快速排序问题

一、实验环境

使用go语言实现，版本为go1.24.3 windows/amd64，在windows11环境下进行实验。

二、实验目的

1. 通过对进程间高级通信问题的编程实现，加深理解进程间高级通信的原理；
2. 对 Windows 或 Linux 涉及的几种高级进程间通信机制有更进一步的了解；
3. 熟悉 Windows 或 Linux 中定义的与高级进程间通信有关的函数。

三、实验原理

首先生成随机数文件，随后读取数据，创建线程进行快速排序。当排序的数据量较大时，程序会试图生成一个新的线程来进行并行排序，当数据量较小时直接进行串行排序，每个串行排序占用一个线程，排序完成后释放资源，使用一个常量`maxWorkers`来限定线程的数量。

排序完成后将数据保存在新文件中，验证通过后即完成实验。

并行处理：

- 使用`minSize`参数控制小于该值的子数组不再分割（代码中未显示具体值，应该设置为1000）
- 使用`maxWorkers`参数限制并行线程数（控制在20个左右）
- 每次分区后，尝试为左半部分创建新线程，右半部分在当前线程处理

通信机制：

通过共享内存方式实现，所有线程直接访问同一个数组（Go语言的切片）

线程同步：

实现：

- 使用`sync.WaitGroup`机制确保所有排序任务完成
- 使用`workerMutex`互斥锁保护`activeWorkers`计数
- 快速排序算法的特性保证了不同线程操作数组的不同部分，无需额外锁

四、项目结构

main.go：程序入口

serialQuickSort.go：串行快速排序函数

parallelQuickSort.go：并行快速排序函数

randNumFile：随机数文件的生成，写入与读取

utils.go：验证文件是否排序、检查文件是否存在

1.部分常量定义

```
const (
    totalNumbers = 1000000 // 随机数个数
    minSize      = 1000 // 串行排序的最大输入
    maxWorkers   = 20 // 最大线程数
    filename     = "random_numbers.txt"
    sortedFile   = "sorted_numbers.txt"
)
```

2.随机数生成

```
// 生成随机数文件
func generateRandomFile() error {
    file, err := os.Create(filename)
    if err != nil {
        return err
    }
    defer file.Close()

    writer := bufio.NewWriter(file)
    for i := 0; i < numCount; i++ {
        _, err := writer.WriteString(strconv.Itoa(rand.Intn(maxNum)) + "\n")
        if err != nil {
            return err
        }
    }
    return writer.Flush()
}
```

3.串行快速排序实现

```
// 串行快速排序的实现
func serialQuickSort(arr []int, low, high int) {
    if low < high {
        pivotIndex := partition(arr, low, high)
        serialQuickSort(arr, low, pivotIndex-1)
        serialQuickSort(arr, pivotIndex+1, high)
    }
}

// 快速排序的分区函数
func partition(arr []int, low, high int) int {
    // 随机选择一个元素作为pivot，避免最坏情况
    pivotIdx := low + rand.Intn(high-low+1)
    arr[pivotIdx], arr[high] = arr[high], arr[pivotIdx]

    // 选择最后一个元素为pivot
    pivot := arr[high]

    // i 是小于pivot的元素应该放的位置
    i := low - 1

    for j := low; j < high; j++ {
        if arr[j] < pivot {
            i++
            arr[i], arr[j] = arr[j], arr[i]
        }
    }
    arr[i+1], arr[high] = arr[high], arr[i+1]
    return i + 1
}
```

4.并行快速排序实现

```
// 并行快速排序函数
func parallelQuickSort(arr []int, low, high int, wg *sync.WaitGroup) {
    // 如果数据量小，使用串行快速排序
    if high-low < minSize {
        serialQuickSort(arr, low, high)
        return
    }

    if low < high {
        // 分区，获取分区点
        pivot := partition(arr, low, high)

        // 检查是否有可用的工作线程来并行处理左半部分
        var canParallel bool

        // 检查是否可以创建新的工作线程
```

```
        workerMutex.Lock()
        if activeWorkers < maxWorkers {
            activeWorkers++
            canParallel = true
        }
        workerMutex.Unlock()

        if canParallel {
            // 并行处理左半部分
            wg.Add(1)
            go func() {
                defer wg.Done()
                parallelQuickSort(arr, low, pivot-1, wg)

                // 工作完成后减少活动工作线程计数
                workerMutex.Lock()
                activeWorkers--
                workerMutex.Unlock()
            }()

            // 当前线程处理右半部分
            parallelQuickSort(arr, pivot+1, high, wg)
        } else {
            // 在当前线程中顺序处理两部分
            parallelQuickSort(arr, low, pivot-1, wg)
            parallelQuickSort(arr, pivot+1, high, wg)
        }
    }
}
```

1. **任务分割**：对于每个分区，算法会尝试创建一个新线程处理左半部分，而当前线程处理右半部分。
2. **线程数量控制**：
   - 设置了`minSize`阈值，当子数组大小小于此阈值时，不再并行处理而是采用串行排序
   - 设置了`maxWorkers`限制，控制最大活动工作线程数量在约20个左右
3. **线程同步**：
   - 使用`sync.WaitGroup`确保所有排序任务完成后主程序才继续执行
   - 使用`workerMutex`互斥锁保护`activeWorkers`计数，防止并发修改

5.主程序流程

```
func main() {
    rand.Seed(time.Now().UnixNano())

    // 生成随机数文件
    if err := generateRandomFile(); err != nil {
        fmt.Println("生成随机数文件时出错:", err)
        return
    }
    fmt.Println("随机数文件已准备")
```

```
    // 读取数据文件
    data, err := readDataFromFile()
    if err != nil {
        fmt.Println("读取数据文件时出错:", err)
        return
    }
    fmt.Printf("已读取 %d 个数字\n", len(data))

    // 开始排序计时
    startTime := time.Now()
    fmt.Println("开始并行排序...")

    // 创建排序数据的副本并排序
    sortedData := make([]int, len(data))
    copy(sortedData, data)

    // 创建一个等待组，用于等待所有排序任务完成
    var wg sync.WaitGroup
    wg.Add(1)

    // 开始并行快速排序
    go func() {
        defer wg.Done()
        // 获取工作线程锁并增加活动工作线程计数
        workerMutex.Lock()
        activeWorkers++
        workerMutex.Unlock()

        // 启动递归并行排序
        parallelQuickSort(sortedData, 0, len(sortedData)-1, &wg)

        // 完成后减少活动工作线程计数
        workerMutex.Lock()
        activeWorkers--
        workerMutex.Unlock()
    }()

    // 等待所有排序任务完成
    wg.Wait()

    // 结束计时并打印信息
    duration := time.Since(startTime)
    fmt.Printf("排序完成，耗时: %v\n", duration)

    // 将排序结果写入文件
    if err := writeDataToFile(sortedData); err != nil {
        fmt.Println("写入排序结果时出错:", err)
        return
    }

    // 验证排序结果
    if isSorted(sortedData) {
```

```
        fmt.Println("排序结果验证: 成功! 数据已正确排序")
    } else {
        fmt.Println("排序结果验证: 失败! 数据未正确排序")
    }
}
```

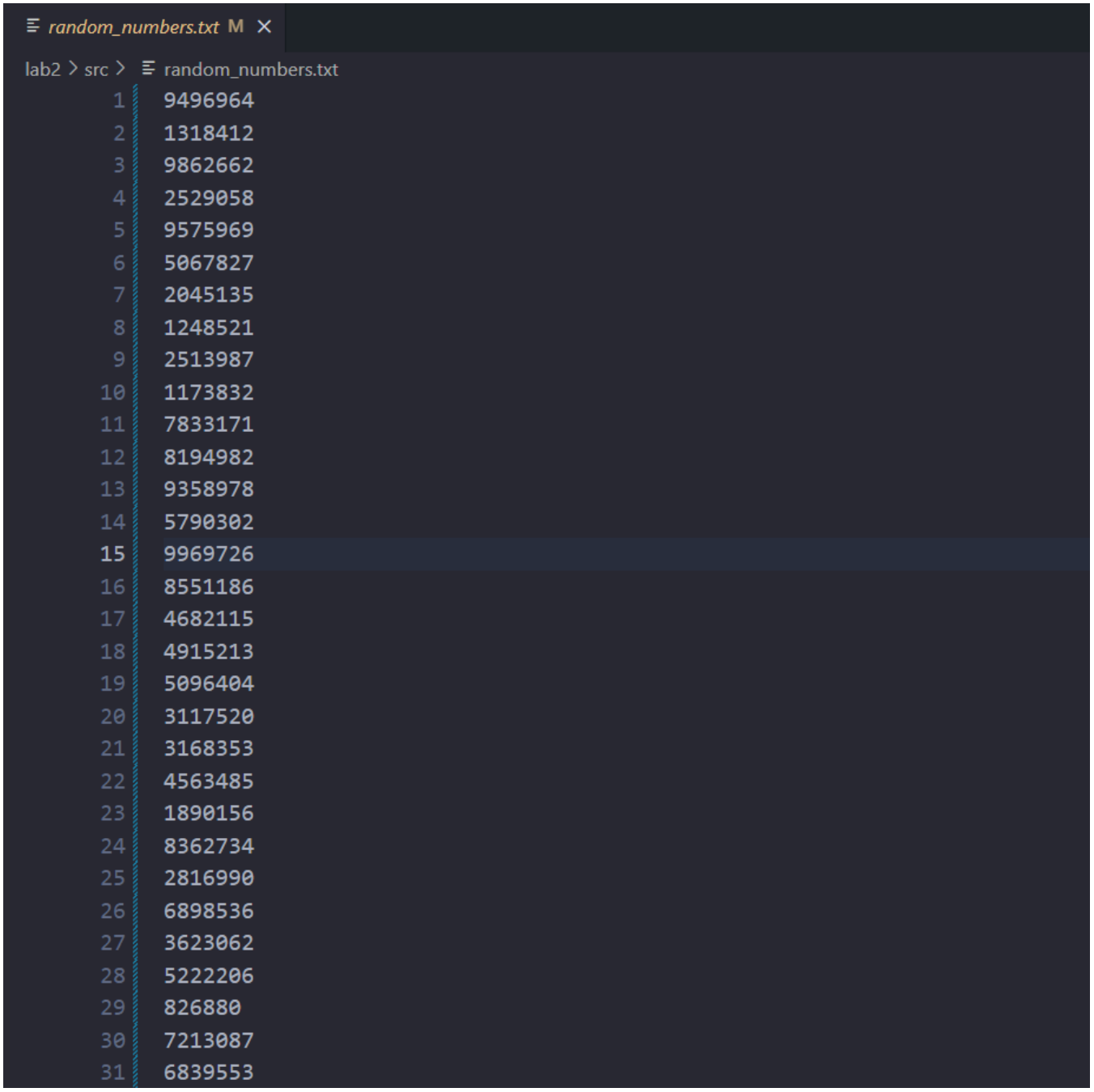
五、项目运行

切换到 lab2/src 目录下，执行 `go run .` 来启动项目：

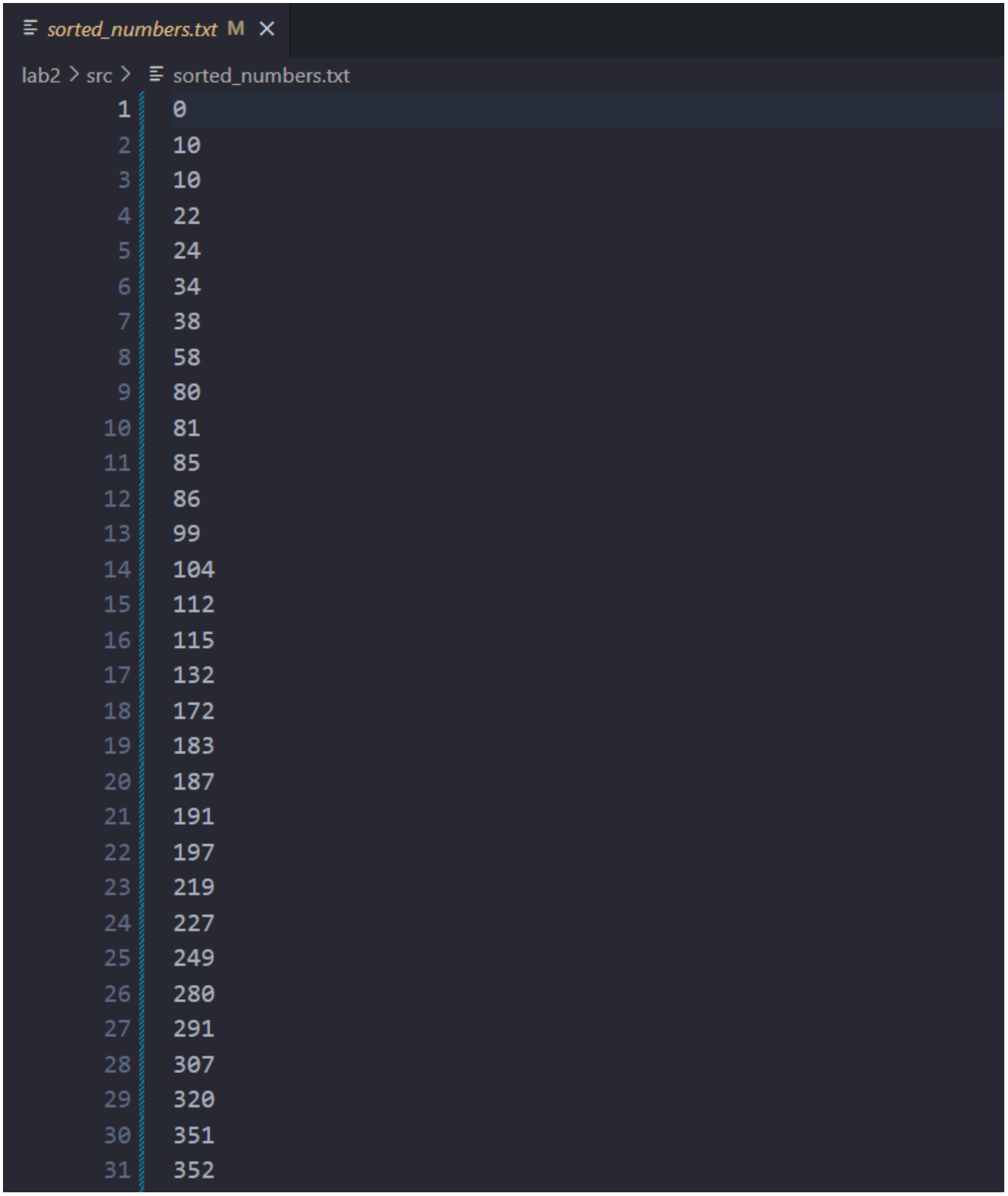
```
(base) D:\CODE\THUEE-OS-lab [master +4 ~2 -0 !]> cd .\lab2\src\
(base) D:\CODE\THUEE-OS-lab\lab2\src [master +4 ~2 -0 !]> go run .
随机数文件已准备
已读取 1000000 个数字
开始并行排序...
排序完成, 耗时: 16.0713ms
排序结果已保存至文件: sorted_numbers.txt
排序结果验证: 成功! 数据已正确排序
(base) D:\CODE\THUEE-OS-lab\lab2\src [master +4 ~2 -0 !]>
```

可以看见`random_numbers.txt`和`sorted_numbers.txt`已经生成在了同一目录下。

打开`random_numbers.txt`，可以看到未排序的随机数：



打开`sorted_numbers.txt`，可以看到排序后的结果：



排序结果通过验证，证明已经成功完成排序。

六、思考题解答

1.你采用了你选择的机制而不是另外的两种机制解决该问题，请解释你做出这种选择的理由。

在本实验中，选择使用共享内存机制而非管道或消息队列，主要基于以下考虑：

数据访问效率：

- **直接访问**: 共享内存允许所有线程直接访问同一块内存区域（数组），无需数据复制和传输操作，特别适合快速排序这类需要大量读写数组的场景
- **低延迟**: 避免了数据传输过程中的序列化/反序列化和系统调用开销，降低了线程间通信延迟

问题特性契合度：

- **分而治之**: 快速排序算法天然具有"分而治之"的特性，不同线程操作数组的不同部分，几乎不需要交换数据
- **原地排序**: 快速排序是原地排序算法，直接在原数组上操作，共享内存模式与此高度契合

2.你认为另外的两种机制是否同样可以解决该问题？如果可以请给出你的思路；如果不能，请解释理由。

1.管道机制

可行性: 可以实现

实现思路:

1. 创建主管道用于发送排序任务，每个任务包含数组区间信息
2. 创建结果管道用于接收已排序的子数组
3. 工作流程:
    - 主线程初始时将整个数组区间作为任务发送到任务管道
    - 工作线程从任务管道接收任务，进行分区操作
    - 对于小数组，工作线程直接排序并将结果放入结果管道
    - 对于大数组，分区后创建新任务并发送回任务管道
    - 主线程合并所有子数组结果

2.消息队列

可行性: 可以实现

实现思路:

1. 创建任务消息队列和结果消息队列
2. 任务消息包含需要排序的数组区间信息和原始数据
3. 工作线程从队列获取任务，排序后将结果发送到结果队列
4. 主线程负责分配任务和合并结果