

北京交通大学

Operating System

Lab02: Multithreaded programming

学 院：	计算机与信息技术学院
学生姓名：	刘嘉鹏
学 号：	20281319

北京交通大学

2022 年 11 月

目录

1. Linux 系统线程控制概述	3
2. 程序设计与实现	5
3. 运行结果及分析	6
4. 总结与建议	7
5. 参考材料	7

1. Linux 系统线程控制概述

1.1 线程的概念

假设一个进程要完成两个任务，并且一个阻塞使另一个也无法执行。那么引入线程就是将一个进程所要做的两个任务分成两个线程来完成，从而一个任务的阻塞不会影响到后面任务的完成。也即，线程是进程内部的一条执行序列（执行流），每个进程至少有一条执行序列：**main** 的执行体。进程可以通过线程库创建若干条线程，这些新创建的线程称之为函数线程，**main** 函数所代表的线程为主线程。

1.2 线程与进程的联系

- (1) 进程是资源分配的最小单位，线程是 CPU 调度的最小单位；
- (2) 线程是“轻量级的进程”，由于线程比进程更小，基本上不拥有系统资源，故对它的调度所付出的开销就会小得多，能更高效的提高系统内多个程序间并发执行的程度；
- (3) 进程间相互独立，同一进程的各线程间共享。

1.3 Linux 系统 C 语言线程控制相关方法

首先，Linux 系统下使用进程控制相关函数，需包含头文件<pthread.h>。

(1) 线程创建

在Linux系统中，我们使用pthread_create(pthread_t *thread, const pthread_attr_t* attr, void *(*start_routine)(void*), void *arg)来创建进程。该函数含有四个参数：

第一个参数：pthread_t 代表创建线程的唯一标识，是一个结构体，需要我们创建好后，将这个结构体的指针传递过去（当成 int 就可以）。

第二个参数：pthread_attr_t，代表创建这个线程的一些配置，比如分配栈的大小等等。一般我们可以填 NULL，代表默认的创建线程的配置。

第三个参数：代表一个函数的地址，创建线程时，会调用这个函数作为子线程，函数的返回值是 void*，函数的参数也是 void*。

第四个参数：调用目标函数所需要的参数，类型也是 void*。

函数成功则返回 0，否则创建线程失败。

当线程创建成功后，子线程就会开始运行，子线程与主线程之间、子线程与子线程之间的运行顺序、快慢不好确定，因此我们需要通过线程挂起（阻塞）来控制子线程的运行次序。

（2）线程挂起

我们使用 `int pthread_join(pthread_t th, void **thr_return);` 来使当前线程挂起（阻塞），等待线程 `id` 为 `th` 的子线程结束后再继续运行。通过调用线程挂起，可以确保 `th` 线程先于当前线程运行结束。

（3）线程终止

一般而言线程运行结束就意味着终止，如果想提前结束当前线程，可以通过调用 `pthread_exit(void *arg);` 来实现，这与子线程通过 `return` 返回是类似的。

如果想在当前线程结束某个其它线程，可以调用线程取消函数：`int pthread_cancel(pthread_t th);` 取消成功返回 `0`，否则取消失败。

（4）线程分离

在当前线程调用 `int pthread_detach(pthread_t th);` 函数使 `th` 线程处于被分离状态。

如果不等待一个线程，同时对线程的返回值不感兴趣，可以设置这个线程为被分离状态，让系统在线程退出的时候自动回收它所占用的资源。

一个线程不能自己调用 `pthread_detach` 改变自己为被分离状态，只能由其他线程调用 `pthread_detach`。

（5）线程锁

上锁、解锁函数分别为：

```
int pthread_mutex_lock(pthread_mutex_t *mutex);
```

```
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

很好理解，当一个线程上锁后，如果有其它线程进入，那么必须等待之前上锁的线程解锁。通过线程锁可以很好地解决数据共享问题。

2. 程序设计与实现

2.1 多线程判定数独矩阵

先不做多线程处理，只考虑一般情况：

建立三个函数 `checkX`，`checkY`，`checkSub`，分别判定某一行、某一列、某一 3×3 小块是否满足数字 1-9 不重不漏，这样只需判断 9 行、9 列、9 块就可以解决问题。当且仅当所有的判断均正确（符合要求），该矩阵才为有效的数独矩阵，否则无效。

依题设要求，进行多线程处理，可以考虑把每一个小判断都独立出一个线程，共计 27 个线程。所有线程间共用一个 `bool` 数组，每个线程依据它所对应的行、列或者 3×3 小块的判断结果对 `bool` 数组一一赋值，当 `bool` 数组均为 1（判断符合要求）时，该矩阵有效，否则无效。

判断行、列时，函数仅需要一个 `int` 类型变量作为参数即可判断在具体某行或列，判断 3×3 小块时，将左上角坐标作为函数参数，所以需要自定义结构体：

```
01: typedef struct {
02:     int x, y;
03: } pos; // 块的左上角坐标，默认块的大小是3*3的
```

对于每一个判断是否有效的子线程，其判断过程需要用到 `check` 数组。循环遍历数据，当数字 `i` 第一次出现时(`check[i]=0`)，把对应 `check[i]`置 1；由于总共有 9 个数字，故当有数字 `j` 是第二次出现（即 `check[j]=1`）时，判定为无效，否则有效。

2.2 多线程（双线程）排序算法

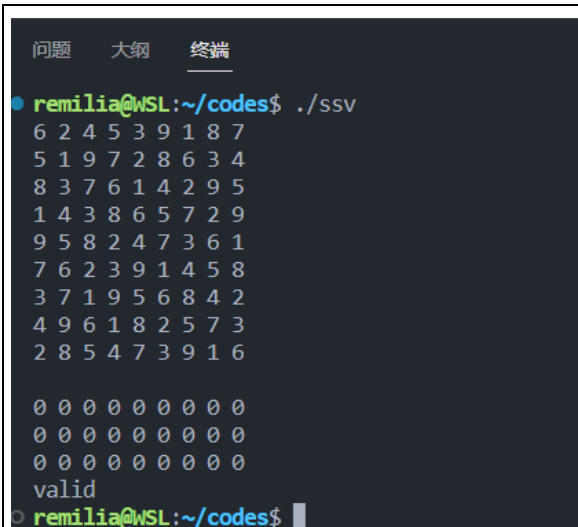
不考虑多线程处理的方法分析题设，就是简单的分两块排序再 `merge` 的过程。把数组从中间拆分，左、右两侧分别排序，这很好实现（最简单的冒泡排序、插入排序等等），而 `merge` 过程也就是设计两个下标变量，都从两个部分数组最小值开始遍历，每次都把二者对应的较小值加入结果数组，依次类推即可，需要注意当一个部分数组遍历完全后，需要把另一个部分数组的数据逐一加入到结果数组内。

加上多线程处理：为左、右部分数组的排序以及 `merge` 过程都分别创建一个子线程，三者共用一个待排序原数组和结果数组。但是这里需要注意，`merge` 必须再两部分数组排序结束后再运行才正确，否则会产生错误！

3. 运行结果及分析

3.1 多线程判定数独矩阵

直接用原矩阵进行判断：



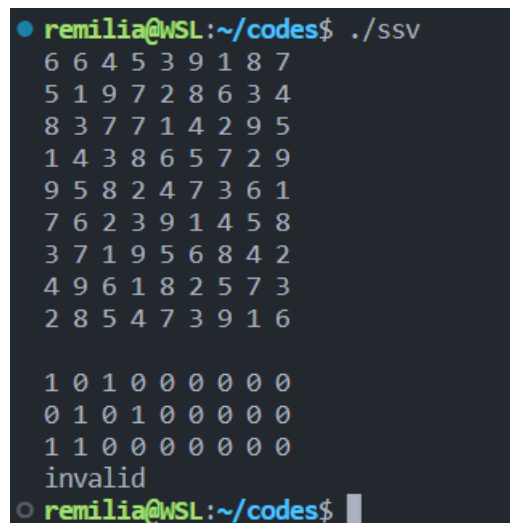
```
问题 大纲 终端
remilia@WSL:~/codes$ ./ssv
6 2 4 5 3 9 1 8 7
5 1 9 7 2 8 6 3 4
8 3 7 6 1 4 2 9 5
1 4 3 8 6 5 7 2 9
9 5 8 2 4 7 3 6 1
7 6 2 3 9 1 4 5 8
3 7 1 9 5 6 8 4 2
4 9 6 1 8 2 5 7 3
2 8 5 4 7 3 9 1 6

0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0
valid
remilia@WSL:~/codes$
```

行、列、块均满足要求，`flag` 均为 0，很正确。

把 1 行 2 列的数字 2 改为 6，3 行 4 列的数字 6 改为 7，再次分析。

第 1、3 行，第 2、4 列，第 1、2 块出错，满足程序逻辑结果。



```
remilia@WSL:~/codes$ ./ssv
6 6 4 5 3 9 1 8 7
5 1 9 7 2 8 6 3 4
8 3 7 7 1 4 2 9 5
1 4 3 8 6 5 7 2 9
9 5 8 2 4 7 3 6 1
7 6 2 3 9 1 4 5 8
3 7 1 9 5 6 8 4 2
4 9 6 1 8 2 5 7 3
2 8 5 4 7 3 9 1 6

1 0 1 0 0 0 0 0 0
0 1 0 1 0 0 0 0 0
1 1 0 0 0 0 0 0 0
invalid
remilia@WSL:~/codes$
```

3.2 多线程（双线程）排序算法

测试原样例，结果正确

```
● remilia@WSL:~/codes$ gcc -o msa Multithreaded_Sorting_Application.c -lpthread
● remilia@WSL:~/codes$ ./msa
input numbers:
7 12 19 3 18 4 2 6 15 8
sorted numbers:
2 3 4 6 7 8 12 15 18 19
○ remilia@WSL:~/codes$
```

随机测试：

```
● remilia@WSL:~/codes$ ./msa
input numbers:
-123 453 657 876 423 75765 234234 -523 768 -4357 34 61 123 -233 0
sorted numbers:
-4357 -523 -233 -123 0 34 61 123 423 453 657 768 876 75765 234234
○ remilia@WSL:~/codes$
```

排序结果正确无误。

4. 总结与建议

这次实验难度较小，重点在于 `pthread_create` 和 `pthread_join` 的应用。利用这两个函数进行线程控制，可以很好地管理子线程和主线程等等之间的执行顺序。相比理论学习，这次实验更偏向于实践联系。

5. 参考材料

Linux 线程控制：<https://www.cnblogs.com/nfcm/p/7653433.html>

归并排序算法：<https://www.geeksforgeeks.org/merge-sort/>