

# Lab02: Multithreaded programming

---

Name : 马一文 Sid : 21231072

## 目录

### Lab02: Multithreaded programming

The thread control methods provided by Linux

pthread\_create

pthread\_join

pthread\_exit

pthread\_cancel

pthread\_mutex\_t

Part I—Sudoku Solution Validator

Design of the program

Snapshots & analysis

Problems encountered and solution

Part II—Multithreaded Sorting Application

Design of the program

Snapshots & analysis

Problems encountered and solution

Reference materials

Suggestions and comments

For Sudoku Solution Validator :

For Multithreaded Sorting Application:

Appendix

README.md

I. Sudoku Validator

Features

Getting Started

Prerequisites

Compilation

Usage

Code Structure

How It Works

II. Multi-threaded Merge Sort

Features

Getting Started

Prerequisites

Compilation

Usage

Code Structure

How It Works

Contact

Code I : sudoku.c

Code II : sort.c

## The thread control methods provided by Linux

---

线程 (Thread) 是计算机程序的执行单元, 它是进程 (Process) 内的一个更小的子任务。线程是操作系统调度的基本单位, 多个线程可以共享同一个进程的内存空间和资源, 使得多任务并发执行成为可能。线程有以下关键特点:

1. 共享进程资源：线程属于同一进程，因此它们共享相同的内存空间、文件句柄、打开文件以及其他进程资源。这使得线程之间更容易进行通信和数据共享。
2. 轻量级：相对于进程，线程是轻量级的，因为它们共享进程的地址空间，不需要像进程那样拥有独立的内存副本。因此，线程的创建和切换开销较小，使得多线程应用程序更高效。
3. 并发执行：多个线程可以同时运行，从而实现并发执行。这有助于提高系统的响应性和利用多核处理器的性能。
4. 独立执行路径：每个线程都有自己的执行路径，可以独立执行任务。线程之间可以协作完成复杂的任务，例如数据处理、图形渲染、网络通信等。
5. 同步和互斥：多线程应用需要谨慎处理共享数据，以避免竞态条件和数据不一致性。为此，线程可以使用同步和互斥机制，如互斥锁、信号量和条件变量，来确保线程之间的协作和数据访问的安全性。

线程在操作系统层面提供了不同的API和库，允许程序员创建、管理和同步线程。在Linux和其他POSIX兼容系统中，通常使用Pthreads（POSIX Threads）库来创建和管理线程。其他操作系统，如Windows，也提供了自己的线程管理API。

以下几个主要的线程控制函数：

## pthread\_create

`pthread_create` 函数是POSIX线程（Pthreads）库中的一个重要函数，用于创建新线程。它的原型如下：

```
int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void *
(*start_routine)(void*), void *arg);
```

1. `thread`：一个指向 `pthread_t` 类型的指针，用于存储新线程的标识符。创建的线程将在 `thread` 中返回其唯一标识符，你可以使用这个标识符来操作或等待线程。
2. `attr`：一个指向 `pthread_attr_t` 类型的指针，用于指定线程的属性。通常可以将其设置为 `NULL`，表示使用默认属性。如果需要更多的线程属性控制，可以创建并设置 `pthread_attr_t` 结构体。
3. `start_routine`：一个函数指针，指向新线程将执行的函数。这个函数应该具有如下签名：  
`void* start_routine(void* arg)`，其中 `arg` 是传递给函数的参数。
4. `arg`：一个指向 `void` 类型的指针，用于传递给 `start_routine` 函数作为其参数。可以用来传递任何数据给新线程的执行函数。

## pthread\_join

`pthread_join` 函数是POSIX线程（Pthreads）库中的一个用于线程同步的函数。它允许一个线程等待另一个线程的结束。`pthread_join` 的原型如下：

```
int pthread_join(pthread_t thread, void **retval);
```

1. `thread`：要等待的线程的标识符，通常由 `pthread_create` 返回。
2. `retval`：一个指向 `void*` 指针的指针，用于存储目标线程的返回值。如果你不关心目标线程的返回值，可以将其设置为 `NULL`。

## pthread\_exit

`pthread_exit` 函数是POSIX线程（Pthreads）库中的一个函数，用于终止调用它的线程的执行。该函数允许线程自行退出，通常在完成任务或遇到错误时使用。`pthread_exit` 的原型如下：

```
void pthread_exit(void *retval);
```

1. `retval`：一个指向 `void` 类型的指针，用于指定线程的返回值。这个返回值可以被其他线程使用 `pthread_join` 函数获取。

`pthread_exit` 的主要作用是将当前线程终止，并返回一个指定的返回值。这可以用于传递线程的执行结果给等待该线程完成的其他线程。

## pthread\_cancel

`pthread_cancel` 函数是 POSIX 线程（Pthreads）库中的一个函数，用于请求取消一个正在执行的线程。取消线程意味着中断线程的执行，但线程可以选择在适当的时机响应取消请求。这是一种线程管理和同步的机制。`pthread_cancel` 的原型如下：

```
int pthread_cancel(pthread_t thread);
```

1. `thread`：要取消的线程的标识符，通常由 `pthread_create` 返回。

`pthread_cancel` 函数向指定的线程发送一个取消请求，但线程是否实际取消取决于线程本身的行为。线程可以设置为忽略取消请求，或者可以选择在适当的时机执行取消操作。通常，取消请求可以通过以下方式之一实现：

- 异步取消（Asynchronous Cancellation）：线程可以在任何时刻被取消。这意味着线程可能在任何指令执行后被取消，不会执行清理工作。
- 推迟取消（Deferred Cancellation）：线程会在取消点（cancellation point）处检查是否有取消请求。取消点通常是一些标准库函数，如 `pthread_join`、`pthread_testcancel`、`sleep` 等，或者可以通过编程方式设置取消点。

## pthread\_mutex\_t

`pthread_mutex_t` 是 POSIX 线程库中用来实现互斥锁（mutex）的数据类型。互斥锁是一种线程同步机制，用于保护临界区（一段代码，只能由一个线程同时执行）免受多个线程的并发访问。

以下是一些常见的 `pthread_mutex_t` 相关函数和操作：

1. `pthread_mutex_init`：用于初始化互斥锁。这个函数会创建一个新的互斥锁，并设置为未锁定状态。

```
int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *attr);
```

2. `pthread_mutex_destroy`：用于销毁互斥锁。这个函数会释放互斥锁所占用的资源。

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

3. `pthread_mutex_lock`：用于锁定互斥锁。如果互斥锁已被其他线程锁定，调用线程将被阻塞，直到互斥锁被释放。

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

4. `pthread_mutex_unlock`：用于释放互斥锁。如果有其他线程在等待这个互斥锁，一个线程释放锁后，其他线程可以竞争锁。

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

5. `pthread_mutex_trylock`：尝试锁定互斥锁，如果锁已被其他线程锁定，则不会阻塞，而是返回一个错误码。

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

6. `pthread_mutexattr_init` 和 `pthread_mutexattr_destroy`：用于初始化和销毁互斥锁属性对象，可以用来设置互斥锁的属性，如递归锁、错误检查等。

```
int pthread_mutexattr_init(pthread_mutexattr_t *attr);  
int pthread_mutexattr_destroy(pthread_mutexattr_t *attr);
```

这些函数允许用户在多线程应用程序中创建和管理互斥锁，以确保多个线程能够协调访问共享资源，避免竞争条件和数据损坏。

## Part I—Sudoku Solution Validator

### Design of the program

该程序目的是为了验证一个数独（Sudoku）是否合法。数独是一个经典的逻辑解谜游戏，要求在一个9x9的格子中填入数字，使得每一行、每一列和每一个3x3的子格中的数字都不重复。这个程序通过多线程的方式来验证数独的合法性，以下是程序的主要思路 and 结构：

1. 数据结构定义：

- `Point` 结构用于表示数独中的坐标点，包括 `x` 和 `y` 坐标。
- `Data` 结构包含了一个数独块（行、列、子格）的上左角和下右角的坐标、线程索引以及一个标志（`flag`），用于指示该数独块是否合法。
- 二维数组 `array[9][9]` 用于存储数独的数字。
- 一个 `Data` 数组 `data[27]` 用于存储验证数独的任务数据。

2. 输入数独：

- 通过 `file_input` 函数从文件中读取数独数据，或通过 `manual_input` 函数手动输入数独数据，读取的数据存储在 `array` 数组中。

3. 初始化数据索引：

- `init_data_index` 函数用于初始化 `data` 数组，将每个数据块的上左角和下右角坐标设置为相应的数独区域（行、列和3x3子网格），并设置线程索引，省去了写多个线程函数的麻烦。

4. 线程验证：

- 主函数 `main` 创建了27个线程（一个用于验证每个数独块），并将数据块作为参数传递给每个线程。
- 每个线程执行 `valid` 函数，根据传递的数据块验证相应的数独块是否合法。
- `valid` 函数使用计数数组 `count` 来检查每个数独块中的数字是否重复，如果发现重复则设置标志 `flag` 并退出线程。

5. 同步和结果判断：

- 主函数使用 `pthread_join` 等待所有线程完成验证。
- 最后，主函数检查每个数据块的 `flag` 值，如果有任何一个数独块不合法，程序输出 "INVALID"，否则输出 "VALID"。

## Snapshots & analysis

为测试方便，测试时使用的输入方式均为文件输入，即 `1.file_input`。

- 测试数据(合法):

```
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
```

- 测试数据(不合法):

```
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 6 3
2 8 5 4 7 3 9 1 7
```

```
maeven@maeven-virtual-machine:~/桌面/assignment_2$ gcc sudoku.c -o sudoku -pthread
maeven@maeven-virtual-machine:~/桌面/assignment_2$ ./sudoku
Please select the input data method:
1. File import 2. Manual input
1
Please input file name(path): test_valid
Sudoku read successfully!
SUDOKU:
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
-----
VALID!
maeven@maeven-virtual-machine:~/桌面/assignment_2$ ./sudoku
Please select the input data method:
1. File import 2. Manual input
1
Please input file name(path): test_invalid
Sudoku read successfully!
SUDOKU:
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 6 3
2 8 5 4 7 3 9 1 7
-----
INVALID!!
```

## Problems encountered and solution

在编写这个程序时，可能会遇到一些问题，下面是一些可能的问题和解决方案：

- 线程并发问题：**由于多线程同时访问共享的 `array` 数组，可能会出现竞争条件和数据不一致性问题。
  - 解决方案：**需要使用同步机制来保护共享资源。可以使用互斥锁（`pthread_mutex_t`）来确保一次只有一个线程访问共享的 `array` 数组。在 `valid` 函数中，需要使用锁来保护对 `array` 的访问。
- 取消线程的处理：**在 `valid` 函数中，如果发现数独无效，线程会使用 `pthread_exit` 提前退出。
  - 解决方案：**在主线程中，需要检查每个线程的返回状态以确定是否有线程已经退出。这可以使用 `pthread_join` 函数来等待线程的退出，并检查返回状态来判断数独是否有效。
- 用户输入问题：**根据用户选择，程序可以从文件或手动输入获取数独数据。可能会出现文件不存在或格式错误的问题。
  - 解决方案：**在文件输入部分，需要检查文件是否存在，以及在读取文件时是否发生错误。可以使用文件输入的返回值来检查文件读取的成功或失败。在手动输入部分，需要确保用户输入的数据符合数独的规则。
- 内存泄漏问题：**在 `valid` 函数中，如果线程退出，它可能会分配内存并在退出前没有释放。
  - 解决方案：**在 `valid` 函数中，需要确保释放任何动态分配的内存（如果有的话）。这可以通过在适当的时候调用 `free` 函数来完成。
- 程序终止问题：**如果有任何线程发现数独无效，程序应该尽早终止。
  - 解决方案：**如果在验证数独时任何线程设置了 `flag` 表示无效，可以在其中一个线程退出后，立即使用 `return` 或 `exit` 终止程序的执行。这可以通过在主函数中检查标志来实现。

## Part II—Multithreaded Sorting Application

### Design of the program

本程序用于实现归并排序（Merge Sort）的多线程版本，用于对输入的一维数组进行排序。以下是代码的设计思路：

- 输入阶段：**
  - 用户输入要排序的元素数量以及每个元素的值。
  - 数组 `A` 用于存储这些输入值。
- 归并排序 (MergeSort)：**
  - 归并排序采用递归的方式进行排序。首先，检查是否需要进一步划分数组。
  - 如果数组需要划分，创建一个 `B` 数组，其中存储了当前排序范围的左、中、和右边界。
  - 使用 `pthread_create` 创建两个线程（`left_sort` 和 `right_sort`），分别用于排序左半部分和右半部分。这样，数组会被并行排序，提高了性能。
  - 使用 `pthread_join` 等待左半部分和右半部分的排序线程完成，然后释放 `B` 数组的内存。
  - 最后，调用 `Merge` 函数将排序好的左半部分和右半部分合并。
- 合并操作 (Merge)：**
  - 创建一个临时数组 `B` 来存储已排序的元素。
  - 使用指针 `i` 和 `j` 分别指向左半部分和右半部分的开始位置，以及 `k` 来表示临时数组 `B` 的索引。
  - 将左半部分和右半部分的元素按顺序比较，将较小的元素存储到临时数组 `B` 中，然后更新相应的指针。
  - 当左半部分或右半部分的元素全部处理完毕后，将剩余的元素依次复制到 `B` 中。
  - 最后，将临时数组 `B` 中的元素复制回原始数组 `A` 中，完成排序。



#### 4. 线程函数 (left\_sort 和 right\_sort):

- 这两个函数用于创建线程并调用 Mergersort 函数来排序左半部分和右半部分。
- 各自的参数 para 包含了左、中、和右边界信息，以便指示要排序的区间。

#### 5. 主函数 (main):

- 用户输入完成后，调用 Mergersort 函数对整个数组进行归并排序。
- 最终，将排序后的结果打印到标准输出。

## Snapshots & analysis

测试数据:

7 12 19 3 18 4 2 6 15 8

```
maeven@maeven-virtual-machine:~/桌面/assignment_2$ ./sort
Input the number of elements to be sorted(max:50):
10
Input each element in turn:
7 12 19 3 18 4 2 6 15 8
Sorted results:
2 3 4 6 7 8 12 15 18 19
```

经过测试，所编写程序运行无误。

## Problems encountered and solution

- 数据共享和竞争条件问题:** 多个线程同时访问和修改全局变量 A，可能会导致数据共享和竞争条件问题，从而破坏排序的正确性。
  - **解决方案:** 使用互斥锁 (mutex) 来保护共享数据，确保一次只有一个线程可以修改数组 A。在 Mergersort 函数中，可以创建一个互斥锁来保护对数组 A 的访问，在 Merge 函数中也可以使用互斥锁来保护对临时数组 B 的访问。
- 内存泄漏:** 在代码中使用了 malloc 分配内存，但没有释放内存，这可能导致内存泄漏问题。
  - **解决方案:** 在使用完内存后，使用 free 函数释放分配的内存，以防止内存泄漏。在 Mergersort 函数中，确保在合并前释放临时数组 B 的内存。
- 线程同步问题:** 多线程程序中，线程之间需要正确地协同工作，以确保排序的正确性。
  - **解决方案:** 使用线程同步机制，如 pthread\_join 来等待线程的完成，以确保在进行合并之前左半部分和右半部分的排序已经完成。此外，也要确保在释放临时数组 B 的内存之前，不要让其被其他线程访问。
- 线程创建和销毁开销:** 创建和销毁线程会带来一定的开销，如果不合理地创建过多的线程，可能会导致性能下降。
  - **解决方案:** 可以使用线程池来重复利用已创建的线程，而不是在每次排序操作中都创建新线程。这可以减小线程创建和销毁的开销。
- 性能问题:** 多线程程序的性能可能受到多个因素的影响，包括线程数量、任务划分等。
  - **解决方案:** 对于不同的硬件和数据规模，需要进行性能测试和优化。可以尝试不同的线程数量，调整任务的分配策略，以获得最佳的性能。

## Reference materials

[1] [Lab02.pdf](#)

[2] [Linux bash cheat sheet-1.pdf](#)

[3] [cplusplus.com](#)

[4] [Linux系统编程\(pthread\)线程创建与使用 - 知乎 \(zhihu.com\)](#)

[5] [Linux下对'pthread create'未定义的引用的解决办法 pthread 未定义-CSDN博客](#)

[6] [数据结构和算法：归并排序（合并排序）详解 合并算法-CSDN博客](#)

## Suggestions and comments

---

对于Lab02中的两个部分，即Sudoku解决验证和多线程排序应用程序，以下是一些建议：

### For Sudoku Solution Validator :

1. **计划和组织线程**：在创建线程之前，仔细规划和组织线程的任务。确保每个线程的任务清晰明确，例如一个线程负责检查列，一个线程负责检查行，其他线程负责检查子网格。这有助于更好地协调线程的工作。
2. **线程通信**：考虑如何实现线程之间的通信，以便在检查完各自的区域后，能够将结果传递回主线程进行最终验证。一种方法是使用共享的数据结构或数组，使线程能够写入其验证结果。确保在并发访问时使用适当的同步机制。
3. **错误处理**：考虑用户输入数据不符合数独规则的情况。在代码中加入适当的错误处理机制，以便提供有意义的错误消息和用户反馈。
4. **性能优化**：如果有需要，可以尝试优化程序以提高性能。例如，可以调整线程数目以充分利用多核处理器，但避免创建过多线程导致开销增加。

### For Multithreaded Sorting Application:

1. **选择排序算法**：选择适当的排序算法来对子数组进行排序。不同的排序算法在不同情况下性能有差异，因此选择适合问题规模和硬件的算法是重要的。
2. **数据划分**：确保正确划分数据给排序线程，以充分利用多线程。每个排序线程应该处理不同的数据块。
3. **数据合并**：合并线程的实现需要特别小心，以确保正确地将两个排序好的子数组合并成一个有序数组。
4. **错误处理**：在排序过程中，考虑错误处理机制，如处理无效的输入数据或排序失败的情况。
5. **性能优化**：在多线程排序应用程序中，要注意线程的创建和销毁开销。可以使用线程池或其他方法来降低这些开销，提高程序的性能。
6. **测试和验证**：对于每个部分，进行充分的测试和验证，以确保程序按预期工作并且产生正确的结果。特别是在多线程环境中，要考虑可能的竞争条件和数据一致性问题。

最重要的是，将任务细化成可管理的部分，对每个部分进行单独的设计、开发和测试，有助于提高代码的可维护性和可理解性。

## Appendix

---

### README.md

#### I. Sudoku Validator

##### Features

- Multi-threaded Sudoku validation: The program validates a Sudoku solution using multiple threads to check rows, columns, and 3x3 subgrids concurrently.
- Input options: You can either input the Sudoku data from a file or manually enter it.
- Efficient and fast: Using multiple threads allows for efficient validation of large Sudoku grids.



## Getting Started

### Prerequisites

- The code is written in C, so you need a C compiler (e.g., GCC) installed on your system.

### Compilation

You can compile the code using the GCC compiler. Open your terminal and navigate to the directory containing the code files and the README. Then, use the following command:

```
gcc sudoku.c -o sudoku -pthread
```

This will compile the code and create an executable named `sudoku`.

### Usage

1. To run the program with a Sudoku file:

```
./sudoku  
1
```

The program will prompt you to enter the file name (path) containing the Sudoku data. Make sure the file format matches the example provided (SudokuTest.txt).

2. To run the program with manual input:

```
./sudoku  
2
```

The program will prompt you to manually input the Sudoku data. Enter the numbers one by one as instructed.

The program will validate the Sudoku solution and print "VALID!" if it's a valid solution or "INVALID!!" if it's not valid.

### Code Structure

- `sudoku.c`: The main source code file containing the implementation of the Sudoku validation program.
- `README.md`: This documentation file.
- `SudokuTest.txt`: An example file containing a Sudoku solution.

### How It Works

1. The program starts by initializing the data structures for the Sudoku grid and thread data blocks.
2. The user can choose to input the Sudoku data from a file or manually.
3. The program creates 27 threads to validate each row, column, and 3x3 subgrid of the Sudoku grid concurrently. Each thread checks for duplicate numbers within its assigned region.
4. If any thread finds duplicate numbers within its region, it sets a flag indicating that the Sudoku solution is invalid and exits the thread.
5. The main thread waits for all validation threads to complete using `pthread_join`.
6. After all threads have finished, the main thread checks the flags set by each thread. If any thread has set the flag, the program reports "INVALID!!"; otherwise, it reports "VALID!"

7. The program terminates.

## II. Multi-threaded Merge Sort

This is a multi-threaded implementation of the Merge Sort algorithm in C using pthreads (POSIX threads). Merge Sort is a popular sorting algorithm known for its efficiency and stability. In this implementation, the sorting process is divided into two threads to take advantage of multi-core processors.

### Features

- Multi-threaded sorting for improved performance on multi-core CPUs.
- Thread-safe handling of shared data.
- Dynamically allocates memory for temporary arrays during sorting.
- Easy-to-use command-line interface.

### Getting Started

#### Prerequisites

To compile and run this code, you'll need a C compiler that supports POSIX threads (pthreads). On most Unix-like systems, this support is included by default. If you're using a different development environment, make sure it supports pthreads or install the necessary libraries.

#### Compilation

To compile the code, open your terminal and navigate to the directory containing the source code. Then, run the following command:

```
gcc sort.c -o sort -pthread
```

This will generate an executable file named `sort`.

#### Usage

You can run the program as follows:

```
./sort
```

Follow the on-screen instructions to input the number of elements to sort and the elements themselves. The program will then sort the elements using multi-threaded Merge Sort and display the sorted results.

#### Code Structure


- `sort.c`: The main source code file containing the implementation of multi-threaded Merge Sort.
- `pthread.h`: The header file for POSIX threads.
- `stdlib.h`: The standard library header file for memory allocation functions.
- `stdio.h`: The standard I/O header file for input and output functions.

#### How It Works

1. The program takes user input for the number of elements to be sorted and the elements themselves.
2. It divides the sorting process into two threads: one for the left half of the array and another for the right half.

3. Each thread further divides its portion into two threads for even finer-grained sorting.
4. This division and sorting process continues recursively until individual elements are sorted.
5. The sorted halves are then merged back together to produce the final sorted array.
6. The program uses dynamic memory allocation for temporary arrays to avoid memory wastage.

## Contact

If you have any questions or need further assistance, you can contact [EVEN-MA] at  E-mail: [21231027@bjtu.edu.cn](mailto:21231027@bjtu.edu.cn).

## Code I : sudoku.c

```
#include <stdio.h>
#include <pthread.h>

typedef struct {
    int x;
    int y;
} Point; /*point coordinates*/
typedef struct {
    Point upper_left;
    Point lower_right;
    int thread_index;
    int flag; /* value == 0 if sudoku valid */
} Data; /*data blocck*/

int array[9][9]; /* sudoku array, shared by the thread(s) */
Data data[27]; /* data passed to each thread */

void print() {
    printf("SUDOKU:\n");
    for (int i = 0; i < 9; i++) {
        for (int j = 0; j < 9; j++) {
            printf("%d ", array[i][j]);
        }
        printf("\n");
    }
}

int file_input() {
    printf("Please input file name(path): ");
    char str[1024];
    scanf("%s", str);
    FILE *fin = fopen(str, "r"); /*SudokuTest.txt*/
    if (!fin) {
        printf("Sudoku read failed!\n");
        fclose(fin);
        return 0;
    }
    for (int i = 0; i < 9; i++)
        for (int j = 0; j < 9; j++)
            fscanf(fin, "%d", &array[i][j]);
    printf("Sudoku read successfully!\n");
    fclose(fin);
    print();
    return 1;
}
```

```

int manual_input() {
    printf("Please input Sudoku:\n");
    for (int i = 0; i < 9; i++)
        for (int j = 0; j < 9; j++)
            scanf("%d", &array[i][j]);
    printf("Sudoku read successfully!\n");
    print();
    return 1;
}

/*init data passed to each thread*/
void init_data_index() {
    /*row*/
    for (int i = 0; i < 9; ++i) {
        int index = i; /*0--8*/
        data[index].upper_left.x = 0;
        data[index].upper_left.y = i;
        data[index].lower_right.x = 8;
        data[index].lower_right.y = i;
        data[index].thread_index = index;
    }
    /*column*/
    for (int i = 0; i < 9; ++i) {
        int index = i + 9; /*9--17*/
        data[index].upper_left.x = i;
        data[index].upper_left.y = 0;
        data[index].lower_right.x = i;
        data[index].lower_right.y = 8;
        data[index].thread_index = index;
    }
    /*3x3 subgrid*/
    for (int i = 0; i < 3; ++i) {
        for (int j = 0; j < 3; ++j) {
            int index = i * 3 + j + 18; /*18--26*/
            data[index].upper_left.x = i * 3;
            data[index].upper_left.y = j * 3;
            data[index].lower_right.x = i * 3 + 2;
            data[index].lower_right.y = j * 3 + 2;
            data[index].thread_index = index;
        }
    }
}

/*valid each sudoku array block*/
void *valid(void *param) {
    Data *data = (Data *) param;
    int count[10] = {0};
    for (int i = data->upper_left.x; i <= data->lower_right.x; ++i) {
        for (int j = data->upper_left.y; j <= data->lower_right.y; ++j) {
            if (++count[array[i][j]] > 1) {
                data->flag = 1;
                /*
                if (0 <= data[i].thread_index && data[i].thread_index < 9)
                    printf("row invalid!\n");
                if (9 <= data[i].thread_index && data[i].thread_index < 18)
                    printf("column invalid!\n");
                if (18 <= data[i].thread_index && data[i].thread_index < 27)
                    printf("subgrid invalid!\n");
                */
            }
        }
    }
}

```

```

        */
        pthread_exit(0);
    }
    else data->flag = 0;
}
}
pthread_exit(0);
}

int main() {

    pthread_t tid[27]; /*the thread identifier*/
    pthread_attr_t attr; /*set of thread attributes*/
    pthread_attr_init(&attr); /*get the default attributes*/
    init_data_index(); /*init data index passed to each thread*/

    /*input*/
    char ch;
    printf("Please select the input data method:\n1. File import \t2. Manual\n");
    ch = getchar();
    fflush(stdin);
    if (ch == '1') {
        if (!file_input()) {
            return 0;
        }
    } else {
        manual_input();
    }
    printf("-----\n");

    /* create the thread */
    for (int i = 0; i < 27; ++i) {
        pthread_create(&tid[i], &attr, valid, &data[i]);
    }

    /* wait for the thread to exit */
    for (int i = 0; i < 27; ++i) {
        pthread_join(tid[i], NULL);
    }

    /*valid sudoku*/
    for (int i = 0; i < 27; ++i) {
        if (data[i].flag != 0) {
            printf("INVALID!!\n");
            return 0;
        }
    }
    printf("VALID!\n");
    return 0;
}

```

## Code II : sort.c

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

#define maxsize 50

int A[maxsize]; // numbers to be sorted

void Mergersort(int left, int mid, int right); // recursive divide and conquer
void Merge(int left, int mid, int right); // orderly merge of arrays
void *left_sort(void* args);
void *right_sort(void* args);

int main() {
    int num;
    printf("Input the number of elements to be sorted(max:%d):\n", maxsize);
    scanf("%d", &num);
    printf("Input each element in turn:\n");
    for (int i = 0; i < num; i++)scanf("%d", &A[i]);
    Mergersort(0, (num - 1) / 2, num - 1);
    printf("Sorted results:\n");
    for (int i = 0; i < num; i++)printf("%d ", A[i]);
    printf("\n");
    return 0;
}

void Mergersort(int left, int mid, int right) {
    if (left < right) {
        int *B = (int*)malloc(sizeof(int) * 4);
        B[0] = left;
        B[1] = mid;
        B[2] = right;
        pthread_t tidL;
        pthread_t tidR;
        pthread_create(&tidL, NULL, left_sort, B);
        pthread_create(&tidR, NULL, right_sort, B);
        pthread_join(tidL, NULL);
        pthread_join(tidR, NULL);
        free(B);
        Merge(left, mid, right);
    }
}

void Merge(int left, int mid, int right) {
    //int *B = new int[right - left + 1];
    int *B = (int *)malloc((right - left + 1) * sizeof(int));
    int i = left;
    int j = mid + 1;
    int k = 0;
    //Sort from small to large, put the smaller elements in A[i] and A[j] into
    B[]
    while (i <= mid && j <= right) {
        if (A[i] <= A[j])B[k++] = A[i++];
        else B[k++] = A[j++];
    }
}
```



```

    while (i <= mid)B[k++] = A[i++];
    while (j <= right)B[k++] = A[j++];
    for (i = left, k = 0; i <= right; i++)A[i] = B[k++];
    //delete[] B;
    free(B);
}

void *left_sort(void* para) {
    int *p = (int*)para;
    Mergersort(p[0], (p[1] + p[0]) / 2, p[1]);
    pthread_exit(0);
}

void *right_sort(void* para) {
    int *p = (int*)para;
    Mergersort(p[1] + 1, (p[2] + p[1] + 1) / 2, p[2]);
    pthread_exit(0);
}

```