《操作系统综合实验》

实验报告

03 多线程编程

姓 名　王 栗 政
班 级　2018039
学 号　20179100018
教 师　冯鹏斌

实验日期　2022.4.29

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. Because global data are shared cross all threads, perhaps the easiest way
to set up the data is to create a global array. Each sorting thread will work on one half of this array. A second global array of the same size as the unsorted integer array will also be established. The merging thread will then merge the two sublists into this second array. Graphically, this program is structured according to Figure 4.20.
This programming project will require passing parameters to each of the sorting threads. In particular, it will be necessary to identify the starting index from which each thread is to begin sorting. Refer to the instructions in Project 1 for details on passing parameters to a thread. The parent thread will output the sorted array once all sorting threads have exited.

二、相关原理与知识

1.线程

线程（英语：thread）是操作系统能够进行运算调度的最小单位。大部分情况下，它被包含在进程之中，是进程中的实际运作单位。一条线程指的是进程中一个单一顺序的控制流，一个进程中可以并发多个线程，每条线程并行执行不同的任务。在Unix System V及SunOS中也被称为轻量进程（lightweight processes），但轻量进程更多指内核线程（kernel thread），而把用户线程（user thread）称为线程。

线程是独立调度和分派的基本单位。线程可以为操作系统内核调度的内核线程，如Win32线程；由用户进程自行调度的用户线程，如Linux平台的POSIX Thread；或者由内核与用户进程，如Windows 7的线程，进行混合调度。

一个进程可以有很多线程来处理，每条线程并行执行不同的任务。如果进程要完成的任务很多，这样需很多线程，也要调用很多核心，在多核或多CPU，或支持Hyper-threading的CPU上使用多线程程序设计的好处是显而易见的，即提高了程序的执行吞吐率。以人工作的样子想像，核心相当于人，人越多则能同时处理的事情越多，而线程相当于手，手越多则工作效率越高。在单CPU单核的计算机上，使用多线程技术，也可以把进程中负责I/O处理、人机交互而常被阻塞的部分与密集计算的部分分开来执行，编写专门的workhorse线程执行密集计算，虽然多任务比不上多核，但因为具备多线程的能力，从而提高了程序的执行效率。

同一进程中的多条线程将共享该进程中的全部系统资源，如虚拟地址空间，文件描述符和信号处理等等。但同一进程中的多个线程有各自的调用栈（call stack），自己的寄存器环境（register context），自己的线程本地存储（thread-local storage）。

状态：
线程有四种基本状态，分别为：

- 产生（spawn）
- 阻塞（block）
- 非阻塞（unblock）
- 结束（finish）

2.Pthread库的使用

Linux系统下的多线程遵循POSIX线程接口，称为pthread。编写Linux下的多线程程序，需要使用头文件pthread.h，连接时需要使用库libpthread.a。

数据类型：

pthread_t：线程句柄
pthread_attr_t：线程属性

线程操纵函数（简介起见，省略参数）：

pthread_create()：创建一个线程
pthread_exit()：终止当前线程
pthread_cancel()：中断另外一个线程的运行
pthread_join()：阻塞当前的线程，直到另外一个线程运行结束
pthread_attr_init()：初始化线程的属性
pthread_attr_setdetachstate()：设置脱离状态的属性（决定这个线程在终止时是否可以被结合）
pthread_attr_getdetachstate()：获取脱离状态的属性
pthread_attr_destroy()：删除线程的属性
pthread_kill()：向线程发送一个信号

同步函数：

用于 mutex 和条件变量
pthread_mutex_init() 初始化互斥锁
pthread_mutex_destroy() 删除互斥锁
pthread_mutex_lock()：占有互斥锁（阻塞操作）
pthread_mutex_trylock()：试图占有互斥锁（不阻塞操作）。当互斥锁空闲时将占有该锁；否则立即返回
pthread_mutex_unlock()：释放互斥锁
pthread_cond_init()：初始化条件变量
pthread_cond_destroy()：销毁条件变量
pthread_cond_wait()：等待条件变量的特殊条件发生
pthread_cond_signal()：唤醒第一个调用pthread_cond_wait()而进入睡眠的线程
Thread-local storage（或者以Pthreads术语，称作 线程特有数据）：
pthread_key_create()：分配用于标识进程中线程特定数据的键
pthread_setspecific()：为指定线程特定数据键设置线程特定绑定
pthread_getspecific()：获取调用线程的键绑定，并将该绑定存储在 value 指向的位置中
pthread_key_delete()：销毁现有线程特定数据键

与一起工作的工具函数：

pthread_equal()：对两个线程的线程标识号进行比较
pthread_detach()：分离线程
pthread_self()：查询线程自身线程标识号

详解：

```
extern int pthread_create __P ((pthread_t *__thread, __const pthread_attr_t
*__attr,void *(*__start_routine) (void *), void *__arg));
```

第一个参数为指向线程标识符的指针，第二个参数用来设置线程属性，第三个参数是线程运行函数的起始地址，最后一个参数是运行函数的参数。

当运行函数需要多个参数时，多个参数需要用结构题封装，而且结构体需要在函数内自己打开

```
extern int pthread_join __P ((pthread_t __th, void **__thread_return));
```

第一个参数为被等待的线程标识符，第二个参数为一个用户定义的指针，它可以用来存储被等待线程的返回值。这个函数是一个线程阻塞的函数，调用它的函数将一直等待到被等待的线程结束为止，当函数返回时，被等待线程的资源被收回。一个线程的结束有两种途径，一种是像我们上面的例子一样，函数结束了，调用它的线程也就结束了；另一种方式是通过函数pthread_exit来实现。

```
extern void pthread_exit __P ((void *__retval)) __attribute__ ((__noreturn__));
```

唯一的参数是函数的返回代码，只要pthread_join中的第二个参数thread_return不是NULL，这个值将被传递给 thread_return。最后要说明的是，一个线程不能被多个线程等待，否则第一个接收到信号的线程成功返回，其余调用pthread_join的线程则返回错误代码ESRCH。

3.结构体内包含函数

```
//与类不同，结构体内不能直接包含一个函数，但是可以通过函数指针的方式，举例如下：
#include<iostream>
#include<string>
using namespace std;
class Hello{
public:
    void sayHello(string name){
      cout<<"你好，"<<name<<endl;
    }
};
int main(){
    Hello* hello=new Hello();
    hello->sayHello("a");
    return 0;
}
```

三、实验过程

实验思路：首先将获取到的数组分为两部分（此处要注意是奇数，还是偶数个数字），之后两个线程分别对这两部分进行排序，最后第三个线程通过归并的方法将两部分合为一部分，从而完成排序。

1. 初始化阶段：获取数组并将其分为两部分，为了实现动态大小的排序，我们首先获取要输入的数字个数。对于单个数字会直接将其划为第二个部分，不需要进行特殊处理，不过也可以进行直接输出的优化。

```
    puts("\033[34mPlease input the amount of the numbers you want to input:\033[m");
    scanf("%zd",&count);//首先获取要输入数组的大小
    if(count < 1){
        puts("\033[31mError: The amount of the numbers must be more than 1 !!!\033[m");
```

```
        exit(-1);
    }
    half = count/2;
    n = count - half;

    arr = (int *) malloc(sizeof(int)*count);
    arr_1 = (int *) malloc(sizeof(int)*count);
    check_malloc(arr);

    puts("\033[34mPlease input your numbers:\033[m");

    size_t i = 0;
    for (size_t i = 0; i < count; i++)
    {
        scanf("%d",arr+i);
    }


    puts("The following nums will be sorted by thread 1:");
    for (size_t i = 0; i < half; i++)
        printf("%d ", arr[i]);
    puts("");

    puts("The following nums will be sorted by thread 2:");
    for (size_t i = half; i < count; i++)
        printf("%d ", arr[i]);
    puts("");
```

2. 创建两个线程对于两部分分别进行排序

此时为了拓展性和高效性，我选择直接使用`qsort ()`函数进行第一步排序，但是面临的问题是`qsort ()`函数需要多个参数要排序的数组，数组的大小，每个数字的size和比较函数，因此我将其封装到一个结构体中，比较函数通过函数指针的形式实现。

由于直接将结构体传入`qsort ()`函数，`qsort ()`函数无法解析，我又定义了一个`sort(void * p1)`函数帮助其解析结构体。此处要注意`pthread_create()`函数传入的指针均是`void`指针，需要进行类型转换，要不然取值的时候一定会出问题（后面我写的时候有遇到这个问题，将`char`型指针直接用`int`型取，就会导致原本应取1字节值，直接取4字节值，而且编译器不会报错）

最后要使用`pthread_join()`函数使主函数等待两个排序函数的完成。

```
typedef struct paramater
{
    int *arr;
    size_t len;
    size_t width;
    int (*cmp) (const void * a, const void * b);

} para;
```

```
void *sort(void * p1){
    para *p = (para *)p1;
    qsort(p->arr,p->len,p->width,p->cmp);
    pthread_exit(0);
}

    para p_1 = {arr,half,sizeof(int),cmp};
    para p_2 = {arr+half,count-half,sizeof(int),cmp};
    pthread_create(&pthread_1, NULL, sort, &p_1);
    pthread_create(& pthread_2, NULL, sort, &p_2);
    pthread_join(pthread_1, NULL);
    pthread_join(pthread_2, NULL);
```

3. 第三个线程将两部分归并排序并输出

此处归并排序属于基础知识，要注意的是使用`pthread_join()`等待第三个线程结束之后再将字符串进行输出

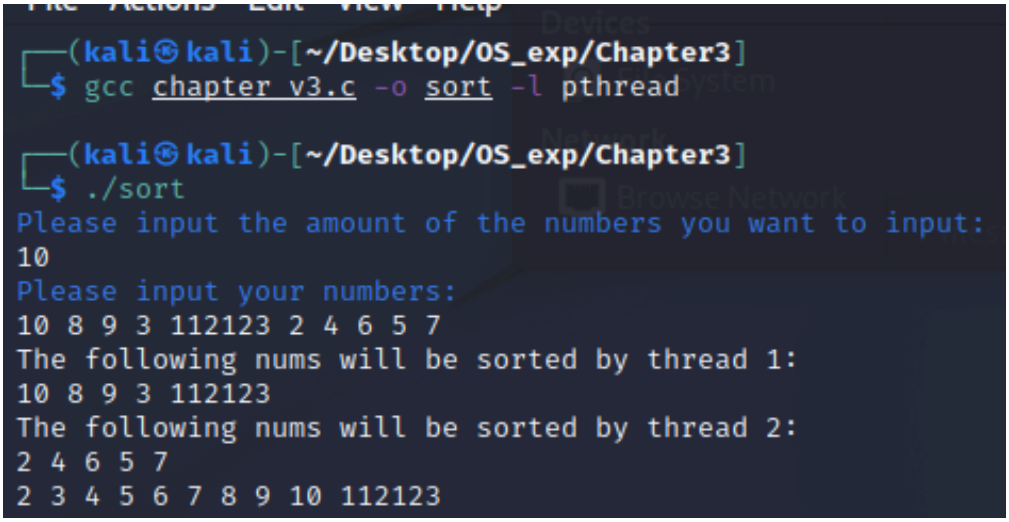
```
void *sort_last(){
    int ptr=0,i=0,j=half;
    while (i < half && j < count)
    {
        if (arr[i]<arr[j])
        {
            arr_1[ptr] = arr[i];
            i++;
        }else{
            arr_1[ptr] = arr[j];
            j++;
        }
        ptr++;
    }
    while (i < half)
    {
        arr_1[ptr] = arr[i];
        i++;ptr++;
    }
    while (j < count)
    {
        arr_1[ptr] = arr[j];
        j++;ptr++;
    }
    pthread_exit(0);
}

    pthread_create(&pthread_3, NULL, sort_last, NULL);
    pthread_join(pthread_3, NULL);
    for (size_t i = 0; i < count; i++)
```

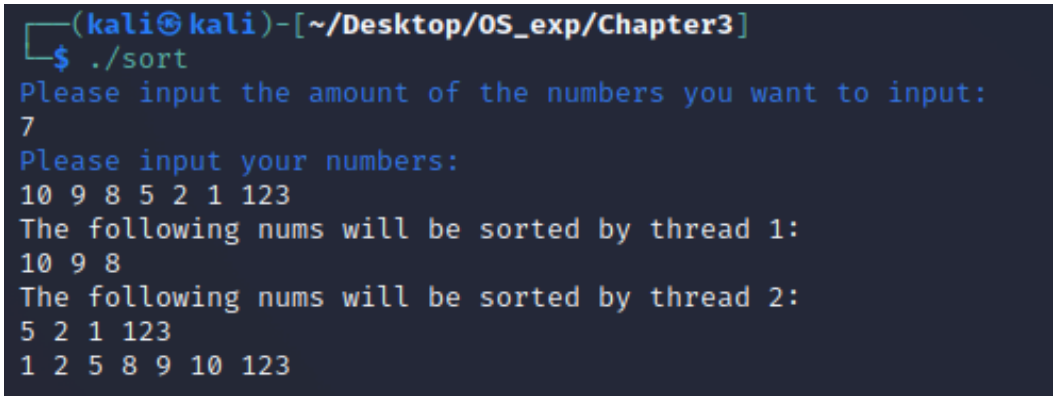
```
    {
        printf("%d ",arr_1[i]);
    }
    puts("");
```

四、实验结果与分析

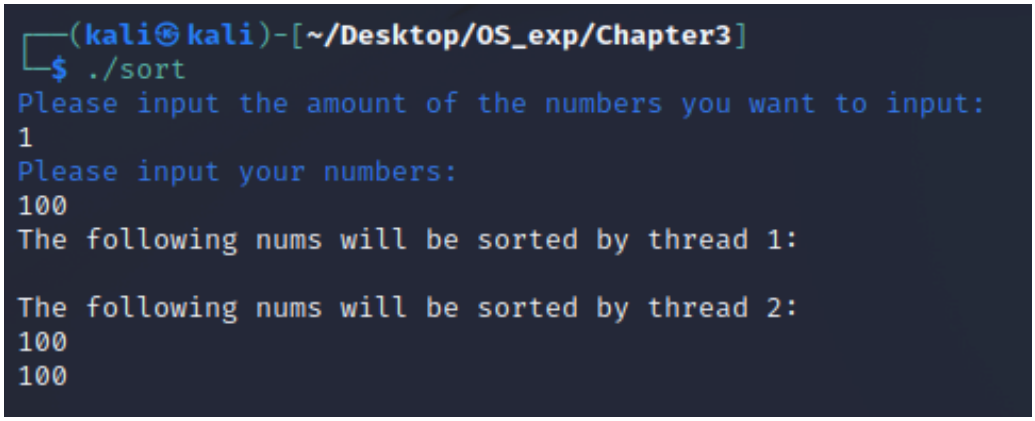
偶数



奇数



单个数字



由于`pthread`非Linux的默认库，所以编译时候记得`-l pthread`，可以看到我们的程序可以将前后两部分分别进行排序之后进行归并排序，同时支持多个数字（奇数、偶数）和单个数字

五、问题总结

1. 指针类型转换

在`cmp()`函数中进行强制类型转换时，将原本是`char`型的指针误转成了`int`型，原本应该只取一字节的值直接取了四字节，并且编译器不报错，找了好久才发现，该程序中很多函数都有涉及到强制类型转换，使用时一定要注意。

对于`pthread_create()`使用结构体传多参数时也要注意指针类型转换，传递的指针均作为`void`型指针进行传递，在使用时一定要注意将`void`型指针转换回结构体指针，才能正常使用。

六、源代码

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


pthread_t pthread_1,pthread_2,pthread_3;
int *arr,*arr_1;
size_t count, half,n;

typedef struct paramater
{
    int *arr;
    size_t len;
    size_t width;
    int (*cmp) (const void * a, const void * b);

} para;

typedef struct array{
    int *arr;
    size_t numbers;
} arry;

int cmp(const void * a, const void * b)
{
    return ( *(int *)a - *(int *)b );
}

void *sort(void * p1){
    para *p = (para *)p1;
    qsort(p->arr,p->len,p->width,p->cmp);
    pthread_exit(0);
}
```

```
void *sort_last(){
    int ptr=0,i=0,j=half;
    while (i < half && j < count)
    {
        if (arr[i]<arr[j])
        {
            arr_1[ptr] = arr[i];
            i++;
        }else{
            arr_1[ptr] = arr[j];
            j++;
        }
        ptr++;
    }
    while (i < half)
    {
        arr_1[ptr] = arr[i];
        i++;ptr++;
    }
    while (j < count)
    {
        arr_1[ptr] = arr[j];
        j++;ptr++;
    }
    pthread_exit(0);
}

void check_malloc(void *t){
    if(t == NULL){
        puts("\033[31mError: Fail to malloc !!!\033[m");
        exit(-1);
    }
}

int main(){
    puts("\033[34mPlease input the amount of the numbers you want to input:\033[m");
    scanf("%zd",&count);//首先获取要输入数组的大小
    if(count < 1){
        puts("\033[31mError: The amount of the numbers must be more than 1 !!!\033[m");
        exit(-1);
    }
    half = count/2;
    n = count - half;

    arr = (int *) malloc(sizeof(int)*count);
    arr_1 = (int *) malloc(sizeof(int)*count);
    check_malloc(arr);
```

```
    puts("\033[34mPlease input your numbers:\033[m");

    size_t i = 0;
    for (size_t i = 0; i < count; i++)
    {
        scanf("%d",arr+i);
    }


    puts("The following nums will be sorted by thread 1:");
    for (size_t i = 0; i < half; i++)
        printf("%d ", arr[i]);
    puts("");

    puts("The following nums will be sorted by thread 2:");
    for (size_t i = half; i < count; i++)
        printf("%d ", arr[i]);
    puts("");
    para p_1 = {arr,half,sizeof(int),cmp};
    para p_2 = {arr+half,count-half,sizeof(int),cmp};
    pthread_create(&pthread_1, NULL, sort, &p_1);
    pthread_create(& pthread_2, NULL, sort, &p_2);
    pthread_join(pthread_1, NULL);
    pthread_join(pthread_2, NULL);


    arry ar = {arr,count};
    pthread_create(&pthread_3, NULL, sort_last, NULL);
    pthread_join(pthread_3, NULL);
    for (size_t i = 0; i < count; i++)
    {
        printf("%d ",arr_1[i]);
    }
    puts("");
}
```