姓名	吴宇贤
学号	2021214266

实验成绩	

华中师范大学计算机科学系

实 验 报 告 书

实验题目：作业调度算法的实现

课程名称：操作系统原理

主讲教师：朱瑄

辅导教师：

课程编号：4872801

班　　级：3 课堂

实验时间：2023 年 10 月 11 日

一、实验目的：

1、 掌握作业调度的基本思想

2、 熟练使用各种作业调度算法描述的过程

3、 掌握各种算法的优缺点

4、 提高理论和实践结合的能力

二、实验内容：

1、 先来先服务算法

2、 最短作业优先算法

3、 最高响应比优先算法

三、实验环境：

实践平台：MacOS

编写环境：Goland

编译器： go

四、实验设计原理

1、 先来先服务算法

2、最短作业算法

3、最高响应比优先算法

五、实验详细实现过程与算法流程

1、 先来先服务

创建一个队列（或者使用一个数组），用于存储待调度的作业。将作业按照它们到达的顺序依次加入队列，首先到达的作业排在队列的前面。从队列头部取出作业，将其分配给 CPU 执行。

当作业完成执行后，从队列头部取出下一个作业继续执行，依此类推，直到队列中的所有作业都完成。

2、最短作业优先算法

这里设置一个优先队列，第一个作业先进入队列，第一个作业出队列，根据出队的作业的结束时间扫描能够满足条件的作业进入队列，重复如此操作直到作业调度完毕，在这个过程中，出队的作业，就能根据前一个作业的信息推算出当前作业的信息。

3、最高响应比优先算法

① 创建一个队列来存储待执行的作业。

② 对于每个作业，计算其响应比（Response Ratio），响应比定义为：1+(等待时间/ 作业时间）。等待时间是指从到达时间开始等待执行的时间。

③ 将作业按照它们的响应比降序排序，即响应比最高的作业排在前面。

④ 初始化一个时间计数器，开始执行作业。

⑤ 依次从队列中取出响应比最高的作业，执行该作业，更新时间计数器。

⑥ 检查队列中是否有新的作业到达，如果有，计算它们的响应比，并将它们添加到队列中。

⑦ 重复执行步骤 5 和 6，直到队列为空，即所有作业都已经执行完

六、源程序（加注释）

//作业调度程序

```
package main

import (
    "fmt"
    "sort"
)

type Job struct {
    JobNumber               int
    ArrivalTime             int
    EstimatedTime           int
    StartTime               int
    EndTime                 int
    TurnaroundTime          int
    WeightedTurnaroundTime float64
    ResponseRatio           float64
}

// 计算时间增加
func TimeAdd(initialTime int, addedTime int) int {
    hour := initialTime / 100
    minute := initialTime % 100
    new_minute := (addedTime + minute) % 60
    new_hour := (hour + (addedTime+minute)/60) % 24
    return 100*new_hour + new_minute
}

// 计算时间相减
func TimeSub(subbedTime int, initialTime int) int {
    ori_hour := initialTime / 100
    ori_minute := initialTime % 100
    subbed_hour := subbedTime / 100
    subbed_minute := subbedTime % 100
    return subbed_hour*60 + subbed_minute - ori_hour*60 -
ori_minute
}
```

```
// 计算响应比
func CalculateResponseRatio(startTime int, arrivalTime int,
estimatedTime int) float64 {
    return 1 + (float64(TimeSub(startTime,
arrivalTime)))/float64(estimatedTime)
}

func PrintTitle() {
    fmt.Println("作业\t\t进入时间\t估计运行时间\t开始时间\t结束时间\t周
转时间(分钟)\t带权周转时间\t")
}

func PrintJob(jobs []Job) {
    for _, m := range jobs {
        fmt.Printf("JOB%d\t", m.JobNumber)
        PrintTime(m.ArrivalTime)
        fmt.Printf("\t    %d \t ", m.EstimatedTime)
        PrintTime(m.StartTime)
        PrintTime(m.EndTime)
        fmt.Printf("\t\t%d\t", m.TurnaroundTime)
        fmt.Printf("\t%.2f\t", m.WeightedTurnaroundTime)
        fmt.Printf("\n")
    }
}

func PrintTime(originalTime int) {
    fmt.Printf("\t%02d:%02d\t", originalTime/100,
originalTime%100)
}

func main() {
    //var jobs []Job
    //var num int
    //fmt.Printf("输入作业个数:")
    //fmt.Scan(&num)
    //for i := 0; i < num; i++ {
```

```
    // var job Job
    // fmt.Scan(&job.JobNumber, &job.ArrivalTime,
&job.EstimatedTime)
    // jobs = append(jobs, job)
    //}
    jobs := []Job{
        {1, 800, 50, 0, 0, 0, 0, 0},
        {2, 815, 30, 0, 0, 0, 0, 0},
        {3, 830, 25, 0, 0, 0, 0, 0},
        {4, 835, 20, 0, 0, 0, 0, 0},
        {5, 845, 15, 0, 0, 0, 0, 0},
        {6, 900, 10, 0, 0, 0, 0, 0},
        {7, 920, 5, 0, 0, 0, 0, 0},
    }

    // 使用先来先服务（FIFO）算法进行作业调度
    fmt.Println("-----------------------------------------------
---先进先出(FIFO)调度算法-----------------------------------------
--------")
    PrintTitle()
    scheduleFIFO(jobs)

    // 使用短作业优先（SJF）算法进行作业调度
    fmt.Println("\n---------------------------------------------
----短作业优先(SJF)调度算法---------------------------------------
---------")
    PrintTitle()
    scheduleSJF(jobs)

    // 使用最高响应比优先（HRRN）算法进行作业调度
    fmt.Println("\n---------------------------------------------
--最高响应比优先(HRRN)调度算法-----------------------------------
---------")
    PrintTitle()
    scheduleHRRN(jobs)

}
```

```
func scheduleFIFO(jobs []Job) {
    copiedJobs := make([]Job, len(jobs))
    copy(copiedJobs, jobs)
    for i, _ := range copiedJobs {
       if i == 0 {
          copiedJobs[i].StartTime = copiedJobs[i].ArrivalTime
       } else {
          copiedJobs[i].StartTime = copiedJobs[i-1].EndTime
       }
       copiedJobs[i].EndTime = TimeAdd(copiedJobs[i].StartTime,
copiedJobs[i].EstimatedTime)
       copiedJobs[i].TurnaroundTime =
TimeSub(copiedJobs[i].EndTime, copiedJobs[i].ArrivalTime)
       copiedJobs[i].WeightedTurnaroundTime =
float64(copiedJobs[i].TurnaroundTime) /
float64(copiedJobs[i].EstimatedTime)
    }
    PrintJob(copiedJobs)
}

func scheduleSJF(jobs []Job) {
    // 第一个作业先执行
    jobs[0].StartTime = jobs[0].ArrivalTime
    jobs[0].EndTime = TimeAdd(jobs[0].ArrivalTime,
jobs[0].EstimatedTime)
    jobs[0].TurnaroundTime = jobs[0].EstimatedTime
    jobs[0].WeightedTurnaroundTime = 1
    copiedJobs := make([]Job, len(jobs))
    copy(copiedJobs, jobs)
    currentTime := jobs[0].EndTime
    scheduledJobs := []Job{}
    for len(copiedJobs) > 0 {
       availableJobs := []Job{}
       // 已经到达的
       for _, job := range copiedJobs {
          if job.ArrivalTime <= currentTime {
```

```
            availableJobs = append(availableJobs, job)
        }
    }
    if len(availableJobs) == 0 {
        // 如果没有可用作业，则将当前时间递增到下一个作业的到达时间
        currentTime = copiedJobs[0].ArrivalTime
        continue
    }
    sort.Slice(availableJobs, func(i, j int) bool {
        return availableJobs[i].EstimatedTime <
availableJobs[j].EstimatedTime
    })
    // 调度估计运行时间最短的作业
    shortestJob := availableJobs[0]
    scheduledJobs = append(scheduledJobs, shortestJob)
    // 更新当前时间并从列表中移除已调度的作业
    currentTime = TimeAdd(currentTime,
shortestJob.EstimatedTime)
    for i, job := range copiedJobs {
        if job.JobNumber == shortestJob.JobNumber {
            copiedJobs = append(copiedJobs[:i],
copiedJobs[i+1:]...)
            break
        }
    }
}

for i, _ := range scheduledJobs {
    if scheduledJobs[i].JobNumber != 1 {
        if i == 0 {
            scheduledJobs[i].StartTime = jobs[0].EndTime
        } else {
            if scheduledJobs[i-1].JobNumber == 1 {
                scheduledJobs[i].StartTime =
scheduledJobs[i-2].EndTime
            } else {
```

```
                scheduledJobs[i].StartTime =
scheduledJobs[i-1].EndTime
            }
        }
        scheduledJobs[i].EndTime =
TimeAdd(scheduledJobs[i].StartTime,
scheduledJobs[i].EstimatedTime)
        scheduledJobs[i].TurnaroundTime =
TimeSub(scheduledJobs[i].EndTime, scheduledJobs[i].ArrivalTime)
        scheduledJobs[i].WeightedTurnaroundTime =
float64(scheduledJobs[i].TurnaroundTime) /
float64(scheduledJobs[i].EstimatedTime)
      }
   }
   sort.Slice(scheduledJobs, func(i, j int) bool {
      return scheduledJobs[i].JobNumber <
scheduledJobs[j].JobNumber
   })
   PrintJob(scheduledJobs)
   return
}

func scheduleHRRN(jobs []Job) {
   jobs[0].StartTime = jobs[0].ArrivalTime
   jobs[0].EndTime = TimeAdd(jobs[0].ArrivalTime,
jobs[0].EstimatedTime)
   jobs[0].TurnaroundTime = jobs[0].EstimatedTime
   jobs[0].WeightedTurnaroundTime = 1
   jobs[0].ResponseRatio = 1
   copiedJobs := make([]Job, len(jobs))
   copy(copiedJobs, jobs)
   currentTime := jobs[0].ArrivalTime
   scheduledJobs := []Job{}

   for len(copiedJobs) > 0 {
      availableJobs := []Job{}
      // 已经到达的
```

```
        for _, job := range copiedJobs {
            if job.ArrivalTime <= currentTime {
                availableJobs = append(availableJobs, job)
            }
        }

        if len(availableJobs) == 0 {
            // 如果没有可用作业，则将当前时间递增到下一个作业的到达时间
            currentTime = copiedJobs[0].ArrivalTime
            continue
        } else {
            for i, _ := range availableJobs {
                if availableJobs[i].JobNumber == 1 {
                    availableJobs[i].ResponseRatio = 0
                } else {
                    lastestIndex := len(scheduledJobs) - 1
                    availableJobs[i].ResponseRatio =
CalculateResponseRatio(scheduledJobs[lastestIndex].EndTime,
availableJobs[i].ArrivalTime, availableJobs[i].EstimatedTime)
                }
            }
        }
        sort.Slice(availableJobs, func(i, j int) bool {
            return availableJobs[i].ResponseRatio >
availableJobs[j].ResponseRatio
        })

        // 调度响应比高的
        respJob := availableJobs[0]
        if respJob.JobNumber != 1 {
            lastIndex := len(scheduledJobs) - 1
            respJob.StartTime = scheduledJobs[lastIndex].EndTime
            respJob.EndTime = TimeAdd(respJob.StartTime,
respJob.EstimatedTime)
            respJob.TurnaroundTime = TimeSub(respJob.EndTime,
respJob.ArrivalTime)
```

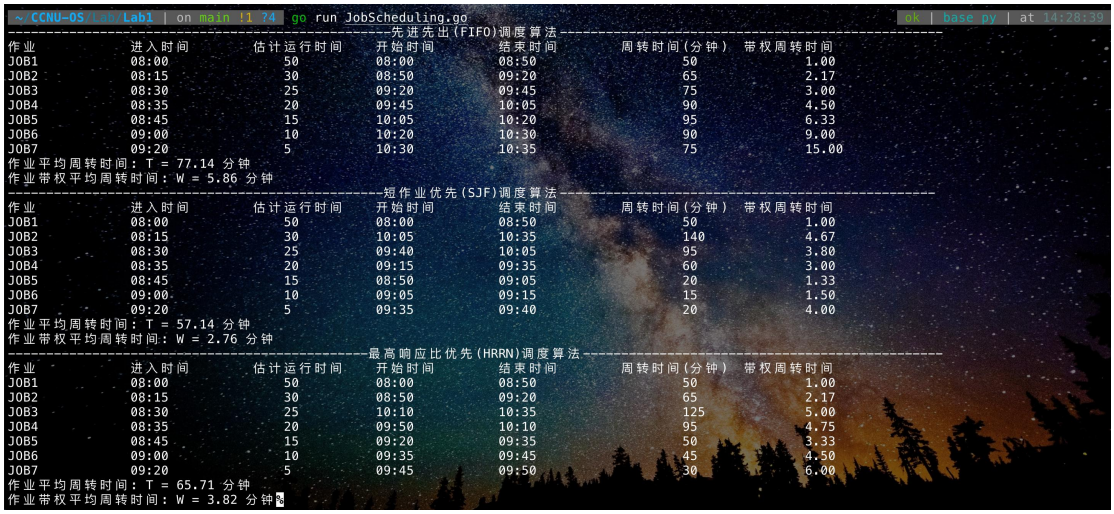
```
            respJob.WeightedTurnaroundTime =
float64(respJob.TurnaroundTime) / float64(respJob.EstimatedTime)
        }
        scheduledJobs = append(scheduledJobs, respJob)
        // 更新当前时间并从列表中移除已调度的作业
        currentTime = TimeAdd(currentTime, respJob.EstimatedTime)

        for i, job := range copiedJobs {
            if job.JobNumber == respJob.JobNumber {
                copiedJobs = append(copiedJobs[:i],
copiedJobs[i+1:]...)
                break
            }
        }

    }
    sort.Slice(scheduledJobs, func(i, j int) bool {
        return scheduledJobs[i].JobNumber <
scheduledJobs[j].JobNumber
    })
    PrintJob(scheduledJobs)
    return
}
```

七、代码运行结果

实验输入数据：

1 800 50
2 815 30
3 830 25
4 835 20
5 845 15
6 900 10
7 920 5

作业调度结果：



八、实验结果分析

从实验结果数据分析三种作业调度算法的性能

1.先进先出（FIFO）调度算法

平均周转时间（T）：77.14 分钟

带权平均周转时间（W）：5.86 分钟

FIFO 算法按照作业到达的顺序执行作业，没有考虑作业的运行时间。这导致了较长的平均周转时间和较高的带权周转时间。在此算法下，先到达的作业等待时间较长，因此性能较差。

2. 短作业优先（SJF）调度算法

平均周转时间（T）：57.14 分钟

带权平均周转时间（W）：2.76 分钟

SJF 算法根据作业的估计运行时间来选择下一个要执行的作业，以最小化周转时间。这导致较短的平均周转时间和较低的带权周转时间。在此算法下，短作业被优先执行，性能较好。

3. 最高响应比优先（HRRN）调度算法

平均周转时间（T）：65.71 分钟

带权平均周转时间（W）：3.82 分钟

HRRN 算法考虑了等待时间和估计运行时间的比例来选择下一个要执行的作业，以最大程度地减少响应时间。这导致了中等长度的平均周转时间和带权周转时间。在此算法下，响应时间相对较好，但不如 SJF 算法。

FIFO 算法表现最差，导致了较长的平均周转时间和较高的带权周转时间。

九、实验改进意见与建议（可以包括编码的心得体会）

在编写程序的过程中，我注意到了代码的结构和可读性非常重要。通过将代码分解成小函数、使用描述性变量名以及添加注释，使代码更易理解和维护。

虽然作业调度的算法并不复杂，但是在算法实现方面，我遇到了一些问题，先进先服务比较容易实现，但是在后两个算法的实现时，我一开始没有使用队列，并且没有考虑到作业是否在上一个作业结束前到达，导致作业执行时间混乱。在重新阅读课本后，我调整思路，最终实现了短作业优先和最高响应比优先两种算法。

改进意见与建议：

在实际应用中，作业调度可能需要处理多个作业同时到达的情况。可以考虑在算法中添加并行性支持，以提高效率。