作业 4

毕定钧 2021K8009906014

本次作业包含:

4.1

pthread 函数库可以用来在 *Linux* 上创建线程，请调研了解 *pthread_create*，*pthread_join*，*pthread_exit* 等 *API* 的使用方法，然后完成以下任务:

(1) 写一个 *C* 程序，首先创建一个值为 1 到 100 万的整数数组，然后对这 100 万个数求和。请打印最终结果，统计求和操作的耗时并打印。(注:可以使用作业 1 中用到的 *gettimeofday* 和 *clock_gettime* 函数测量耗时)

(2) 在 (1) 所写程序基础上，在创建完 1 到 100 万的整数数组后，使用 *pthread* 函数库创建 N 个线程 (N 可以自行决定，且 $N > 1$)，由这 N 个线程完成 100 万个数的求和，并打印最终结果。请统计 N 个线程完成求和所消耗的总时间并打印。和 (1) 的耗费时间相比，你能否解释 (2) 的耗时结果？(注意：可以多运行几次看测量结果)

(3) 在 (2) 所写程序基础上，增加绑核操作，将所创建线程和某个 *CPU* 核绑定后运行，并打印最终结果，以及统计 N 个线程完成求和所消耗的总时间并打印。和 (1)、(2) 的耗费时间相比，你能否解释 (3) 的耗时结果？(注意：可以多运行几次看测量结果)

提示:

cpu_set_t 类型，*CPU_ZERO*、*CPU_SET* 宏，以及 *sched_setaffinity* 函数可以用来进行绑核操作，它们的定义在 *sched.h* 文件中。请调研了解上述绑核操作。以下是一个参考示例。

假设你的电脑有两个核 *core*0 和 *core*1，同时你创建了两个线程 *thread*1 和 *thread*2，则可以用以下代码在线程执行的函数中进行绑核操作。

示例代码:

```c
//需要引入的头文件和宏定义
#define __USE_GNU
#include <sched.h>
#include <pthread.h>

//线程执行的函数
void *worker(void *arg){
cpu_set_t cpuset;   //CPU核的位图
   CPU_ZERO(&cpuset);  //将位图清零
   CPU_SET(N, &cpuset);  //设置位图第N位为1，表示与core N绑定。N从0开始计数
   sched_setaffinity(0, sizeof(cpuset), &cpuset);  //将当前线程和cpuset位图中指定的核绑定运行

   //其他操作
}
```

提交内容:

(1) 所写 *C* 程序，打印结果截图等

(2) 所写 *C* 程序，打印结果截图，分析说明等

(3) 所写 *C* 程序，打印结果截图，分析说明等

4.2

请调研了解 *pthread_create*, *pthread_join*, *pthread_exit* 等 *API* 的使用方法后，完成以下任务：

（1）写一个 *C* 程序，首先创建一个有 100 万个元素的整数型空数组，然后使用 *pthread* 创建 N 个线程（N 可以自行决定，且 $N > 1$），由这 N 个线程完成前述 100 万个元素数组的赋值（注意：赋值时第 i 个元素的值为 i）。最后由主进程对该数组的 100 万个元素求和，并打印结果，验证线程已写入数据。

提交内容：

（1）所写 *C* 程序，打印结果截图，关键代码注释等

pthread 函数库

pthread（*POSIX Threads*）是一种多线程编程库，允许在一个进程中创建和管理多个线程。它提供了一组函数和数据类型，用于在一个进程中创建、管理和同步多个线程，通常用于 *Unix* 和类 *Unix* 操作系统（如 *Linux*）上，以支持多线程编程。

pthread_create

用于创建一个新的线程。函数参数指定了线程的属性、入口函数和入口函数的参数。

函数原型如下：

```
int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
                   void *(*start_routine)(void *), void *arg);
```

函数参数：

thread	用于存储新线程的标识符的 *pthread_t* 类型指针。
attr	线程属性，通常可以传入 NULL 以使用默认属性。
start_routine	线程的入口函数，是线程开始执行的地方。
arg	传递给入口函数的参数。

pthread_join

用于等待一个指定线程的结束执行，并获取线程的返回值。

函数原型如下：

```
int pthread_join(pthread_t thread, void **retval);
```

函数参数：

thread	要等待的线程的标识符。
retval	用于存储线程返回值的指针，如果不需要返回值，可以传入 *NULL*。

pthread_exit

用于退出当前线程，并可选地返回一个值。函数原型如下：

```
void pthread_exit(void *retval);
```

函数参数：

retval	线程的返回值，可以是一个指针，如果不需要返回值，可以传入 *NULL*。

其它函数

pthread_mutex_init	初始化互斥锁，以确保多个线程能够安全地访问共享资源。
pthread_mutex_destroy	销毁互斥锁，以确保多个线程能够安全地访问共享资源。
pthread_mutex_lock	在访问共享资源前锁定互斥锁，以防止竞争条件。
pthread_mutex_unlock	在访问共享资源后解锁互斥锁，以防止竞争条件。
pthread_cond_init	初始化条件变量，以便线程能够等待某个条件的发生。
pthread_cond_destroy	销毁条件变量，以便线程能够等待某个条件的发生。
pthread_cond_wait	等待条件的发生。
pthread_cond_signal	唤醒一个或多个等待线程。
pthread_cond_broadcast	唤醒一个或多个等待线程。
pthread_detach	用于将线程标记为“分离”，使其结束时自动释放资源。
pthread_attr_init	初始化线程属性对象，以配置线程的属性。
pthread_attr_destroy	销毁线程属性对象，以配置线程的属性。
pthread_attr_setxxx	设置线程属性，如栈大小、调度策略等。
pthread_attr_getxxx	获取线程属性，如栈大小、调度策略等。

赋值操作一般需要三步：从栈中取出数据放到寄存器上、寄存器修改数据、将数据存回栈上。

两个线程如果同时取栈上数据，那么会取到相同值，最后两者计算结果是一样的，写回栈上的也是同一个值，而非每个单独累加。因此引入了锁的概念，对一个互斥量加锁后，其他希望获取锁的线程将会阻塞，直到已经获取锁的线程释放锁，其他线程才能够获取锁。这个锁称为互斥锁。

Task4.1

(1)

C 程序源码：

```c
#include <stdio.h>
#include <time.h>
#define NUMTOADD 1000000

int main()
{
    /* 创建从 1 到 100 万的数组 */
    int num[NUMTOADD];
    for (int i = 0; i < NUMTOADD; ++i)
        num[i] = i + 1;

    /* 求和 */
    long long sum = 0;
    struct timespec time1 = {0, 0};
    struct timespec time2 = {0, 0};
    clock_gettime(CLOCK_REALTIME, &time1);//开始计时
    for (int j = 0; j < NUMTOADD; ++j)
        sum += num[j];
    clock_gettime(CLOCK_REALTIME, &time2);//停止计时
    printf("sum: %lld\n", sum);
    printf("time = %lu.%09ld\n", time2.tv_sec-time1.tv_sec, time2.tv_nsec-time1.tv_nsec);
}
```

运行结果：

```
stu@stu:~/OS/task4$ ./a.out
sum  = 500000500000
time = 0.001331992
stu@stu:~/OS/task4$ ./a.out
sum  = 500000500000
time = 0.000994696
stu@stu:~/OS/task4$ ./a.out
sum  = 500000500000
time = 0.001017970
stu@stu:~/OS/task4$ ./a.out
sum  = 500000500000
time = 0.001411886
stu@stu:~/OS/task4$ ./a.out
sum  = 500000500000
time = 0.001002161
stu@stu:~/OS/task4$ ./a.out
sum  = 500000500000
time = 0.000987852
```

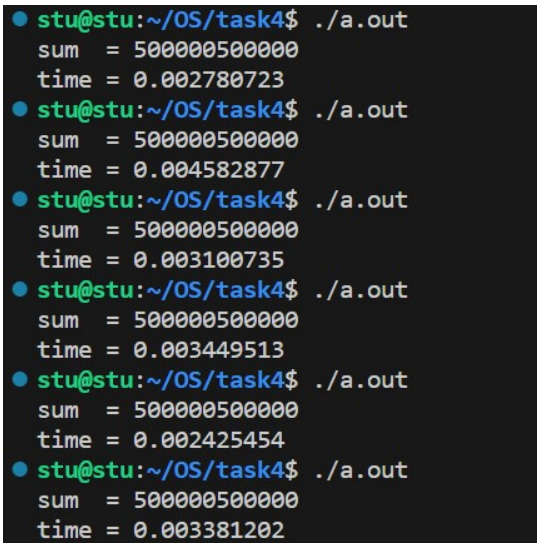
使用 shell 脚本运行 10000 次，取平均值得到结果为 0.0011328859990989188。

(2)

C 程序源码：（采用五个线程）

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <time.h>
#define NUM_THREADS 5
#define NUMTOADD    1000000

long long sum = 0;
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
int num[NUMTOADD];

void *part_sum(void *arg) {
    long long thread_id = (long long)arg;
    long long start = thread_id * (NUMTOADD / NUM_THREADS);
    long long end = (thread_id + 1) * (NUMTOADD / NUM_THREADS);
    long long local_sum = 0;

    for (long long i = start; i < end; i++) {
        local_sum += num[i];
    }
    pthread_mutex_lock(&mutex);
    sum += local_sum;
    pthread_mutex_unlock(&mutex);
    pthread_exit(NULL);
}

int main() {
    struct timespec time1, time2;
    double elapsed_time;
    pthread_t threads[NUM_THREADS];
    long long thread_ids[NUM_THREADS];

    /* 创建从 1 到 100 万的数组 */
    for (int i = 0; i < NUMTOADD; ++i)
        num[i] = i + 1;

    clock_gettime(CLOCK_MONOTONIC, &time1);
    // 创建多个线程
    for (int i = 0; i < NUM_THREADS; i++) {
        thread_ids[i] = i;
        pthread_create(&threads[i], NULL, part_sum, (void *)thread_ids[i]);
    }
    // 等待所有线程完成
    for (int i = 0; i < NUM_THREADS; i++) {
        pthread_join(threads[i], NULL);
    }
    clock_gettime(CLOCK_MONOTONIC, &time2);
    printf("sum  = %lld\n", sum);
    printf("time = %lu.%09ld\n", time2.tv_sec-time1.tv_sec, time2.tv_nsec-time1.tv_nsec);

    pthread_mutex_destroy(&mutex);
    return 0;
}
```

运行结果：（部分）



使用 shell 脚本运行 10000 次，取平均值得到结果如下：

线程数	平均耗时
2	0.0023797551834853193
3	0.003068566911416533
4	0.003805266041984733
5	0.005478056498892692
6	0.006763863197543294

分析结果可以看出，使用多线程进行求和操作耗时反而高于单线程求和，而且所使用的线程数越多，平均耗时就越久。原因可能是因为多线程涉及到线程的创建、管理和同步开销，这些开销可能会超过线程并行执行所带来的性能提升，尤其是在计算密集型的任务中。求和操作本身是一个相对简单的计算密集型任务，在这种情况下，创建和管理多个线程以进行求和可能会增加额外的开销，包括线程的创建、上下文切换和互斥锁的开销。这些开销可能会抵消多线程并行计算所带来的好处。

同时，还发现随着线程的增加，出现超时的情况也会增多。由于我的时间计算机制为整数与小数部分分开计算，仅能支持 1 秒以内的数字，整数部分不为 0 意味着超时。容易观察到出现大于 1 秒的情况属于异常情况，原因可能是各个线程之间出现了冲突，导致等待时间被拉长。

(3)

C 程序源码：（采用五个线程）

```c
#define _GNU_SOURCE
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <sched.h>
#include <time.h>
#define NUM_THREADS 5
#define NUMTOADD    1000000

long long sum = 0;
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
int num[NUMTOADD];

void *part_sum(void *arg) {
    long long thread_id = (long long)arg;
    long long start = thread_id * (NUMTOADD / NUM_THREADS);
    long long end = (thread_id + 1) * (NUMTOADD / NUM_THREADS);
    long long local_sum = 0;

    cpu_set_t cpuset;
    CPU_ZERO(&cpuset);
    CPU_SET(thread_id, &cpuset);
    sched_setaffinity(0, sizeof(cpuset), &cpuset);

    for (long long i = start; i < end; i++) {
        local_sum += num[i];
    }
    pthread_mutex_lock(&mutex);
    sum += local_sum;
    pthread_mutex_unlock(&mutex);
    pthread_exit(NULL);
}

int main() {
    struct timespec time1, time2;
    double elapsed_time;
    pthread_t threads[NUM_THREADS];
    long long thread_ids[NUM_THREADS];

    /* 创建从 1 到 100 万的数组 */
    for (int i = 0; i < NUMTOADD; ++i)
        num[i] = i + 1;

    clock_gettime(CLOCK_MONOTONIC, &time1);
    // 创建多个线程
    for (int i = 0; i < NUM_THREADS; i++) {
        thread_ids[i] = i;
        pthread_create(&threads[i], NULL, part_sum, (void *)thread_ids[i]);
    }
    // 等待所有线程完成
    for (int i = 0; i < NUM_THREADS; i++) {
        pthread_join(threads[i], NULL);
    }
    clock_gettime(CLOCK_MONOTONIC, &time2);
    printf("sum  = %lld\n", sum);
    printf("time = %lu.%09ld\n", time2.tv_sec-time1.tv_sec, time2.tv_nsec-time1.tv_nsec);

    pthread_mutex_destroy(&mutex);
    return 0;
}
```

运行结果：（部分）

```
stu@stu:~/OS/task4$ ./a.out
sum  = 500000500000
time = 0.018955955
stu@stu:~/OS/task4$ ./a.out
sum  = 500000500000
time = 0.001834900
stu@stu:~/OS/task4$ ./a.out
sum  = 500000500000
time = 0.001938791
stu@stu:~/OS/task4$ ./a.out
sum  = 500000500000
time = 0.020183815
stu@stu:~/OS/task4$ ./a.out
sum  = 500000500000
time = 0.008574136
stu@stu:~/OS/task4$ ./a.out
sum  = 500000500000
time = 0.001983793
```

使用 *shell* 脚本运行 10000 次，取平均值得到结果为 0.006247565142310789。

可以发现，绑核操作后，平均耗时有一定减少。由于我的电脑为 6 核，我将计算分为 5 部分由 5 个线程分别执行，同时将这些线程进行绑核，这样一来五个线程同时开始进行，但是由于最终结果 sum 只能同时被一个线程修改，而这些线程几乎同时进行到修改 sum 这一步，所以需要排队修改 sum，主要在这一步耗费时间，因此绑核的提升并不大。

Task4.2

```c
#define _GNU_SOURCE
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <sched.h>
#define NUM_THREADS 5
#define NUMTOADD    1000000

long long sum = 0;
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
int num[NUMTOADD];

void *init_num(void *arg) {
    long long thread_id = (long long)arg;
    long long start = thread_id * (NUMTOADD / NUM_THREADS);
    long long end = (thread_id + 1) * (NUMTOADD / NUM_THREADS);

    for (long long i = start; i < end; i++) {
        num[i] = i + 1;
    }
    pthread_exit(NULL);
}

int main() {
    struct timespec time1, time2;
    double elapsed_time;
    pthread_t threads[NUM_THREADS];
    long long thread_ids[NUM_THREADS];
    long long sum = 0;

    for (int i = 0; i < NUM_THREADS; i++) {
        thread_ids[i] = i;
        pthread_create(&threads[i], NULL, init_num, (void *)thread_ids[i]);
    }
    for (int i = 0; i < NUM_THREADS; i++) {
        pthread_join(threads[i], NULL);
    }

    for (int j = 0; j < NUMTOADD; ++j)
        sum += num[j];
    printf("sum  = %lld\n", sum);

    pthread_mutex_destroy(&mutex);
    return 0;
```

```
sum  = 500000500000
```