|  |  |  |  |
| --- | --- | --- | --- |
| **《操作系统原理》实验报告** | | | |
| 实验名称 | 多线程排序 | 实验序号 | 3 |
| 实验日期 | 2023/4/14 | 实验人 |  |
| 一、实验题目  编写一个多线程排序程序，其工作方式如下：  一个整数列表被分成两个大小相等的较小列表。两个单独的线程（我们将其称为 sorting threads）使用您选择的排序算法对每个子列表进行排序。这两个子列表随后由第三个线程——a merging thread——它将两个子列表合并为一个排序列表。 | | | |
| 二、相关原理与知识   1. pthread   Pthread是由POSIX标准(IEEE 1003.1c)为线程创建和同步定义的API。这是线程行为的规范，而不是实现。  所有Pthread程序都需要包括pthread.h头文件。语句pthread\_t tid 声明了所创建线程的标识符。每个线程都有一组属性， 包括栈大小和调度信息。pthread\_attr\_t atr表示线程的属性，通过函数调用pthread\_attr\_init(&atr)来设置这些属性。若没有明确地设置任何属性,则使用提供的默认属性。通过函数调用pthread\_create()来创建一个独立线程。除了传递线程标识符和线程属性外，还要传递函数名称，以便新线程可以开始执行。最后传递特定函数的参数。   1. 函数pthread\_create用来创建一个独立线程，并让它开始运行指定的线程函数。   **头文件**：#include <pthread.h>  **函数定义**：int pthread\_create(pthread\_t \*thread, const pthread\_attr\_t \*attr, void \*(\*start\_routine)(void \*), void \*arg);  **描述**：pthread\_create函数是 POSIX 线程库（Pthreads）中用于创建新线程的函数。pthreads 是一种用于 C 语言编写的多线程程序的线程库，遵循 POSIX 标准。  **参数**：  pthread\_t \*thread：线程标志符，此处输入指定线程的指针。  const pthread\_attr\_t \*attr：一个设置线程属性的指针，若为’NULL’则使用默认属性。  void \*(\*start\_routine)(void \*)：一个指向线程函数的指针。线程函数接受一个void\*类型的参数，并返回一个void\*类型的值。线程函数是新线程开始执行时调用的函数。  void \*arg：一个指向传递给线程函数的参数的指针。可以是任意类型的指针，但需要转换为void\*类型。  **返回值**：如果成功创建新线程，则返回0；否则返回一个非零的错误码。   1. 函数pthread\_join用来等待一个线程的结束,线程间同步的操作。   **头文件** ： #include <pthread.h>  **函数定义**： int pthread\_join(pthread\_t thread, void \*\*retval);  **描述** ：pthread\_join()函数，以阻塞的方式等待thread指定的线程结束。当函数返回时，被等待线程的资源被收回。如果线程已经结束，那么该函数会立即返回。并且thread指定的线程必须是joinable的。  **参数** ：  thread: 线程标识符，即线程ID，标识唯一线程。  retval: 用户定义的指针，用来存储被等待线程的返回值。如果不关心返回值，可以设置为NULL。  **返回值**：0代表成功。 失败，返回的则是错误号。   1. 排序算法   排序算法数量很多，此处介绍三种常见的排序算法，即冒泡排序、归并排序、快速排序。  冒泡排序：冒泡排序是一种简单的排序算法，它重复地遍历列表，比较每对相邻的元素并交换它们（如果它们的顺序错误）。每次遍历将下一个最大的元素放到正确的位置。这个过程一直持续到整个列表完全有序。  归并排序：归并排序是一种采用分治策略的排序算法。首先将列表递归地分成两半，对每一半进行排序，然后将排序后的两半合并成一个有序列表。归并排序是稳定的，但需要额外的存储空间。  快速排序：快速排序是一种高效的排序算法，采用分治策略。它的基本思想是：在列表中选择一个元素（称为"基准"）并将列表分为两部分，一部分包含比基准小的元素，另一部分包含比基准大的元素。然后，对这两部分递归地应用快速排序。最后，将两个已排序的部分连接在一起。  鉴于对上述排序算法的熟练度和降低程序运行时间复杂度的要求，代码中使用快速排序作为排序算法。 | | | |
| 三、实验过程  根据实验的要求，整体的实现思路大致可以分为三个步骤，即变量的初始化、线程一和线程二的创建与排序的实现、线程一和线程二的合并。  简单概括，即定义四个全局变量数组，origin\_array、result\_array、array1、array2，在初始输入origin\_array中的元素后，将其中的元素等量分到array1和array2两个数组中；分别创建两个线程，分别对array1和array2进行排序；最后将两个线程合并成一个线程，并在这个线程中对array1和array2中的结果再次排序，并将最终结果存放到result\_array数组中。  以下是具体实现思路与代码的解释。   1. 变量的初始化   此处主要定义结构体ThreadArgs和四个全局变量数组，origin\_array、result\_array、array1、array2，并根据用户提示输入相应的数据，完成对于准备排序的数组、创建线程所需的结构体的初始化。  首先全局定义结构体和数组：   |  | | --- | | typedef struct  {  int \*array;  int size;  int start;  int end;  }ThreadArgs;    int \*array1;  int \*array2;  int \*origin\_array;  int \*result\_array; |   由于线程创建的需要，此处定义了结构体。对于该结构体，array为存储数值的数组；size为数组中的元素；start和end供排序使用，因为在使用快速排序时需要使用递归算法，每次递归时的需要给出不同的start和end，以方便排序，故此处在结构体中给出定义。  对于数组，array1为线程1使用的数组；array2为线程2使用的数组；origin\_array为初始数组；result\_array为存储排序结果使用的数组。  之后在main函数中首先完成对于变量的初始化：   |  | | --- | | int size;  printf("Enter the number of all elements: **\n**");  scanf("%d", &size);    array1 = (int \*)malloc(size / 2 \* sizeof(int));  array2 = (int \*)malloc((size - size / 2) \* sizeof(int));  origin\_array = (int \*)malloc(size \* sizeof(int));  result\_array = (int \*)malloc(size \* sizeof(int));    printf("Enter the elements: **\n**");  for(int i = 0; i < size; i ++ )  scanf("%d", &origin\_array[i]);  for(int i = 0; i < size / 2; i ++ )  array1[i] = origin\_array[i];  for(int i = 0; i < size - size / 2; i ++ )  array2[i] = origin\_array[i + size / 2];    ThreadArgs args1 = {array1, size / 2, 0, size / 2 - 1};  ThreadArgs args2 = {array2, size - size / 2, 0, size - size / 2 - 1};  ThreadArgs merge\_args = {result\_array, size, 0, size - 1}; |   首先给出交互提示，让用户输入size，并根据size为origin\_array、result\_array、array1、array2分配空间，之后再次给出交互提示，让用户输入待排序的数字并存储到origin\_array中；之后对数字进行分配，前一半放在array1中，后一半放在array2中，当size为奇数时，array2中的元素数量比array1多1个。最后创建线程所需的参数结构体。   1. 线程一和线程二的创建和排序的实现   此处主要实现两个线程的创建。在C语言中，主要使用POSIX 线程库（Pthreads）中的pthread\_create函数。  上面介绍过，此函数需要线程ID、执行的函数和结构体参数。我们已经定义了结构体参数，因此我们首先对线程ID进行定义：   |  | | --- | | pthread\_t sorting\_thread1, sorting\_thread2, merging\_thread; |   其中sorting\_thread1为array1排序所用的线程ID；sorting\_thread2为array2排序所用的线程ID；merging\_thread为执行线程合并的线程的ID。  对于此处执行排序的函数，其定义如下：   |  | | --- | | void \*quick\_sort(void \*arguments)  {  ThreadArgs \*args = (ThreadArgs \*)arguments;  int \*array = args->array;  int start = args->start;  int end = args->end;    if (start < end)  {  int pivot = array[end];  int i = start - 1;    for (int j = start; j <= end - 1; j++)  {  if (array[j] < pivot)  {  i++;  int temp = array[i];  array[i] = array[j];  array[j] = temp;  }  }    int temp = array[i + 1];  array[i + 1] = array[end];  array[end] = temp;    ThreadArgs args\_left = {array, 0, start, i};  ThreadArgs args\_right = {array, 0, i + 2, end};    quick\_sort(&args\_left);  quick\_sort(&args\_right);  }    return NULL;  } |   此为线程1和线程2进行快速排序使用的函数。对于创建线程时所用的函数，其应接受一个void\*类型的参数，并返回一个void\*类型的值。  与常规快排不同的是，线程函数首先从arguments参数中提取array、start和end；之后是正常的快排步骤；在最后进行递归之前，还要先创建ThreadArgs结构体参数，强制转换为(void\*)类型后才可作为参数参与递归。  以上，线程ID、执行的函数和结构体参数三个pthread\_create函数所需参数均创建完成，此时可以进行线程的创建和排序的执行。  具体代码如下：   |  | | --- | | pthread\_create(&sorting\_thread1, NULL, quick\_sort, (void \*)&args1);  pthread\_create(&sorting\_thread2, NULL, quick\_sort, (void \*)&args2); |   进程执行完成以后，由于我们之后还会进行线程的合并，而线程合并对应的新线程的开始，应当在两个排序线程完成以后，因此我们需要等待线程结束。  即：   |  | | --- | | pthread\_join(sorting\_thread1, NULL);  pthread\_join(sorting\_thread2, NULL); |   当两个线程均结束以后，array1和array2两个子数组排序完成。此时我们可以执行输出操作，观察两个子进程对于array1和array2的排序是否正确。  对应代码：   |  | | --- | | printf("The sorted result of thread 1:**\n**");  for(int i = 0; i < args1.size; i ++ )  printf("%d ", args1.array[i]);  printf("**\n**");  printf("The sorted result of thread 2:**\n**");  for(int i = 0; i < args2.size; i ++ )  printf("%d ", args2.array[i]);  printf("**\n**"); |   到此线程一和线程二的创建和排序的实现步骤完成。   1. 线程一和线程二的合并   线程一和二的合并同样依托于新线程的创建来完成。根据步骤2，使用pthread\_create函数创建线程需要线程ID、执行的函数和结构体参数。在该步骤之前我们已经定义了结构体参数和线程ID：merging\_thread，因此我们需要定义执行函数。  其具体定义如下：   |  | | --- | | void \*merge\_arrays(void \*arguments) {  ThreadArgs \*args = (ThreadArgs \*)arguments;  int \*result\_array = args -> array;  int size = args -> size;    int index1 = 0;  int index2 = 0;  int index\_merged = 0;    while (index1 < size / 2 && index2 < (size - size / 2)) {  if (array1[index1] < array2[index2]) {  result\_array[index\_merged ++ ] = array1[index1 ++ ];  } else {  result\_array[index\_merged ++ ] = array2[index2 ++ ];  }  }    while (index1 < size / 2) {  result\_array[index\_merged ++ ] = array1[index1 ++ ];  }    while (index2 < (size - size / 2)) {  result\_array[index\_merged ++ ] = array2[index2 ++ ];  }    return NULL;  } |   大致思路即为依次对比array1和array2中的元素，较小的先放入result\_array数组中；当其中某一个数组全部被放入result\_array数组以后，将另一个数组的剩余元素直接放在result\_array数组的最后即可。  此时可以实现线程一和线程二合并的线程操作，代码如下：   |  | | --- | | pthread\_create(&merging\_thread, NULL, merge\_arrays, (void \*)&merge\_args);  pthread\_join(merging\_thread, NULL); |   此处针对主线程也需要等待其停止，因为接下来我们要进行输出验证。  然后针对完全排序后的result\_array数组，我们可以进行输出操作观察排序是否正确。   |  | | --- | | printf("The sorted result of merged thread:**\n**");  for(int i = 0; i < size; i ++ )  printf("%d ", result\_array[i]);  printf("**\n**"); |   以上即线程一和线程二的合并操作。 | | | |
| 四、实验结果与分析    当输入为个数为偶数时，可以看到每步的排序均正确    当输入数字个数为奇数时，也可观察到各步的排序均正确。 | | | |
| 五、问题总结   1. 合并两个子数组时，忘记将未完全填入result\_array数组的子数组剩余值填入，导致结果出错   解决：将未完全填入result\_array数组的子数组剩余值填入   1. 未在主线程中等待合并线程结束，导致主线程先跑完，打印的排序结果全为0   解决方式：创建合并线程后，主线程也需等待其结束   1. 在使用银河麒麟系统编译c文件时，报错linux undefined reference to pthread\_create。经过查阅资料发现这是由于pthread不是linux下的默认的库，也就是在链接的时候，无法找到phread库中该函数的入口地址，于是链接会失败。   解决：在编译指令结尾加上 -lpthread即可，即使用gcc thread.c -o thread -lpthread指令进行编译即可。 | | | |
| 六、源代码  #include <stdio.h>  #include <stdlib.h>  #include <pthread.h>  typedef struct  {  int \*array;  int size;  int start;  int end;  }ThreadArgs;  int \*array1;  int \*array2;  int \*origin\_array;  int \*result\_array;  void \*quick\_sort(void \*arguments);  void \*merge\_arrays(void \*arguments);  int main()  {  int size;  printf("Enter the number of all elements: \n");  scanf("%d", &size);  array1 = (int \*)malloc(size / 2 \* sizeof(int));  array2 = (int \*)malloc((size - size / 2) \* sizeof(int));  origin\_array = (int \*)malloc(size \* sizeof(int));  result\_array = (int \*)malloc(size \* sizeof(int));  printf("Enter the elements: \n");  for(int i = 0; i < size; i ++ )  scanf("%d", &origin\_array[i]);  for(int i = 0; i < size / 2; i ++ )  array1[i] = origin\_array[i];  for(int i = 0; i < size - size / 2; i ++ )  array2[i] = origin\_array[i + size / 2];  ThreadArgs args1 = {array1, size / 2, 0, size / 2 - 1};  ThreadArgs args2 = {array2, size - size / 2, 0, size - size / 2 - 1};  ThreadArgs merge\_args = {result\_array, size, 0, size - 1};  pthread\_t sorting\_thread1, sorting\_thread2, merging\_thread;  pthread\_create(&sorting\_thread1, NULL, quick\_sort, (void \*)&args1);  pthread\_create(&sorting\_thread2, NULL, quick\_sort, (void \*)&args2);  pthread\_join(sorting\_thread1, NULL);  pthread\_join(sorting\_thread2, NULL);  printf("The sorted result of thread 1:\n");  for(int i = 0; i < args1.size; i ++ )  printf("%d ", args1.array[i]);  printf("\n");  printf("The sorted result of thread 2:\n");  for(int i = 0; i < args2.size; i ++ )  printf("%d ", args2.array[i]);  printf("\n");  pthread\_create(&merging\_thread, NULL, merge\_arrays, (void \*)&merge\_args);  pthread\_join(merging\_thread, NULL);  printf("The sorted result of merged thread:\n");  for(int i = 0; i < size; i ++ )  printf("%d ", result\_array[i]);  printf("\n");  return 0;  }  void \*quick\_sort(void \*arguments)  {  ThreadArgs \*args = (ThreadArgs \*)arguments;  int \*array = args -> array;  int start = args -> start;  int end = args -> end;  if(start < end)  {  int pivot = array[end];  int i = start - 1;  for(int j = start; j <= end - 1; j ++ )  {  if(array[j] < pivot)  {  i ++ ;  int temp = array[i];  array[i] = array[j];  array[j] = temp;  }  }  int temp = array[i + 1];  array[i + 1] = array[end];  array[end] = temp;  ThreadArgs args\_left = {array, start, i ,0};  ThreadArgs args\_right = {array, i + 2, end, 0};  quick\_sort(&args\_left);  quick\_sort(&args\_right);  }  return NULL;  }  void \*merge\_arrays(void \*arguments) {  ThreadArgs \*args = (ThreadArgs \*)arguments;  int \*result\_array = args -> array;  int size = args -> size;  int index1 = 0;  int index2 = 0;  int index\_merged = 0;  while (index1 < size / 2 && index2 < (size - size / 2)) {  if (array1[index1] < array2[index2]) {  result\_array[index\_merged ++ ] = array1[index1 ++ ];  } else {  result\_array[index\_merged ++ ] = array2[index2 ++ ];  }  }  while (index1 < size / 2) {  result\_array[index\_merged ++ ] = array1[index1 ++ ];  }  while (index2 < (size - size / 2)) {  result\_array[index\_merged ++ ] = array2[index2 ++ ];  }  return NULL;  } | | | |