《操作系统》

实 验 报 告

学 生 姓 名　夏雨轩
学　　号　21009201006
班　　级　2118021

20 22 — 20 23 学年 第 2 学期

《操作系统》实验报告			
实验名称	多线程编程	实验序号	3
实验日期	2022.4.4	实验人	夏雨轩

一、实验题目

Project 2-Multithreaded Sorting Application

Write a multithreaded sorting program that works as follows: A list of integers is divided into two smaller lists of equal size. Two separate threads (which we will term sorting threads) sort each sublist using a sorting algorithm of your choice. The two sublists are then merged by a third thread --a merging thread which merges the two sublists into a single sorted list.

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。两个单独的线程使用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将其合并为一个排序列表。

二、相关原理与知识

（完成实验所用到的相关原理与知识）

1. 对乱序数字的排序方法，采用归并排序和选择排序算法。

归并排序，是创建在归并操作上的一种有效的排序算法。算法是采用分治法（Divide and Conquer）的一个非常典型的应用，且各层分治递归可以同时进行。归并排序思路简单，速度仅次于快速排序，为稳定排序算法，一般用于对总体无序，但是各子项相对有序的数列。归并排序是用分治思想，分治模式在每一层递归上有三个步骤：（1）**分解（Divide）**：将 n 个元素分成个含 n/2 个元素的子序列。（2）**解决（Conquer）**：用合并排序法对两个子序列递归的排序。（3）**合并（Combine）**：合并两个已排序的子序列已得到排序结果。

归并排序主要通过迭代法和递归法完成。

选择排序法是一种简单直观的排序算法。首先在未排序序列中找到最小（大）元素，存放到排序序列的起始位置，然后，再从剩余未排序元素中继续寻找最小（大）元素，然后放到已排序序列的末尾。以此类推，直到所有元素均排序完毕。选择排序的思想其实和冒泡排序有点类似，都是在一次排序后把最小的元素放到最前面，或者将最大值放在最后面。但是过程不同，冒泡排序是通过相邻的比较和交换。而选择排序是通过对整体的选择，每一趟从前往后查找出无序区最小值，将最小值交换至无序区最前面的位置。

选择排序法还可以优化改进为二元选择排序和堆排序。

2. 多线程并发

多线程并发指的是在同一个进程中执行多个线程。

线程是轻量级的进程，每个线程可以独立的运行不同的指令序列，但是线程不独立的拥有资源，依赖于创建它的进程而存在。也就是说，同一进程中的多个线程共享相同的地址空间，可以访问进程中的大部分数据，指针和引用可以在线程间进行传递。这样，同一进程内的多个线程能够很方便的进行数据共享以及通信，也就比进程更适用于并发操作。

但是由于缺少操作系统提供的保护机制，在多线程共享数据及通信时，就需要程序员

做更多的工作以保证对共享数据段的操作是以预想的操作顺序进行的，并且要极力的避免死锁(deadlock)。

3. pthread 库的概念与使用

概念：POSIX Threads 简称 Pthread，是线程的 POSIX 标准，被定义在 POSIX.1c, Threads extensions (IEEE Std1003.1c-1995)标准里，该标准定义了一套 C 程序语言的类型、函数和常量，定义在 pthread.h 头文件和一个线程库里，内容包括线程管理、互斥锁、条件变量、读写锁和屏障。

本次实验可能使用到的函数：

（1）创建线程：pthread_create

函数原型：int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void *(*start_routine)(void *), void *arg);

参数：thread ：线程的句柄。当一个新的线程调用成功之后，就会通过这个参数将线程的句柄返回给调用者，以便对这个线程进行管理。

attr ：线程的属性，大多数情况下默认为 NULL。

start_routine：线程的入口函数。arg:入口函数的参数。

返回值：返回值为 0，代表线程创建成功，其他值表示创建失败。

（2）结束线程：pthread_join

函数原型：int pthread_join(pthread_t thread, void **retval);

说明：函数 pthread_join 用来等待一个线程的结束,线程间同步的操作。

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。

（3）线程的合并：

线程的合并是一种主动回收线程资源的方案。当一个进程或线程调用了针对其它线程的 pthread_join()接口，就是线程合并了。这个接口会阻塞调用进程或线程，直到被合并的线程结束为止。当被合并线程结束，pthread_join()接口就会回收这个线程的资源，并将这个线程的返回值返回给合并者。

（4）线程分离： pthread_detach

函数原型：int pthread_detach(pthread_t thread);

功能说明：将线程 ID 为 thread 的线程分离出去，所谓分离出去就是指主线程不需要再通过 pthread_join 等方式等待该线程的结束并回收其线程控制块（TCB）的资源，被分离的线程结束后由操作系统负责其资源的回收。

返回值说明：成功的时候返回 0，失败时返回错误码。

（5）线程 ID 比较：pthread_equal

函数原型：int pthread_equal(pthread_t t1, pthread_t t2);

功能说明：比较两个线程 ID 是否相等，在 Linux 系统中 pthread_t 都设计为 unsigned long 类型，所以可以直接用 == 判别是否相等，但是如果某些系统设计为结构体类型，那么就可以通过 pthread_equal 函数判别是否相等了。

返回值说明：成功的时候返回 0，失败时返回错误码。

二、实验过程

用 vi 编辑器编写代码后，进行编译并手动连接 phtread 库，用于程序中的调用

```
#include<stdio.h>
#include<string.h>
#include<stdlib.h>
#include<unistd.h>
#include<pthread.h>

int getIntArrLen(int *num){
  int i = 0;
  while(num[i]!='\0'){
i++;
}
return i;
}

void*thread_func(void*arg){
   int*num=(int*)arg;
int len=getIntArrLen(num);
    int i,j;

for(i=0;i<len-1;i++){
   for(j=0;j<len-1-i;j++){
       if(num[j]>num[j+1]){
           int temp=num[j];
          num[j]=num[j+1];
           num[j+1]=temp;
       }
   }
}
return NULL;
}
```

```
int main(int argc,char*argv[]){
  int num[40]={0};
  int *num1=malloc(sizeof(int)*40+1);
  int *num2=malloc(sizeof(int)*40/2+1);
   int i;
int k=0;
char ch;
do{
scanf("%d",&num[k++]);
scanf("%c",&ch);
}while(ch !='\n');

  for (i=0;i<k/2;i++){
     num1[i]=num[i];
  }
   int j=i;
i=0;
   for(;j<k;j++){
     num2[i++]=num[j];
   }

   pthread_t tid1,tid2,tid3;

  pthread_create(&tid1,NULL,(void*)thread_func,(void*)num1);
   pthread_create(&tid2,NULL,(void*)thread_func,(void*)num2);

   char*res=NULL;
   pthread_join(tid1,(void*)&res);
pthread_join(tid2,(void*)&res);

   for(i=0;i<getIntArrLen(num1);i++){
     printf("%d?",?num1[i]);
  }
printf("\n");

   for(i=0;i<getIntArrLen(num2);i++){
      printf("%d?",?num2[i]);
   }
printf("\n");

  int len=getIntArrLen(num1);
   for(i=0;i<getIntArrLen(num2);i++){
    num1[len]=num2[i];
     len++;
   }
   pthread_create(&tid3,NULL,(void*)thread_func,(void*)num1);
   pthread_join(tid3,(void*)&res);
```

```
[windamoon@cent 3]$ gcc -g -o multhread multithreaded.c -lpthread
[windamoon@cent 3]$ ./multhread

 1 2 5 3
 1 2
 3 5
 1 2 3 5
```

用 gcc -g -o multhread multithreded.c 指令对程序进行编译，编译成功后用命令./multhread 开始运行该程序并且输出第一个线程处理的前半部分的有序数列,第二个线程处理的后半部分的有序数列以及输出第三个线程合的最后完整的排序好的有序数列。

四、实验结果与分析

```
[windamoon@cent 3]$ ./multhread
1 2 5 3
1 2
3 5
1 2 3 5
[windamoon@cent 3]$ ./multhread
7 12 19 3 18 4 2 6 15 8
3 7 12 18 19
2 4 6 8 15
2 3 4 6 7 8 12 15 18 19
[windamoon@cent 3]$ ./multhread
1 4 2 6 8 15 7 16 10
1 2 4 6
7 8 10 15 16
1 2 4 6 7 8 10 15 16
```

1.输入若干乱序的数字，利用归并和选择排序算法

2.第一行输出原数组前半部分的有序数列

3.第二行输出原数组后半部分的有序数列

4.最后输出合并后的整体的有序数列

五、总结

1. 忘记等待两个子进程都结束进程再进行合并。

在多进程编程的时候，需要等多个子线程同时结束以后，然后再进行后续的流程。首先可以调用线程的.join()方法，卡住主线程的进行，直到两个子线程运行完毕再能让主线程继续运行后面的代码。因为线程是共享内存的，所以它们可以直接修改主线程传入的列表。其次还可以使用 multiprocessing.dummy 里面的 pool 来实现更简单的多线程。

2.对线程和进程的区别问题的思考

一个程序至少有一个进程，一个进程至少有一个线程。线程的划分尺度小于进程，使得多进程程序的并发性高。另外，进程在执行过程中拥有独立的内存单元，而多个线程共享内存，从而极大地提高了程序的运行效率。线程在执行过程中与进程还是有区别的。每个独立的线程有一个程序运行的入口、顺序执行序列和程序的出口。但是线程不能够独立执行，必须依存在应用程序中，由应用程序提供多个线程执行控制。

从逻辑角度来看，多线程的意义在于一个应用程序中，有多个执行部分可以同时执行。但操作系统并没有将多个线程看做多个独立的应用，来实现进程的调度和管理以及资源分配。这就是进程和线程的重要区别。

3.多线程的目的与意义

多线程的意义：其实任何一个程序的执行都需要获得 cpu 的执行权，是由 cpu 来去决定到底是由哪个程序来去执行，那么多线程的存在其实就是“最大限度的利用 cpu 资源”，当某一个线程的处理不需要占用 cpu 而和 I/O 打交道的时候，这就让需要占用 cpu 资源的其他线程有更多的机会获得 cpu 资源。

多线程的目的：使用多线程，可以帮助我们编写出 cpu 最大利用率的高效程序，使得空闲时间降到最低，这个程序编程语言的网络互联环境是至关重要的，因为空闲时间是公共的。例如，网络的传输效率远远低于计算机的处理速度，而本地文件系统资源的读写速度也远远低于 cpu 的处理能力。而多线程的使用使得我们能够充分利用这些空闲时间来进行程序的执行，极大地提高了计算机的执行效率。

4.在多线程协作执行时产生的问题与思考

第一点就是在进行多线程编程时一定注意考虑同步的问题，因为多数情况下我们创建多线程的目的是让他们协同工作，如果不进行同步，可能会出现问题。

第二点，死锁的问题。在多个线程访问多个临界资源时，处理不当会发生死锁。如果遇到编译通过，运行时卡住了，有可能是发生死锁了，可以先思考一下是那些线程会访问多个临界资源，这样查找问题会快一些。

第三点，临界资源的处理，多线程出现问题，很大原因是多个线程访问临界资源时的问题，一种处理方式是将对临界资源的访问与处理全部放到一个线程中，用这个线程服务其他线程的请求，这样只有一个线程访问临界资源就会解决很多问题。

第四点，线程池，在处理大量短任务时，我们可以先创建好一个线程池，线程池中的线程不断从任务队列中取任务执行，这样就不用大量创建线程与销毁线程了。

六、源代码

```
1.  #include <stdio.h>
2.  #include <string.h>
3.  #include <stdlib.h>
4.  #include <unistd.h>
5.  #include <pthread.h>
6.
7.
8.  int getIntArrLen(int *num){
9.          int i = 0;
```

```
10.         while(num[i]  !=  '\0'){
11.                 i++;
12.         }
13.         return  i;
14. }
15.
16. void*  thread_func(void*  arg)  {
17.      int  *num  =  (int  *)arg;
18.         int  len  =  getIntArrLen(num);
19.     int  i,  j;
20.
21.         for  (i  =  0;  i  <  len  -  1;  i++){
22.              for  (j  =  0;  j  <  len  -  1  -  i;  j++){
23.                  if  (num[j]  >  num[j  +  1]){
24.                      int  temp  =  num[j];
25.                      num[j]  =  num[j  +  1];
26.                      num[j  +  1]  =  temp;
27.                  }
28.              }
29.       }
30.       return  NULL;
31. }
32.
33. int  main(int  argc,  char  *argv[]){
34.      int  num[40]  =  {0}  ;
35.      int  *num1  =  malloc(sizeof(int)  *  40  +  1);
36.      int  *num2  =  malloc(sizeof(int)  *  40  /  2  +  1);
37.      int  i;
38.         int  k  =  0;
39.         char  ch;
40.         do{
41.                scanf("%d",&num[k++]);
42.                 scanf("%c",&ch);
43.         }while(ch  !=  '\n');
44.
45.      for  (i  =  0;  i  <  k  /  2;  i++){
46.          num1[i]  =  num[i];
47.      }
48.      int  j  =  i;
49.        i  =  0;
50.      for  (;  j  <  k;  j++){
51.          num2[i++]  =  num[j];
```

```
52.         }
53.
54.         pthread_t  tid1,  tid2,  tid3;
55.
56.         pthread_create(&tid1,  NULL,  (void  *)thread_func,  (void  *)num1);
57.         pthread_create(&tid2,  NULL,  (void  *)thread_func,  (void  *)num2);
58.
59.         char*  res  =  NULL;
60.         pthread_join(tid1,  (void  *)&res);
61.            pthread_join(tid2,  (void  *)&res);
62.
63.         for  (i  =  0;  i  <  getIntArrLen(num1);  i++){
64.              printf("%d  ",  num1[i]);
65.         }
66.            printf("\n");
67.
68.         for  (i  =  0;  i  <  getIntArrLen(num2);  i++){
69.              printf("%d  ",  num2[i]);
70.         }
71.            printf("\n");
72.
73.         int  len  =  getIntArrLen(num1);
74.         for  (i  =  0;  i  <  getIntArrLen(num2);  i++){
75.              num1[len]  =  num2[i];
76.              len++;
77.         }
78.         pthread_create(&tid3,  NULL,  (void*)thread_func,  (void  *)num1);
79.         pthread_join(tid3,  (void  *)&res);
80.
81.
82.         for  (i  =  0;  i  <  getIntArrLen(num1);  i++){
83.               printf("%d  ",  num1[i]);
84.         }
85.         printf("\n");
86.
87.            free(num1);
88.            free(num2);
89.
90.         return  0;
91.  }
```