并行复习4 Pthread编程

> **并行程序设计的复杂性**
>
> **Pthread一些基础API、同步相关概念、忙等待\互斥量\信号量\障碍、了解条件变量\读写锁、负载均衡\任务划分**
>
> 同步锁、读写锁、barrier
>
> 忙等待的版本和互斥量的版本各自有怎样的异同
>
> 条件变量不考

基础API

启动线程和结束线程

```
pthread_t 是一种Pthread编程中的数据类型
// 启动线程
int pthread_create(
    pthread_t *thread_id,                // 输出参数，指向线程ID或句柄（可用于控制停止线程等）
    const pthread_attr_t *thread_attribute, // 输入参数，用于设置新线程的属性,NULL表示默认属性值
    void *(*thread_fun)(void *),         // 输入参数，新线程执行的函数
    void *fun_arg                         // 输入参数，传递给新线程执行函数的参数
);
pthread_create 若成功，返回0；若出错，返回非0出错编号
// 结束线程
int pthread_join(
    pthread_t pthread,  //要等待结束的线程标识符
    void** value_ptr    //存储线程返回值的指针,无返回值则通常是NULL
);
pthread_join若成功，返回0；若出错，返回非0出错编号
        1. 主线程借助操作系统创建一个新线程
        2. 线程执行一个特定函数thread_fun
        3. 所有创建的线程执行相同的函数，表示线程的计算任务分解
        4. 对于程序中不同线程执行不同任务的情况，可以用创建线程时传递的参数区分线程的id，以及其他线程的独特特性

int main()
{
   pthread_t threads[16];
```

```
    int tn;
    for(tn=0; tn < 16; tn++)
    {
            pthread_create(&threads[tn], NULL, ParFun, NULL);  //执行ParFun函数
    }
    for(tn=0; tn < 16; tn++)
    {
            pthread_join(threads[tn], NULL); //等待所有线程执行完毕
    }
    return 0;
}
```

其他

```
//终止线程
void pthread_exit(void *value_ptr);
//请求终止指定线程
int pthread_cancel(pthread_t thread);
```

Hello World程序

```

```

相关概念

并行基本概念

- 原子性：一组操作要么全部执行，要么全部不执行。即，不会得到部分执行的结果。
- 互斥：任何时刻都只有一个线程在执行。
- 临界区：是一个更新共享资源的**代码段**，一次只能允许一个线程执行该代码段。

竞争条件

☒ 执行结果依赖于两个或更多事件的时序，则存在竞争条件（race condition）

☒ **多个进程/线程尝试更新同一个共享资源时，结果可能是无法预测的，则存在竞争条件。**

☒ 更一般地，当多个进程/线程都要访问共享变量或共享文件等共享资源时，如果至少其中一个访问是更新操作，那么这些访问就可能导致某种错误，称之存在竞争条件。

数据依赖与同步

- 数据依赖：两个内存操作的序, 为了保证结果的正确性, 必须保持这个序
- 同步：在时间上, 使所有进程/线程强制在某一点必须相互等待, 确保进程/线程的正常顺序和对共享可写数据的正确访问

同步方法

忙等待

```
1  void *pi_busywaiting(void *parm) {
2      // 从参数中提取线程信息
3      threadParm_t *p = (threadParm_t *) parm;
4      int r = p->threadId;
5      int n = p->n;
6      int my_n = n / THREAD_NUM;
7      int my_first = my_n * r;
8      int my_last = my_first + my_n;
9
10     // 初始化本地和及符号因子
11     double my_sum = 0.0, factor;
12     if (my_first % 2 == 0)
13         factor = 1.0;
14     else
15         factor = -1.0;
16
17     // 计算分配给线程的迭代范围内的部分和
18     for (int i = my_first; i < my_last; i++, factor = -factor) {
19         my_sum += factor / (2 * i + 1);
20     }
21
22     // 忙等待同步，等待全局标志
23     while (flag != r)
24         Sleep(0);
25
26     // 累加本地和到全局和，增加全局标志
27     sum += my_sum;
28     flag++;
29
30     // 线程退出
31     pthread_exit(NULL);
32 }
33
```

互斥量（锁）

- 创建互斥锁

```
//静态初始化
#include <pthread.h>

pthread_mutex_t amutex = PTHREAD_MUTEX_INITIALIZER;
//动态初始化
pthread_mutex_t amutex;
pthread_mutex_init(&amutex, NULL);
```

- 使用互斥锁

```
//加锁
int pthread_mutex_lock(&amutex);
//尝试加锁
int pthread_mutex_trylock(&amutex);
两者区别：
第一种如果互斥锁已被其他线程锁定，那么该线程会被阻塞，直到获取到互斥锁为止；
第二种如果互斥锁已被其他线程锁定，不会阻塞线程，而是返回一个非零值表示未成功获取锁。
```

```
//解锁
int pthread_mutex_unlock(&amutex);
释放当前持有锁的线程，允许其他线程获取该锁。
//销毁互斥锁
int pthread_mutex_destroy(&amutex);
调用此函数后，互斥锁变为未初始化状态，并且不能再被使用。
需要谨慎确保在互斥锁不再需要时才调用该函数，以免造成潜在的错误或内存泄漏。
```

```
void *pi_mutex(void *parm) {
    // 从参数中提取线程信息
    threadParm_t *p = (threadParm_t *) parm;
    int r = p->threadId;
    int n = p->n;
    int my_n = n / THREAD_NUM;
    int my_first = my_n * r;
    int my_last = my_first + my_n;
```

```
9
10      // 初始化本地和及符号因子
11      double my_sum = 0.0, factor;
12      if (my_first % 2 == 0)
13          factor = 1.0;
14      else
15          factor = -1.0;
16
17      // 计算分配给线程的迭代范围内的部分和
18      for (int i = my_first; i < my_last; i++, factor = -factor) {
19          my_sum += factor / (2 * i + 1);
20      }
21
22      // 使用互斥锁确保对共享变量的安全访问
23      pthread_mutex_lock(&mutex);
24      sum += my_sum;
25      pthread_mutex_unlock(&mutex);
26
27      // 线程退出
28      pthread_exit(NULL);
29  }
30
```

> ❗ 忙等待与互斥量的区别：
>
> - **互斥量是阻塞等待, 没有占用cpu资源. 而忙等待是依然占用cpu资源的.**
> - 忙等待必须指定顺序, 依次执行临界区. 互斥量, 则是操作系统自行决定顺序.

死锁

信号量

- 初始化信号量

```
1  #include <semaphore.h>
```

```
int sem_init(sem_t *sem, //要初始化的信号量
             int pshared,  //通常置为0表示信号量在进程内的线程之间共享，非0表示在进程之间共享。
             unsigned value //信号量的初始值
);

```

- 使用信号量

```
//sem：要等待的信号量
//信号量值减1，如果信号量的值已经为0，则该函数会阻塞当前线程，直到信号量的值大于0。
int sem_wait(sem_t *sem);
//信号量的值加1。如果原来的值为0，那么该函数可能唤醒等待该信号量的线程之一
int sem_post(sem_t *sem);

```

- 释放信号量

```
//sem：要销毁的信号量
//销毁信号量。使用完信号量后，应该调用 sem_destroy 来释放资源。
int sem_destroy(sem_t *sem);
```

障碍barrier

一种同步机制，确保多个线程在执行过程中达到某个点时等待彼此，并同时开始执行下一阶段的任务。

线程数3，表示有3个线程到达路障时，就不用再等待。

- 初始化barrier的方法如下所示（本例中线程数为3）：

```
pthread_barrier_t b;
pthread_barrier_init(&b,NULL,3);
```

- 第二个参数指出对象属性，NULL表示默认属性
- 为等待barrier，线程应执行

```
pthread_barrier_wait(&b);
```

- 可通过下面的宏，指定一个初始值来初始化barrier

```
PTHREAD_BARRIER_INITIALIZER(3).
```

负载均衡/任务划分

- 负载不均衡

如果数据在块中的分布不均匀，某些线程可能会处理更多或更少的数据，导致负载不平衡。这可能由于数据的分布不均匀或者每个块的大小不一致引起。在这种情况下，一些线程可能完成任务比其他线程更早，而另一些线程仍在忙于处理更多的数据，降低了整体性能。

- 任务划分

1. 动态任务划分：每个线程一开始只分配一部分，之后哪个线程先完成，哪个线程继续取任务。
2. 块划分

将整个数据集划分为若干个块，通常是相等大小的块。

将每个块分配给一个处理单元，每个处理单元负责处理一个块。

负载不均衡可能是块划分会出现的问题。

3. 循环划分

任务数＞线程数，循环分配给线程。

每个线程负责的区域散布在整个矩阵中，**负载不均大大缓解**。

小结