：

- `sem_init(sem_t *sem, int pshared, unsigned int value)`：初始化信号量。`sem` 是要初始化的信号量，`pshared` 指定信号量是否在进程间共享（0 表示线程间共享），`value` 是信号量的初始值。
- `sem_wait(sem_t *sem)`：获取信号量。如果信号量的值为 0，则当前线程会被阻塞，直到信号量的值大于 0。
- `sem_trywait(sem_t *sem)`：尝试获取信号量。如果信号量的值为 0，函数会立即返回而不会阻塞。
- `sem_post(sem_t *sem)`：释放信号量，使得其他线程可以获取信号量。
- `sem_destroy(sem_t *sem)`：销毁信号量。

信号量的作用：

- 控制资源访问：通过信号量的值控制同时访问某个共享资源的线程数量。
- 实现同步：线程可以通过信号量实现复杂的同步操作，如生产者-消费者模型。

信号量的优缺点：

- 优点：灵活性高，可以实现多种同步模式；可以控制同时访问资源的线程数量。
- 缺点：使用不当可能导致死锁；信号量值的管理复杂，容易出错。

3. 条件变量（Condition Variable）

条件变量（condition variable）是一种高级的同步机制，用于在线程之间传递和等待条件。条件变量通常与互斥锁一起使用，以保护共享资源的访问。线程在等待条件变量时会释放互斥锁，这样其他线程就可以获取锁并访问共享资源。当特定条件满足时，线程会重新获取互斥锁并继续执行。

pthread 中的条件变量基本函数：

- pthread_cond_init(pthread_cond_t *cond, const pthread_condattr_t *attr)：初始化条件变量。cond 是要初始化的条件变量，attr 是条件变量的属性，可以为 NULL，表示使用默认属性。
- pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex)：等待条件变量。当条件不满足时，线程会释放互斥锁并等待条件变量的变化。
- pthread_cond_signal(pthread_cond_t *cond)：发送信号给一个等待条件变量的线程，使其重新获取互斥锁并继续执行。
- pthread_cond_broadcast(pthread_cond_t *cond)：发送信号给所有等待条件变量的线程。
- pthread_cond_destroy(pthread_cond_t *cond)：销毁条件变量。

条件变量的作用：

- 等待条件：线程可以等待某个特定条件的变化，当条件满足时被唤醒。
- 协作机制：与互斥锁结合使用，可以实现复杂的线程同步和协作。

条件变量的优缺点：

- 优点：灵活性高，可以实现复杂的同步和协作机制；线程在等待条件时可以释放互斥锁，避免死锁。
- 缺点：使用复杂，需要正确管理互斥锁和条件变量的配合。

三种同步机制的区别

- **互斥锁**提供了互斥访问共享资源的能力，只允许一个线程进入临界区，其他线程需要等待解锁才能进入，适用于简单的互斥访问场景。
- **信号量**用于控制同时访问某个资源的线程数量，通过适当增加和减少信号量的值，实现线程的同步和互斥操作，适用于需要控制并发线程数量的场景。
- **条件变量**用于在线程之间传递和等待条件，线程在条件变量上等待时会阻塞，直到其他线程发送信号通知条件满足时才被唤醒，适用于需要复杂条件同步和线程协作的场景。

这些同步机制各有优缺点，适用于不同的应用场景，开发者可以根据具体需求选择合适的同步机制来实现线程的安全同步和高效协作。

（五）举实际操作中（如数据库、共享文件、操作系统）中出现了关于临界区操作的例子，并说明针对这种情况适合使用互斥锁、信号量、条件变量的哪种方式实现。

1. 数据库访问

场景：

多个线程同时访问和修改数据库中的记录。

问题：

多个线程同时修改同一条记录时，可能会导致数据不一致和竞争条件，造成数据库数据的紊乱和系统的不稳定性。

适用的同步机制：

- **互斥锁**：
  - **分析**：互斥锁适用于数据库的简单读写操作，如插入、更新或删除记录。通过互斥锁，可以确保在一个线程执行这些操作时，其他线程无法访问相同的资源，从而防止数据竞争和不一致。互斥锁可以确保同一时刻只有一个线程能够进入临界区，保证数据库操作的原子性和数据的一致性。
  - **适用场景**：单一记录的读写操作、小型数据库应用、事务操作较为简单的场景。
- **条件变量**：
  - **分析**：条件变量适用于需要等待特定条件满足后才能进行数据库操作的场景。与互斥锁结合使用，条件变量可以实现复杂的同步机制，如等待某条记录的状态变为可用。当条件不满足时，线程可以释放互斥锁并进入等待状态，直到条件满足时被唤醒继续操作。这种机制可以有效地管理线程等待和资源分配，提高系统的并发性能。
  - **适用场景**：需要等待特定条件的复杂查询和更新操作、事务处理较为

复杂的大型数据库应用。

2. 共享文件操作

场景：

多个线程同时读取和写入共享文件。

问题：

如果多个线程同时写入文件，可能会导致文件损坏；同时读取和写入可能导致数据不一致，造成文件内容的紊乱和数据丢失。

适用的同步机制：

- **互斥锁**：
  - **分析**：互斥锁适用于对共享文件的简单读写操作。通过互斥锁，可以确保同一时刻只有一个线程能够执行文件写入操作，防止多个线程同时写入导致文件损坏。同样，文件读取操作也可以通过互斥锁进行保护，确保读写操作的互斥性，避免数据不一致。
  - **适用场景**：文件读写操作频繁但复杂度较低的场景、小型文件共享系统。
- **信号量**：
  - **分析**：信号量适用于控制同时访问共享文件的线程数量。通过信号量，可以限制同时读取文件的线程数目，从而防止过多的线程竞争导致的性能下降和系统资源耗尽。信号量适合用于读多写少的场景，通过设置适当的信号量值，可以有效地管理并发读操作，同时确保写操作的独占性。

- **适用场景**：高并发文件读取操作、大型文件服务器、需要控制访问并发量的文件系统。

3. 操作系统资源管理

场景：

多个线程同时申请和释放操作系统资源（如内存、设备等）。

问题：

竞争访问操作系统资源可能导致资源争用和死锁，造成系统性能下降和稳定性问题。

适用的同步机制：

- **互斥锁**：
  - **分析**：互斥锁适用于简单的资源申请和释放操作。通过互斥锁，可以确保线程在申请和释放资源时的原子性，防止多个线程同时操作同一资源导致的资源争用和数据不一致。互斥锁的使用可以有效地保护临界区，确保资源分配的安全性。
  - **适用场景**：简单的资源管理、单一资源的申请和释放操作、小型系统资源管理。
- **信号量**：
  - **分析**：信号量适用于控制资源的可用数量，适合用于资源数量有限的场景。通过信号量，可以限制同时使用某种资源的线程数量，避免资源争用导致的性能问题。信号量可以动态调整资源的可用数量，适用于资源需求波动较大的场景。

- **适用场景**：资源数量有限的操作系统资源管理、大型系统中的资源池管理、需要动态调整资源分配的场景。

- **条件变量**：

  - **分析**：条件变量适用于需要等待资源状态变化的情况。例如，当某个资源不可用时，线程可以等待条件变量，直到资源可用时被唤醒继续执行。条件变量与互斥锁结合使用，可以实现复杂的资源管理和线程协作，避免资源争用和死锁问题。

  - **适用场景**：复杂的资源管理系统、需要等待特定资源状态的操作、需要实现线程间协作的大型系统。

结论

在实际操作中，选择合适的同步机制取决于具体的应用场景和需求：

- **互斥锁**适用于简单的互斥访问场景，能够有效地保护临界区，防止数据竞争和不一致。

- **信号量**适用于控制并发线程数量和资源数量有限的场景，可以通过信号量值动态管理资源的分配和使用。

- **条件变量**适用于需要等待特定条件和实现线程协作的复杂同步场景，与互斥锁结合使用可以实现高效的线程间通信和资源管理