Project 3

秦啸涵　**5210121910604**

多线程校验数独

在这部分中，要求使用`pthread`库创建多线程校验一个9 × 9的数独是否正确。我为每一行，每一列和每一个3 × 3的格子分别创建一个子线程，共创建27个线程用于校验正确性。

首先是用校验的函数(提前封装好方便调用)：

```
bool checkrow(int row)
{
    int check[9] = {0};
    for (int i = 0; i < 9; i++)
    {
        if (check[A[row][i] - 1] == 1)
        {
            flag = false;
            return false;
        }
        else
        {
            check[A[row][i] - 1] = 1;
        }
    }
    return true;
}
bool checkcol(int col)
{
    int check[9] = {0};
    for (int i = 0; i < 9; i++)
    {
        if (check[A[i][col] - 1] == 1)
        {
            flag = false;
            return false;
        }
```

```
        else
        {
            check[A[i][col] - 1] = 1;
        }
    }
    return true;
}

bool checkgrid(int row, int col)
{
    int check[9] = {0};
    for (int i = row; i < row + 3; i++)
    {
        for (int j = col; j < col + 3; j++)
        {
            if (check[A[i][j] - 1] == 1)
            {
                flag = false;
                return false;
            }
            else
            {
                check[A[i][j] - 1] = 1;
            }
        }
    }
    return true;
}
```

其次是用于传递给线程的函数指针：

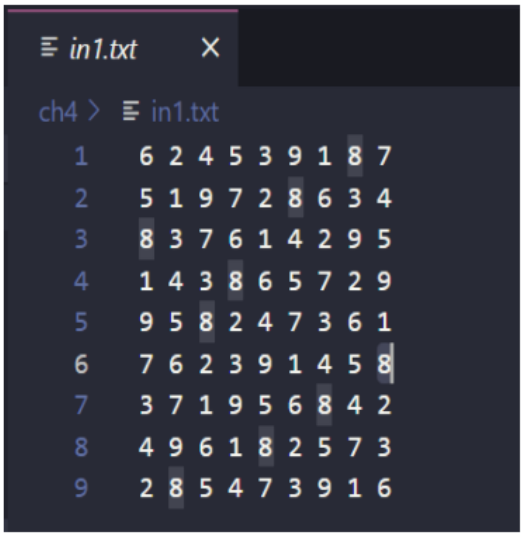
```
//检查数独的所有行的线程函数
void *checkrow_thread(void *param)
{
    int row = (int)param;
    if (checkrow(row))
    {
        printf("\033[1;32mrow %d is legal\033[0m\n", row);
    }
    else
```

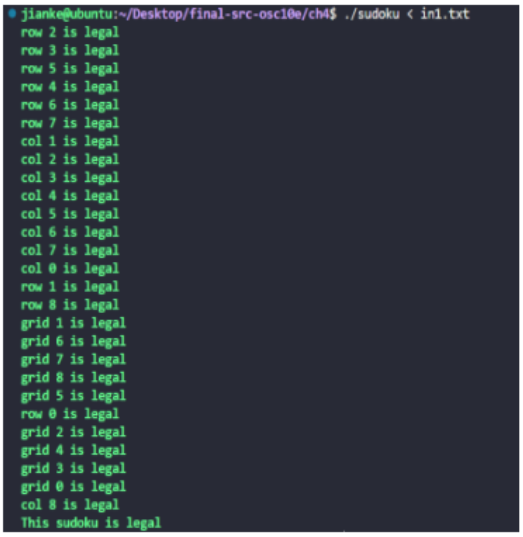
```
    {
        printf("\033[1;31mrow %d is illegal\033[0m\n", row);
    }
    pthread_exit(0);
}
//检查数独的所有列的线程函数
void *checkcol_thread(void *param)
{
    int col = (int)param;
    if (checkcol(col))
    {
        printf("\033[1;32mcol %d is legal\033[0m\n", col);
    }
    else
    {
        printf("\033[1;31mcol %d is illegal\033[0m\n", col);
    }
    pthread_exit(0);
}
//检查数独的所有九宫格的线程函数
void *checkgrid_thread(void *param)
{
    int grid = (int)param;
    int row = grid / 3 * 3;
    int col = grid % 3 * 3;
    if (checkgrid(row, col))
    {
        printf("\033[1;32mgrid %d is legal\033[0m\n", grid);
    }
    else
    {
        printf("\033[1;31mgrid %d is illegal\033[0m\n", grid);
    }
    pthread_exit(0);
}
```

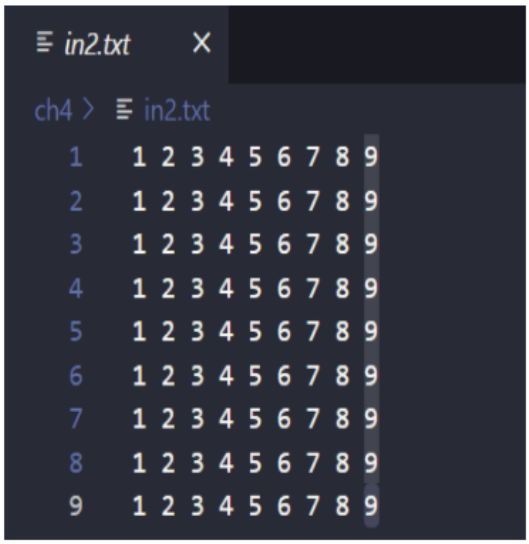
在`main`函数中，使用`pthread_create()`创建子线程并调用`pthread_join()`等待线程结束：

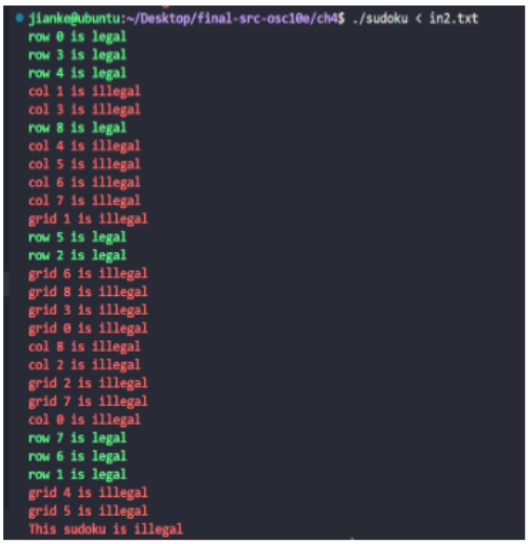
```
pthread_t tid[27];
pthread_attr_t attr;
pthread_attr_init(&attr);
//创建检查数独的所有行的线程
for (int i = 0; i < 9; i++)
{
    pthread_create(&tid[i], &attr, checkrow_thread, (void *)i);
}
//创建检查数独的所有列的线程
for (int i = 0; i < 9; i++)
{
    pthread_create(&tid[i + 9], &attr, checkcol_thread, (void
*)i);
}
//创建检查数独的所有九宫格的线程
for (int i = 0; i < 9; i++)
{
    pthread_create(&tid[i + 18], &attr, checkgrid_thread, (void
*)i);
}
//等待所有线程结束
for (int i = 0; i < 27; i++)
{
    pthread_join(tid[i], NULL);
}
```

运行结果如下:









使用pthread库进行多线程排序

定义两个函数指针 `sort()` 和 `merge()`，分别用于做子数组的排序和最终的归并

```
void *sort(void *args)
{
    struct pthread_sort *arg = (struct pthread_sort *)args;
    int start = arg->start;
    int end = arg->end;
    qsort(arr + start, end - start + 1, sizeof(int), compare);
    // for (int i = start; i <= end; i++)
    // {
    //     printf("%d ", arr[i]);
    // }
    // printf("\n");
```

```
12      pthread_exit(NULL);
13  }
14
15  void *merge(void *args)
16  {
17      struct pthread_sort *arg = (struct pthread_sort *)args;
18      int start = arg->start;
19      int end = arg->end;
20      int mid = (start + end + 1) / 2;
21      int i = start, j = mid + 1, k = start;
22      while (i <= mid && j <= end)
23      {
24          if (arr[i] < arr[j])
25          {
26              sorted[k++] = arr[i++];
27          }
28          else
29          {
30              sorted[k++] = arr[j++];
31          }
32      }
33      while (i <= mid)
34      {
35          sorted[k++] = arr[i++];
36      }
37      while (j <= end)
38      {
39          sorted[k++] = arr[j++];
40      }
41      pthread_exit(NULL);
42  }
```

使用 `pthread_create()` 和 `pthread_join()` 完成多线程排序，在这里由于 `merge` 需要等待两个排序的子线程完成后才能进行，因此需要先执行排序线程的 `join` 再进行 `merge`

```
pthread_t tid[3];
for (int i = 0; i < 2; i++)
{
    pthread_create(&tid[i], NULL, sort, &args[i]);
}
for (int i = 0; i < 2; i++)
{
    pthread_join(tid[i], NULL);
}
pthread_create(&tid[2], NULL, merge, &args[2]);
pthread_join(tid[2], NULL);
```

运行结果:

```
jianke@ubuntu:~/Desktop/final-src-osc10e/ch4$ ./mergesort
Enter the number of elements:
 6
Enter the elements:
1 5 2 4 3 7
After multithreaded sort, the sorted array is:
1 2 3 4 5 7
```

java fork-join API实现mergesort和quicksort

quicksort

使用 `java` 实现 `quicksort` 类。在fork/join框架中实现排序，任务被分割成更小的子任务，每个子任务都需要实现 `compute()` 方法来执行具体的任务。在 `quicksort` 类中，我们定义的 `compute()` 为:

```
protected void compute() {
    if (left < right) {
        int pivotIndex = left + (right - left) / 2;
        pivotIndex = partition(pivotIndex);
        invokeAll(new quicksort(data, left, pivotIndex - 1),
                  new quicksort(data, pivotIndex + 1, right));
    }
}
```

按照快速排序的逻辑，选定分割点后执行 `partition()` 将数组分为两个子数组，之后调用 `invokeAll()` 并行地执行左右分区的快速排序。

在 `main` 函数中，我们新建一个线程池，之后调用 `invoke()` 进行并行排序即可：

```
ForkJoinPool pool = new ForkJoinPool();
quicksort sorter = new quicksort(data, 0, data.length - 1);
pool.invoke(sorter);
```

运行结果如下：

```
jianke@ubuntu:~/Desktop/final-src-osc10e/ch4$ javac quicksort.java
jianke@ubuntu:~/Desktop/final-src-osc10e/ch4$ java quicksort
Enter the size of the array:
7
Enter the elements of the array:
1 7 6 4 5 3 9
The sorted array is:
1 3 4 5 6 7 9
```

mergesort

与 `quicksort` 方法基本相同，只需更改实现逻辑

```
protected void compute() {
    if (left < right) {
        if (right - left < threshold) {
            insertionsort(data, left, right);
        } else {
            int mid = left + (right - left) / 2;
            invokeAll(new mergesort(data, left, mid, threshold),
                      new mergesort(data, mid + 1, right, threshold));
            merge(data, left, mid, right);
        }
    }
}
```

对于 `compute()` 函数，我们定义一个常量 `threshold`，当要排序的数组小于这个常量时，不再继续分割子任务而是调用 `insertionsort()` 直接进行插入排序，否则就将数组分割为两部分，调用 `invokeAll()` 并行执行归并排序，并将排序好的数组进行 `merge` 得到最终结果。

`main()` 函数中实现如下：

```
1  ForkJoinPool pool = new ForkJoinPool();
2  mergesort sorter = new mergesort(data, 0, data.length - 1, 4);
3  pool.invoke(sorter);
```

运行结果如下：

```
jianke@ubuntu:~/Desktop/final-src-osc10e/ch4$ javac mergesort.java
jianke@ubuntu:~/Desktop/final-src-osc10e/ch4$ java mergesort
Enter the size of the array:
7
Enter the elements of the array:
1 5 7 4 2 6 3
The sorted array is:
1 2 3 4 5 6 7
```