操作系统 第三次作业

姓名：朱首赫

学号：2023K8009906029

3.1 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 值可以自行决定，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。要求：1）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）2）请尝试设置不同的 N 值进行求和，请描述并尝试解释求和耗费时间与 N 值之间的关系。例如，是否 N 值越大，耗费时间越短？建议使用量化数据（图或表形式）来展现耗费时间和 N 值的关系。

（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（N 值自行决定）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。

Listing 1: 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

解答：

(1) 程序源码如表 2 ，为避免随机性的影响，重复实验了 10 次并最终取平均值，运行结果如图 1 ，平均用时为 3510799 ns。

Listing 2: 单线程数组求和

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define ARR_LEN 1000000
#define TEST_TIMES 10
int main(int argc, char const *argv[])
{
    int *arr = (int*)malloc(sizeof(int) * ARR_LEN);
    for (int i = 0; i < ARR_LEN; i++)
    {
        arr[i] = i + 1;
    }
    struct timespec start, end;
    long diff = 0;
    long average = 0;
    for (int j = 0; j < TEST_TIMES; j++)
    {
        long long sum = 0;
        clock_gettime(CLOCK_MONOTONIC, &start);
        for (int i = 0; i < ARR_LEN; i++)
        {
            sum += arr[i];
        }
        clock_gettime(CLOCK_MONOTONIC, &end);
        diff = (end.tv_sec - start.tv_sec) * 1000000000L + (end.tv_nsec - start.tv_nsec);
        average += diff;
        printf("[test %2d] Single-thread took %7ld ns\n", j, diff);
    }
    average /= TEST_TIMES;
    printf("[average] Single-thread took %7ld ns\n", average);
    free(arr);
    return 0;
}
```

```
zsh@kolp:~/VScodeproject/UCAS-2025-OS-Theory/os_hw_code/hw3$ ./3.1.1
[test  0] Single-thread took 3289991 ns
[test  1] Single-thread took 3328709 ns
[test  2] Single-thread took 3507226 ns
[test  3] Single-thread took 3241472 ns
[test  4] Single-thread took 3399935 ns
[test  5] Single-thread took 3599730 ns
[test  6] Single-thread took 3647650 ns
[test  7] Single-thread took 3735299 ns
[test  8] Single-thread took 3934036 ns
[test  9] Single-thread took 3423942 ns
[average] Single-thread took 3510799 ns
```

图 1: 单线程数组求和用时

(2) 1) 由于 pthread_create 只能给调用的 partial_add 函数传递一个 void* 类型的参数，所以必须定义一个结构体 thread_arg_t 来传递多个参数。还要注意 pthread 是一个外部库，不属于 C 语言的标准库，编译时需要加上 -pthread 明确链接这个库。

程序源码如表 3 ，同样重复实验了 10 次并最终取运行结果如图 2 ，平均运行时间为 1266436 ns。通过计算得出，双线程是单线程运行速度的 2.58 倍，高于预期的 2 倍。经调研后推测，这种超线性的加速结果应该和 CPU 缓存 (CPU Cache) 有关：由于每个核心处理的数据量更小了，这些数据可能更好地装进了 CPU 的高速缓存里，导致访问速度更快。于是我在任务管理器中查看了本机 CPU Cache 大小，如图 3 所示，一级缓存的大小为 2.1 MB，恰巧 50 万 int 数组的大小为 $500,000 \times 4B = 2MB$，基本与 Cache 大小一致，为该推测提供了支持。

Listing 3: 双线程数组求和

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <pthread.h>
#define ARR_LEN 1000000
#define TEST_TIMES 10
#define N 2
#define CHUNK_SIZE (ARR_LEN / N)

// 定义一个结构体来打包线程参数
typedef struct {
    int thread_id;      // 线程ID
    int* arr;           // 数组的指针
    int start_index;    // 起始索引
    int end_index;      // 结束索引
} thread_arg_t;
void* partial_add(void* arg){
    thread_arg_t *my_arg = (thread_arg_t *)arg;
    int start_index = my_arg->start_index, end_index = my_arg->end_index;
    int *arr = my_arg->arr;
    long long *local_sum = (long long *)malloc(sizeof(long long));
    *local_sum = 0;
```

```
23
24      for (int i = start_index; i <= end_index; i++)
25      {
26          *local_sum += arr[i];
27      }
28      return (void *)local_sum;
29  }
30
31  int main(int argc, char const *argv[])
32  {
33      long long sum = 0;
34      int *arr = (int*)malloc(sizeof(int) * ARR_LEN);
35      for (int i = 0; i < ARR_LEN; i++)
36          arr[i] = i + 1;
37
38      struct timespec start, end;
39      long diff = 0;
40      long average = 0;
41
42      for (int j = 0; j < TEST_TIMES; j++)
43      {
44          sum = 0;
45          pthread_t threads[N];
46          thread_arg_t args[N];
47          clock_gettime(CLOCK_MONOTONIC, &start);
48          for (int i = 0; i < N; i++)
49          {
50              args[i].thread_id = i;
51              args[i].arr = arr;
52              args[i].start_index = i * CHUNK_SIZE;
53              args[i].end_index = (i == N - 1) ? (ARR_LEN - 1) : ((i+1) * CHUNK_SIZE - 1);
54              pthread_create(&threads[i], NULL, partial_add, (void *)&args[i]);
55          }
56          // 循环等待所有子线程，并收集它们的返回值
57          long long *partial_sum;
58          for (int i = 0; i < N; i++)
59          {
60              void *ret_val;
61              pthread_join(threads[i], &ret_val);
62              partial_sum = (long long *)ret_val;
63              sum += *partial_sum;
64              free(partial_sum); // 释放子线程中malloc的内存
65          }
66          clock_gettime(CLOCK_MONOTONIC, &end);
67          diff = (end.tv_sec - start.tv_sec) * 1000000000L + (end.tv_nsec - start.tv_nsec);
68          average += diff;
69          printf("[test %2d] Multi-thread (N=%d) took %7ld ns\n", j, N, diff);
70      }
71      average /= TEST_TIMES;
```

```
    printf("[average] Multi-thread (N=%d) took %7ld ns\n", N, average);
    free(arr);
    return 0;
}
```

```
zsh@kolp:~/VScodeproject/UCAS-2025-OS-Theory/os_hw_code/hw3$ ./3.1.2
[test  0] Multi-thread (N=2) took 1462930 ns
[test  1] Multi-thread (N=2) took 1499694 ns
[test  2] Multi-thread (N=2) took 1126436 ns
[test  3] Multi-thread (N=2) took 1228659 ns
[test  4] Multi-thread (N=2) took 1148210 ns
[test  5] Multi-thread (N=2) took 1213711 ns
[test  6] Multi-thread (N=2) took 1290024 ns
[test  7] Multi-thread (N=2) took 1048322 ns
[test  8] Multi-thread (N=2) took 1194764 ns
[test  9] Multi-thread (N=2) took 1451619 ns
[average] Multi-thread (N=2) took 1266436 ns
```

图 2: 双线程数组求和用时

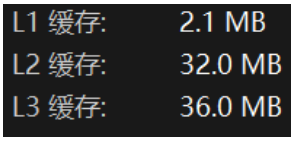

图 3: 运行电脑的 CPU Cache 大小

2) 程序源码如表 4 ，对 N=1 16 个线程重复实验了 10 次，图 4 即为各个线程数时 10 次实验的运行结果，发现结果波动较大，且重复运行程序后得到的平均结果也有明显差异（体现为不同线程的速度排名不稳定），怀疑 10 次实验次数太少，随机性对最终平均结果的影响仍然较大（注：该程度的随机性并不影响上面两次实验的结论）。于是改为重复实验 100 次，重复运行程序得到的平均结果基本一致，最终得到的平均结果如图 5 用简易柱状图表示，说明并不是 N 值越大，耗费时间就越短。因为当线程数超过了物理核心数时，核心的上下文切换开销以及创建/销毁开销会超过多线程并行带来的收益，导致耗时反而增加。我的电脑在 N=7 时表现最佳，说明电脑运行该程序时可用的核心数很可能是 7 个。

Listing 4: N 线程数组求和

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <pthread.h>
#define ARR_LEN 1000000
#define TEST_TIMES 10
#define MAX_N 16
typedef struct {
    // ... 略
} thread_arg_t;
void* partial_add(void* arg){
```

```
12      // ... 略
13  }
14  int main(int argc, char const *argv[])
15  {
16      long long sum = 0;
17      int *arr = (int*)malloc(sizeof(int) * ARR_LEN);
18      for (int i = 0; i < ARR_LEN; i++)
19          arr[i] = i + 1;
20      struct timespec start, end;
21      long diff = 0;
22      long time_table[MAX_N] = {0};
23      printf("num_threads  ");
24      for (int i = 1; i <= TEST_TIMES; i++)printf("[test%2d]    ", i);
25      printf("[average]\n");
26      for (int N = 1; N <= MAX_N; N++)
27      {
28          printf("   N=%2d      ", N);
29          thread_arg_t args[N];
30          int CHUNK_SIZE = (ARR_LEN / N);
31          for (int i = 0; i < N; i++)
32          {
33              args[i].thread_id = i;
34              args[i].arr = arr;
35              args[i].start_index = i * CHUNK_SIZE;
36              args[i].end_index = (i == N - 1) ? (ARR_LEN - 1) : ((i+1) * CHUNK_SIZE - 1);
37          }
38          for (int j = 0; j < TEST_TIMES; j++)
39          {
40              pthread_t threads[N];
41              clock_gettime(CLOCK_MONOTONIC, &start);
42              for (int i = 0; i < N; i++)
43              {
44                  pthread_create(&threads[i], NULL, partial_add, (void *)&args[i]);
45              }
46              long long *partial_sum;
47              for (int i = 0; i < N; i++)
48              {
49                  void *ret_val;
50                  pthread_join(threads[i], &ret_val);
51                  partial_sum = (long long *)ret_val;
52                  sum += *partial_sum;
53                  free(partial_sum); // 释放子线程中malloc的内存
54              }
55              clock_gettime(CLOCK_MONOTONIC, &end);
56              diff = (end.tv_sec - start.tv_sec) * 1000000000L + (end.tv_nsec - start.
                    tv_nsec);
57              printf("%8ld    ", diff);
58              time_table[N - 1] += diff;
59          }
```

```
        time_table[N - 1] /= TEST_TIMES;
        printf("%8ld", time_table[N - 1]);
        printf("\n");
    }
    printf("num_threads   average_time\n");
    for (int N = 1; N <= MAX_N; N++)
    {
        printf("   N=%d\t", N);
        printf("      %7ld ns ", time_table[N - 1]);
        int time_len = time_table[N - 1] / 50000;
        for (int j = 0; j < time_len; j++) printf("-");
        printf("\n");
    }
    return 0;
}
```

num_threads	[test 1]	[test 2]	[test 3]	[test 4]	[test 5]	[test 6]	[test 7]	[test 8]	[test 9]	[test10]	[average]
N= 1	2222616	2154523	1986320	2019766	2036709	2083261	1870988	1842791	2002209	1998557	2021774
N= 2	1200003	1270535	1066850	1076893	1048698	2216005	1327828	1284254	1367285	1307310	1316566
N= 3	1211268	1169929	784921	843429	764192	959950	867642	991660	1055957	1001431	965037
N= 4	919439	718680	785822	665286	895541	808122	633220	817317	684978	715065	764347
N= 5	917980	814992	728952	807861	947511	974064	794929	1019968	921274	820441	874797
N= 6	966485	808598	859988	911340	724872	743917	788160	1023397	685091	678163	819001
N= 7	730023	703129	1100847	657003	733185	641014	713509	777827	784865	841593	768299
N= 8	735307	975757	679732	815863	757604	748786	939250	792649	1235387	732369	841270
N= 9	751928	840977	943422	807940	760868	649538	813475	906461	1098218	786770	835959
N=10	669706	1102985	1134031	883300	963606	766316	1007239	1061660	879169	1006768	947478
N=11	981202	906690	896358	872885	938827	953374	918595	911926	1050610	1029391	945985
N=12	914471	1091499	1097453	997227	934718	924705	997020	1125770	1560714	1027280	1067085
N=13	987850	1070307	976818	1072316	979094	892461	955666	1169589	1333848	1073182	1051113
N=14	1049342	1142721	1002607	1130075	1202904	1111076	1137252	1237561	1007636	997478	1101865
N=15	1111930	1160047	1008542	1069796	1001666	1069612	1093058	1126278	1086368	992371	1071966
N=16	1319994	1057148	1141869	1208872	1046522	1099160	1118387	1095984	1182996	1296219	1156715

图 4: 不同 N 线程数组求和用时表格（重复 10 次）

```
num_threads   average_time
   N=1        2242081 ns ----------------------------------------------
   N=2        1239209 ns ------------------------
   N=3         849369 ns ----------------
   N=4         743023 ns --------------
   N=5         729367 ns --------------
   N=6         711560 ns --------------
   N=7         697296 ns -------------
   N=8         739739 ns --------------
   N=9         757675 ns ---------------
   N=10        807671 ns ----------------
   N=11        880078 ns -----------------
   N=12        942240 ns ------------------
   N=13        973947 ns -------------------
   N=14       1033827 ns --------------------
   N=15       1098653 ns ---------------------
   N=16       1146674 ns ----------------------
```

图 5: 不同 N 线程数组求和平均用时对比（重复 100 次）

(3)___USE_GNU 是一个 glibc 内部使用的宏，使用 gcc 编译时会报错，改为 _USE_SOURCE 或者使用 g++ 编译可以解决。

程序源码如表 5 ， 参数结构体中定义的 thread_id 派上了用场， 用于指定绑定的核心序

号，为防止绑定序号超出核心数，使用 unistd.h 库的 sysconf(_SC_NPROCESSORS_ONLN); 获取在线核心数, 以对绑定序号进行取模调整，同样重复实验 100 次取平均，其他部分逻辑与 (2) 2) 相同，不再重复放出。

运行结果如图 6，和不绑核时的结果图 7对比可知：1. 无论是否绑核，随着线程数的增加，运行时间变化趋势都是先减少再增加的。2. 绑核后，性能峰值点从 N=7 变为 N=5，最短运行时间由 697296 ns 降为 679,852 ns，可能的解释是绑核操作提高了 CPU 缓存的利用率：在不绑核时，操作系统调度器为了负载均衡，可能会做出线程迁移，从而导致该线程在原核心的 Cache 中已经加载的数据全部失效，需要新核心上重新从内存读取，从而浪费了时间；绑核则杜绝了线程迁移，每个核心的缓存都能被稳定利用，因此获得了更高的极限速度。

Listing 5: 绑核 N 线程数组求和

```c
#define _GNU_SOURCE
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <pthread.h>
#include <sched.h>
#include <unistd.h> // 包含 sysconf
#define ARR_LEN 1000000
#define TEST_TIMES 100
#define MAX_N 16
long num_cores; // 逻辑核心数
typedef struct {
    int thread_id;      // 线程ID
   // ...略
} thread_arg_t;

void* partial_add(void* arg){
    thread_arg_t *my_arg = (thread_arg_t *)arg;
    cpu_set_t cpuset;   //CPU核的位图
    CPU_ZERO(&cpuset);  //将位图清零
    CPU_SET(my_arg->thread_id % num_cores, &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
    // ...略
}

int main(int argc, char const *argv[])
{
    num_cores = sysconf(_SC_NPROCESSORS_ONLN);
    printf("这台电脑有 %ld 个逻辑核心可用。\n", num_cores);
    // ...略
}
```

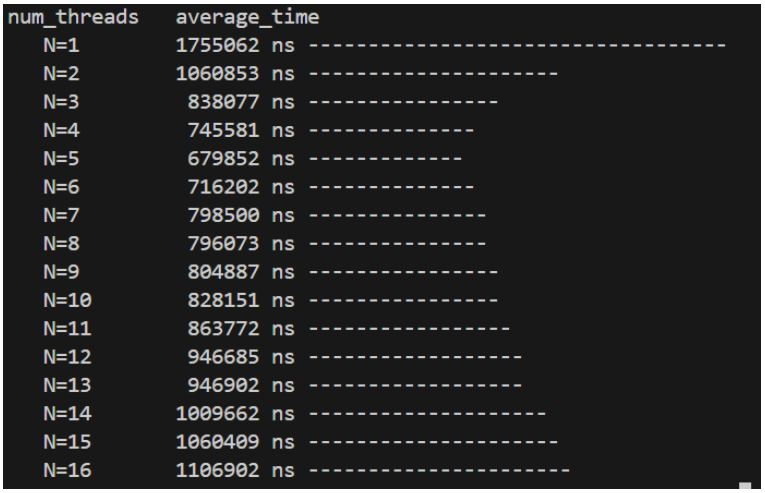

图 6: 绑核时不同 N 线程求和用时对比（重复 100 次）

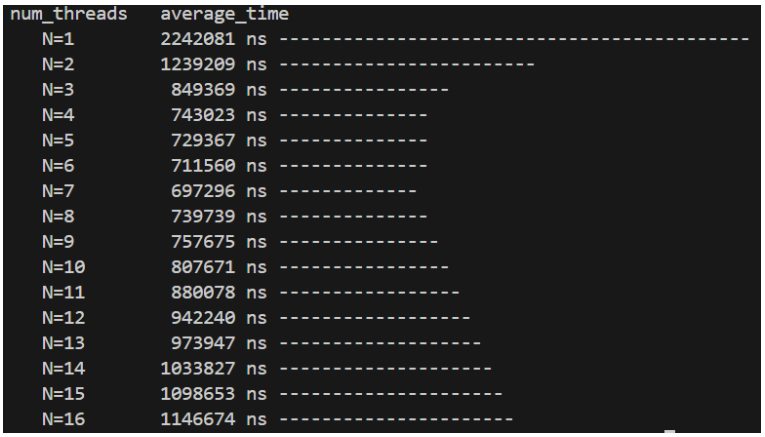

图 7: 不绑核时不同 N 线程求和用时对比（重复 100 次）

3.2 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：

写一个 C 程序，首先创建一个有 1 万个元素的整数型空数组，然后初始化该数组，数组的每一个元素为 [1,100000] 区间的一个随机整数。请分别使用以下两种方式找出该数组种的最大值：

1）仅使用进程，在数组初始化完成后，当前进程继续在该数组中查找最大值（算法不限），并打印结果。

2）使用 pthread 创建 N 个线程（N 值自行决定, 且 N>1），每个线程负责在 10000/N 个数组元素中查找最大值（自行决定各个线程对应的数组元素）。每个线程查找完成后，将查到的最大值写入一个全局数组中，最后由主进程在该全局数组中查找最大值，并打印结果。注意：主进程需要在所有线程完成查找后再开始在全局数组中查找。

请统计并比较上述两种查找最大值方法的耗时，两者是否有差异？是否符合你的预期？如果原始数组的大小不是 1 万，而是有 1000 万个元素，上述两种方法会有差异么？请结合你的测试结果进行阐述。

提交内容： 所写 C 程序，打印结果截图, 关键代码注释，分析说明等

解答： 仅使用进程查找最大值的源码如表 6 ，使用 N=4 线程查找最大值的源码如表 7 。运行结果如图 8 所示，四线程版本慢了将近 50 倍，这是因为原始数组只有 1 万个元素，是很小的任务，创建和等待线程所产生的开销远远超过了查找工作本身。将数组大小改为 1000 万后，运行结果如图 9 ，多线程版本快了近 3 倍，即在工作量变大后，线程管理的开销相对于庞大的工作量变得微不足道了，使用多线程得到的收益大于额外开销。

Listing 6: 单线程查找最大值

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <limits.h>
#define ARR_LEN 10000
int find_max(int *arr, int len){
    int max = INT_MIN;
    for (int i = 0; i < len; i++)
    {
        if(arr[i] > max) max = arr[i];
    }
    return max;
}
int main(int argc, char const *argv[])
{
    int *arr = (int *)malloc(sizeof(int) * ARR_LEN);
    for (int i = 0; i < ARR_LEN; i++)
        arr[i] = rand() % 100000 + 1;
    struct timespec start, end;
    long diff = 0;
    clock_gettime(CLOCK_MONOTONIC, &start);
    int max_num = find_max(arr, ARR_LEN);
    clock_gettime(CLOCK_MONOTONIC, &end);
    diff = (end.tv_sec - start.tv_sec) * 1000000000L + (end.tv_nsec - start.tv_nsec);
    printf("the max num is %d\n", max_num);
    printf("Signgle-thread find_max took %ld ns\n", diff);
    return 0;
}
```

Listing 7: 四线程查找最大值

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <limits.h>
#include <pthread.h>
#define ARR_LEN 10000
#define N 4
#define CHUNK_SIZE (ARR_LEN / N)
int res_arr[N];
typedef struct {
    int* arr;           // 数组指针
```

```
12      int start_index;    // 起始索引
13      int end_index;      // 结束索引
14      int *res;           // 写入全局数组的位置
15  } thread_arg_t;
16  void* find_max(void *arg)
17  {
18      thread_arg_t *my_arg = (thread_arg_t *)arg;
19      int start_index = my_arg->start_index, end_index = my_arg->end_index;
20      int *arr = my_arg->arr;
21      int max = INT_MIN;
22      for (int i = start_index; i <= end_index; i++)
23      {
24          if(arr[i] > max) max = arr[i];
25      }
26      *my_arg->res = max;
27      return NULL;
28  }
29  int main(int argc, char const *argv[])
30  {
31      int *arr = (int *)malloc(sizeof(int) * ARR_LEN);
32      int max = INT_MIN;
33      for (int i = 0; i < ARR_LEN; i++)
34          arr[i] = rand() % 100000 + 1;
35      struct timespec start, end;
36      long diff = 0;
37      clock_gettime(CLOCK_MONOTONIC, &start);
38      thread_arg_t args[N];
39      pthread_t threads[N];
40      for (int i = 0; i < N; i++)
41      {
42          args[i].arr = arr;
43          args[i].start_index = i * CHUNK_SIZE;
44          args[i].end_index = (i == N - 1) ? (ARR_LEN - 1) : ((i+1) * CHUNK_SIZE - 1);
45          args[i].res = res_arr + i;
46          pthread_create(&threads[i], NULL, find_max, (void *)&args[i]);
47      }
48      for (int i = 0; i < N; i++)
49      {
50          void *ret_val;
51          pthread_join(threads[i], &ret_val);
52          if(res_arr[i] > max) max = res_arr[i];
53      }
54      clock_gettime(CLOCK_MONOTONIC, &end);
55      diff = (end.tv_sec - start.tv_sec) * 1000000000L + (end.tv_nsec - start.tv_nsec);
56      printf("the max num is %d\n", max);
57      printf("Multi-thread (N=%d) find_max took %ld ns\n", N, diff);
58      return 0;
59  }
```

```
zsh@kolp:~/VScodeproject/UCAS-2025-OS-Theory/os_hw_code/hw3$ ./3.2.1 && ./3.2.2
the max num is 99973
Signgle-thread find_max took 12021 ns
the max num is 99973
Multi-thread (N=4) find_max took 538995 ns
```

图 8: 单/四线程查找 1 万元素的最大值运行结果

```
zsh@kolp:~/VScodeproject/UCAS-2025-OS-Theory/os_hw_code/hw3$ ./3.2.1 && ./3.2.2
the max num is 100000
Signgle-thread find_max took 12032786 ns
the max num is 100000
Multi-thread (N=4) find_max took 4209701 ns
```

图 9: 单/四线程查找 1000 万元素的最大值运行结果

注：本作业使用了 Gemini2.5 Pro 模型，用于查阅并分析 API 用法。