作业 4

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 core1,同时你创建了两个线程 thread1 和 thread2，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：

```
//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

各函数 API 用法详解：

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

1、pthread_t *thread：

传递一个 pthread_t 类型的指针变量，也可以直接传递某个 pthread_t 类型变量的地址。pthread_t 是一种用于表示线程的数据类型，每一个 pthread_t 类型的变量都可以表示一个线程。(pthread_t 类型在 linux 下被定义为： “unsigned long int”)

2、const pthread_attr_t *attr：

用于手动设置新建线程的属性，例如线程的调用策略、线程所能使用的栈内存的大小等。大部分场景中，我们都不需要手动修改线程的属性，将 attr 参数赋值为 NULL，pthread_create() 函数会采用系统默认的属性值创建线程。

3、void *(start_routine) (void)：

以函数指针的方式指明新建线程需要执行的函数，该函数的参数最多有 1 个（可以省略不写），形参和返回值的类型都必须为 void 类型。void 类型又称空指针类型，表明指针所指数据的类型是未知的。使用此类型指针时，我们通常需要先对其进行强制类型转换，然后才能正常访问指针指向的数据。

4、void *arg：

指定传递给 start_routine 函数的实参，当不需要传递任何数据时，将 arg 赋值为 NULL 即可。如果成功创建线程，pthread_create() 函数返回数字 0，反之返回非零值。各个非零值都对应着不同的宏，

指明创建失败的原因，常见的宏有以下几种：

- EAGAIN：系统资源不足，无法提供创建线程所需的资源。
- EINVAL：传递给 pthread_create() 函数的 attr 参数无效。
- EPERM：传递给 pthread_create() 函数的 attr 参数中，某些属性的设置为非法操作，程序没有相关的设置权限。

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

thread 参数用于指定接收哪个线程的返回值；retval 参数表示接收到的返回值，如果 thread 线程没有返回值，又或者我们不需要接收 thread 线程的返回值，可以将 retval 参数置为 NULL。

pthread_join() 函数会一直阻塞调用它的线程，直至目标线程执行结束（接收到目标线程的返回值），阻塞状态才会解除。如果 pthread_join() 函数成功等到了目标线程执行结束（成功获取到目标线程的返回值），返回值为数字 0；反之如果执行失败，函数会根据失败原因返回相应的非零值，每个非零值都对应着不同的宏，例如：

- EDEADLK：检测到线程发生了死锁。
- EINVAL：分为两种情况，要么目标线程本身不允许其它线程获取它的返回值，要么事先就已经有线程调用 pthread_join() 函数获取到了目标线程的返回值。

- ESRCH：找不到指定的 thread 线程。

以上这些宏都声明在 <errno.h> 头文件中，如果程序中想使用这些宏，需提前引入此头文件。

注意：一个线程执行结束的返回值只能由一个 pthread_join() 函数获取，当有多个线程调用 pthread_join() 函数获取同一个线程的执行结果时，哪个线程最先执行 pthread_join() 函数，执行结果就由那个线程获得，其它线程的 pthread_join() 函数都将执行失败。

```
void pthread_exit(void *value_ptr);
```

`pthread_exit` 函数用于在线程中显式地退出，单个线程可以通过下列三种方式退出，在不终止整个进程的情况下停止它的控制流。

●线程只是从启动例程中返回，返回值是线程的退出码。

●线程可以被同一进程中的其他线程取消

●线程调用 pthread_exit()

value_ptr:是一个无类型指针，与传给启动例程的单个参数类似进程中的其他线程可以通过调用 pthread_join()函数访问到这个指针。其指向的数据将作为线程退出时的返回值。如果线程不需要返回任何数据，将 value_ptr 参数置为 NULL 即可

4.1

(1)、单线程求和

源码：

```
#define __USE_GNU
#include<stdio.h>
#include<stdlib.h>
#include<sys/time.h>
#define MAXNUM 1000000
int main(){
   long long int sum=0;
   int array[MAXNUM];
   struct timeval START,END;
   for(int i=0;i<MAXNUM;i++){array[i]=i+1;}
   gettimeofday(&START,NULL);
   for(int j=0; j<MAXNUM; sum+=array[j++]);
   gettimeofday(&END,NULL);
   printf("sum=%lld\n",sum);
   printf("the eplased time is :%ld\n",(END.tv_usec-START.tv_usec));
   return 0;
}
```

运行结果如下：

```
solomon@DESKTOP-23PER74:~/CODES/C/OS/homework4$ ./sum
sum=500000500000
the eplased time is :1704
solomon@DESKTOP-23PER74:~/CODES/C/OS/homework4$ ./sum
sum=500000500000
the eplased time is :1523
solomon@DESKTOP-23PER74:~/CODES/C/OS/homework4$ ./sum
sum=500000500000
the eplased time is :1440
solomon@DESKTOP-23PER74:~/CODES/C/OS/homework4$ ./sum
sum=500000500000
the eplased time is :1510
solomon@DESKTOP-23PER74:~/CODES/C/OS/homework4$ ./sum
sum=500000500000
the eplased time is :1651
```

（2）无绑核多线程求和：

```
#define __USE_GNU
#include <sched.h>
#include <pthread.h>
#include <sys/time.h>
#include <stdio.h>

#define ARRAY_SIZE 1000000
#define THREADS_NUM 5
int cur=0;
int array[ARRAY_SIZE];
long long int P_sum=0;

void *sum(void* arg){
   while(cur<ARRAY_SIZE)
      P_sum+=array[cur++];
   pthread_exit(NULL);
}

int main(){
   pthread_t threads[THREADS_NUM];
   int state[THREADS_NUM];
   struct timeval START,END;
   long elapse_time;
   int i;
   for(i=0; i<ARRAY_SIZE; array[i++]=i+1);
    gettimeofday(&START, NULL);
   for ( i = 0; i < THREADS_NUM; i++) {
      pthread_create(&threads[i], NULL, &sum, &i);
   }
   for(i=0; i<THREADS_NUM; i++)
      state[i]=pthread_join(threads[i], NULL);
   gettimeofday(&END, NULL);
   for(i=0; i<THREADS_NUM; i++){
      if(state[i]){
         printf("thread %d to finish failed.\n", i);
         return -1;
      }}
   elapse_time = (END.tv_sec - START.tv_sec) * 1000000 + (END.tv_usec -
START.tv_usec);
   printf("sum=%lld, elapse_time=%ld\n", P_sum, elapse_time);
   return 0;
}
```

共五个线程，运行结果如下：

```
solomon@DESKTOP-23PER74:~/CODES/C/OS/homework4$ ./sum_pthread
sum=530696057490, elapse_time=4063
solomon@DESKTOP-23PER74:~/CODES/C/OS/homework4$ ./sum_pthread
sum=512053606355, elapse_time=3224
solomon@DESKTOP-23PER74:~/CODES/C/OS/homework4$ ./sum_pthread
sum=518356811276, elapse_time=3678
solomon@DESKTOP-23PER74:~/CODES/C/OS/homework4$ ./sum_pthread
sum=510905400149, elapse_time=3555
solomon@DESKTOP-23PER74:~/CODES/C/OS/homework4$ ./sum_pthread
sum=545980275338, elapse_time=3640
solomon@DESKTOP-23PER74:~/CODES/C/OS/homework4$ ./sum_pthread
sum=529985889408, elapse_time=3901
solomon@DESKTOP-23PER74:~/CODES/C/OS/homework4$
```

开辟四个线程，求和结果如下：

```
solomon@DESKTOP-23PER74:~/CODES/C/OS/homework4$ ./sum_pthread
sum=504634170660, elapse_time=3457
solomon@DESKTOP-23PER74:~/CODES/C/OS/homework4$ ./sum_pthread
sum=525646629350, elapse_time=3770
solomon@DESKTOP-23PER74:~/CODES/C/OS/homework4$ ./sum_pthread
sum=533377802724, elapse_time=3575
solomon@DESKTOP-23PER74:~/CODES/C/OS/homework4$ ./sum_pthread
sum=491849456832, elapse_time=3314
```

注意到此时多线程的耗时反而远大于单线程求和，同时求和结果不等于预期值。
可以通过 gdb 添加对 cur 变量的监视，可以发现：

```
Old value = 1
New value = 3
sum (arg=0x7fffffffdc04) at sum_pthread.c:15
15              P_sum+=array[cur++];
(gdb)
[Switching to Thread 0x7ffff6da1700 (LWP 18597)]

Thread 4 "sum_pthread" hit Hardware watchpoint 2: cur

Old value = 3
New value = 4
sum (arg=0x7fffffffdc04) at sum_pthread.c:15
15              P_sum+=array[cur++];
(gdb)
[Switching to Thread 0x7ffff65a0700 (LWP 18622)]

Thread 5 "sum_pthread" hit Hardware watchpoint 2: cur

Old value = 4
New value = 6
sum (arg=0x7fffffffdc04) at sum_pthread.c:15
```

这样的原因是不同线程交替访问全局变量导致数据冲突，如若两个线程恰都执行到递增步骤，就有可能跳过对某项数据的求和，导致求和结果偏小。冲突访问和线程的不断切换也导致耗时长于单线程。

（3）、绑核多线程求和

首先确定了虚拟机使用的 CPU 核数为：

```
solomon@DESKTOP-23PER74:~/CODES/C/OS/homework4$ cat /proc/cpuinfo| grep "cpu cores"| uniq
cpu cores       : 8
```

可以进行绑核操作

源码：

```
#include <stdio.h>
```

```c
#define __USE_GNU
#include <sched.h>
#include <pthread.h>
#include <sys/time.h>


#define ARRAY_SIZE 1000000
#define THREADS_NUM 5
int array[ARRAY_SIZE];
int cur = 0;
long long int P_sum=0;
void *sum(void* arg){
   int core_id = *(int*)arg;
   cpu_set_t cpuset;   //CPU 核的位图
   CPU_ZERO(&cpuset); //将位图清零
   CPU_SET(core_id, &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
   while(cur<ARRAY_SIZE)
      P_sum+=array[cur++];
   pthread_exit(NULL);
}

int main(){
   pthread_t threads[THREADS_NUM];
   int state[THREADS_NUM];
   struct timeval START,END;
   int core_id[]={0, 1, 2, 3, 4};
   long elapse_time;
   int i;
   for(i=0; i<ARRAY_SIZE; array[i++]=i+1);
    gettimeofday(&START, NULL);
   for ( i = 0; i < THREADS_NUM; i++) {
      pthread_create(&threads[i], NULL, &sum,&core_id[i]);
   }
   for(i=0; i<THREADS_NUM; i++)
      state[i]=pthread_join(threads[i], NULL);
   gettimeofday(&END, NULL);
   for(i=0; i<THREADS_NUM; i++){
      if(state[i]){
         printf("Fail to wait thread %d to finish.\n", i);
         return -1;
      }}
```

```
    elapse_time = (END.tv_sec - START.tv_sec) * 1000000 + (END.tv_usec -
START.tv_usec);
    printf("sum=%lld, elapse_time=%ld\n", P_sum, elapse_time);
    return 0;
}
```

注：务必注意头文件的顺序，pthread.h 中也 include 了<sched.h>，但是没有定义__USE_GNU，所以后面的 include <sched.h>就没有作用了。如果是 pthread.h 是在 sched.h 前面，会导致我们自己的 c 文件中的 sched.h 没有被包含进来，而是包含别的头文件中的 sched.h，因此 thread.h 的定义需要放到 sched.h 的后面，否则会出现无法编译的错误。

运行结果：

```
solomon@DESKTOP-23PER74:~/CODES/C/OS/homework4$ ./sum_pthread_core
bound to core 0
bound to core 2
bound to core 1
bound to core 3
bound to core 4
sum=521216504293, elapse_time=3966
```

可以看到每个线程绑定的核 id 并不同，但是发现运行时间并未缩短，且求和的结果仍是小于预期，猜测原因是因为由于代码定义的 P_sum 和 cure 均为经过初始化的全局变量，会将其放置在 bss 段，因此程序实际上会通过寄存器寻址的方式将 cur 的值从 bss 段中读出，将 cur 递增后再存于 bss 段一次求和后再将 cur 的值从 bss 段中读出，并比较其值与循环结束值的大小。其对 sum 求和也是如此，反复读出、加和、再写回 bss 段。因此仍然会发生访问冲突，跳过部分值的求和。

```
    .globl  cur
    .bss
    .align 4
    .type   cur, @object
    .size   cur, 4
cur:
    .zero   4
    .globl  P_sum
    .align 8
    .type   P_sum, @object
    .size   P_sum, 8
P_sum:
    .zero   8
    .section    .rodata
```

总结：

在汇编代码中一次加和操作被分为若干条指令，而在多线程运行的过程中，各个线程在高速切换的过程中可能互相打断，可能导致 cur 的值更新若干次才执行一次累加求和操作，导致计算出的值偏小，同时耗时也会增加。

另外，多核的性能也未必就优于单核。在单核 CPU 上，数百次的间隔检查才会导致一次线程切换；而在多核 CPU 上，存在严重的线程颠簸（thrashing），将导致多核多线程的效率反而低于单核多线程。另一方面，若一块内存被多个 CPU 频繁使用，它就会出现在多个 CPU 的 L1 Cache 中。若其中某个 CPU 修改了这块内存，其他 CPU 中相应的 L1

Cache 会更新或失效以保持缓存一致性，故如果多个核频繁读写同一块内存，会比单核更慢。

以 python 的全局解释器 GIL （Global Interpreter Lock）为例， Python 的 GIL 全局解释器锁会阻止 Python 代码同时在多个处理器核心上运行，因为一个 Python 解释器在同一时刻只能运行于一个处理器之中。

单核下多线程，每次释放 GIL，唤醒的那个线程都能获取到 GIL 锁，所以能够无缝执行，但多核下，CPU0 释放 GIL 后，其他 CPU 上的线程都会进行竞争，但 GIL 可能会马上又被 CPU0 拿到，导致其他几个 CPU 上被唤醒后的线程会醒着等待直到切换时间结束后又进入待调度状态，导致线程颠簸，降低了系统效率。

如何产生正确结果:

规定好代码中每个线程负责加法的部分，就可以有效避免冲突。同时可以有效减少多进程的时耗。

```
#include  <stdio.h>
#define __USE_GNU
#include <sched.h>
#include <pthread.h>
#include <sys/time.h>
#define THREADS_NUM 5
#define ARRAY_SIZE 1000000

int array[ARRAY_SIZE];
long long int result_from_pthread[THREADS_NUM]={0};

void *sum(void* arg){
    int core_id = *(int*)arg;
    cpu_set_t cpuset;   //CPU 核的位图
    CPU_ZERO(&cpuset);  //将位图清零
    CPU_SET(core_id, &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
    int cur=*(int *)arg ;
    int i,low,max;
    low=cur*ARRAY_SIZE/THREADS_NUM;
    max=(cur+1)*ARRAY_SIZE/THREADS_NUM;
    for(i=low ; i<max;result_from_pthread[cur]+=array[i++]);
    pthread_exit(NULL);
}

int main(){
    pthread_t threads[THREADS_NUM];
    int state[THREADS_NUM];
    struct timeval START,END;
```

```
    long elapse_time;
    long long int P_sum=0;
    int cpu_id[THREADS_NUM];
    int i;
    for(i=0; i<ARRAY_SIZE; array[i++]=i+1);
    for(i=0; i<THREADS_NUM; cpu_id[i++]=i-1);
    gettimeofday(&START, NULL);
    for ( i = 0; i < THREADS_NUM; i++) {
        pthread_create(&threads[i], NULL, &sum, &cpu_id[i]);
    }
    for(i=0; i<THREADS_NUM; i++)
        state[i]=pthread_join(threads[i],NULL);
    gettimeofday(&END, NULL);
    for(i=0; i<THREADS_NUM; i++){
        if(state[i]){
            printf("Fail to wait thread %d to finish.\n", i);
            return -1;
        }}
    for(i=0; i<THREADS_NUM; i++){
        P_sum+=result_from_pthread[i];
    }
    elapse_time = (END.tv_sec - START.tv_sec) * 1000000 + (END.tv_usec
- START.tv_usec);
    printf("sum=%lld, elapse_time=%ld\n", P_sum, elapse_time);
    return 0;
}
```

运行结果如下：

可以发现其结果表明，耗时并未优于单线程，至于原因已在上文中解释。这里加法求得的和与理论值不符合，但是求和的结果是稳定的，我尚未找到原因。

```
solomon@DESKTOP-23PER74:~/CODES/C/OS/homework4$ ./sum_pthread2
sum=500001500000, elapse_time=1772
solomon@DESKTOP-23PER74:~/CODES/C/OS/homework4$ ./sum_pthread2
sum=500001500000, elapse_time=1583
solomon@DESKTOP-23PER74:~/CODES/C/OS/homework4$ ./sum_pthread2
sum=500001500000, elapse_time=1664
solomon@DESKTOP-23PER74:~/CODES/C/OS/homework4$ ./sum_pthread2
sum=500001500000, elapse_time=1903
```

4.2

源码：

```
#define __USE_GNU
#include <sched.h>
#include <pthread.h>
#include <sys/time.h>
#include  <stdio.h>
```

```c
#define MAXNUM 1000000
#define THREAD_NUM 5
#define STRIDE  MAXNUM/THREAD_NUM
int array[MAXNUM];
int cur;

//线程执行的函数
void *assignment(void *arg){
    int low, max;
    low = *(int*)arg;
    max = low + STRIDE;
    while(low<max){
        array[low++]= low;
    }
    pthread_exit(NULL);
}

int main(){
    pthread_t threads[THREAD_NUM];
    struct timeval START, END;
    int i;
    long elapse_time;
    int state[THREAD_NUM];
    long long int sum=0;
    int low[THREAD_NUM];
    // 确定每个线程赋值的起始位置
    for(i=0;i<THREAD_NUM;i++)
        low[i]= i*STRIDE;
    // 创建 N 个线程，并对数组赋值
    gettimeofday(&START, NULL);
    for(i=0; i<THREAD_NUM; i++){
        pthread_create(&threads[i], NULL, &assignment, &low[i]);
    }
    //等待所有线程结束
    for(i=0; i<THREAD_NUM; i++)
        state[i]=pthread_join(threads[i], NULL);
    gettimeofday(&END, NULL);
    for(i=0; i<THREAD_NUM; i++){
        if(state[i]){   // 若失败返回值为 0
            printf("Fail to wait thread %d to finish.\n", i);
            return -1;
        }
    }
    for(i=0;i<MAXNUM;i++)
```

```
        sum+=array[i];
    elapse_time = (END.tv_sec - START.tv_sec) * 1000000 + (END.tv_usec
- START.tv_usec);
    // 打印当前线程数、计算结果、总用时
    printf("sum=%lld, elapse_time=%ld\n", sum, elapse_time);
}
```

运行结果：

线程数为 5：

```
solomon@DESKTOP-23PER74:~/CODES/C/OS/homework4$ ./sum_assign
sum=500000500000, elapse_time=950
solomon@DESKTOP-23PER74:~/CODES/C/OS/homework4$
solomon@DESKTOP-23PER74:~/CODES/C/OS/homework4$ ./sum_assign
sum=500000500000, elapse_time=901
solomon@DESKTOP-23PER74:~/CODES/C/OS/homework4$ ./sum_assign
sum=500000500000, elapse_time=859
solomon@DESKTOP-23PER74:~/CODES/C/OS/homework4$ ./sum_assign
sum=500000500000, elapse_time=1652
solomon@DESKTOP-23PER74:~/CODES/C/OS/homework4$ ./sum_assign
sum=500000500000, elapse_time=1294
```

线程数为 2：

```
solomon@DESKTOP-23PER74:~/CODES/C/OS/homework4$ ./sum_assign
sum=500000500000, elapse_time=1535
solomon@DESKTOP-23PER74:~/CODES/C/OS/homework4$ ./sum_assign
sum=500000500000, elapse_time=2720
solomon@DESKTOP-23PER74:~/CODES/C/OS/homework4$ ./sum_assign
sum=500000500000, elapse_time=802
solomon@DESKTOP-23PER74:~/CODES/C/OS/homework4$ ./sum_assign
sum=500000500000, elapse_time=1357
solomon@DESKTOP-23PER74:~/CODES/C/OS/homework4$ ./sum_assign
sum=500000500000, elapse_time=1143
solomon@DESKTOP-23PER74:~/CODES/C/OS/homework4$
```

从结果表现上来看，多线程也能确保每个元素都被赋值，且发现此多线程的情况浮动较大，有时会优于单线程的表现。