计算机科学与技术学院课程设计成绩单

课程名称：操作系统课程设计

姓名	何朝晖	性别	男	学号	202313407011	班级	计科2301班
电话	13479501746		综合成绩		成绩等级		
程序运行情况 （占总成绩 20%）	☐能正确运行 （18~20 分） ☐基本能正确运行 （15~17 分） ☐能运行但结果不完善（10~14 分）						
程序功能完善程度 （占总成绩 20%）	☐完善 （18~20 分） ☐基本完善 （15~17 分） ☐不完善 （10~14 分）						
对问题的答辩情况 （占总成绩 30%）	☐概念正确有创新 （27~30 分） ☐能正确回答所有问题 （23~26 分） ☐基本能正确回答 （19~22 分） ☐部分问题回答概念不清晰（15~18 分）						
学生的工作态度与独立工作能力 （占总成绩 10%）	☐工作态度认真能独立完成任务 （9~10 分） ☐工作态度基本认真,独立性尚可（7~8 分） ☐工作态度和独立性较差 （5~6 分）						
设计报告的规范性 （占总成绩 20%）	☐符合规范 （18~20 分） ☐基本符合规范 （14~17 分） ☐规范性较差 （10~13 分）						

A：90~100 分　A-：85~89 分　B+：82~84 分　B：78~81 分　B-：75~77 分
C+：72~74 分　C：68~71 分　C-：64~67 分　D：60~63 分　F：<60 分

武汉科技大学计算机科学与技术学院制表

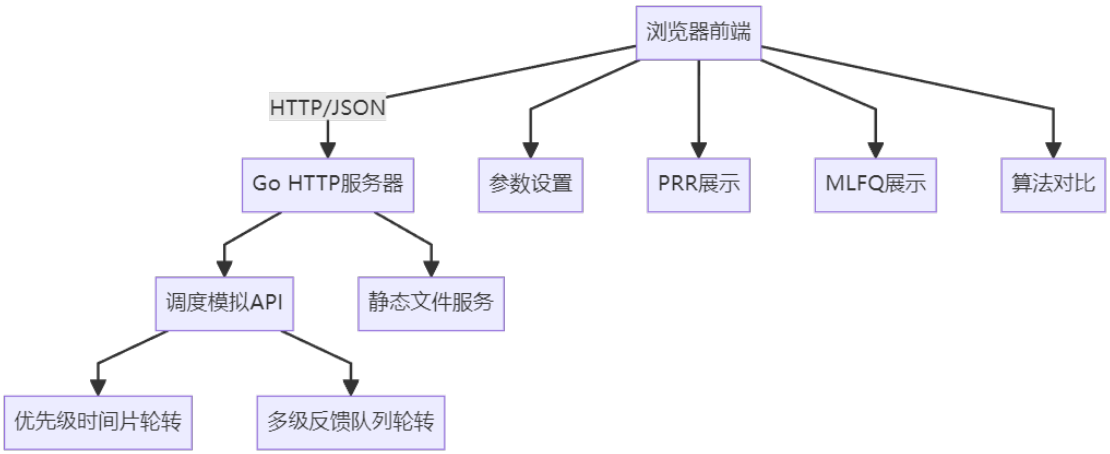
操作系统课程设计：进程调度模拟器实验报告

1. 设计目的及意义

进程调度是操作系统核心功能，负责从就绪队列中选择进程分配 CPU 资源。本课程设计通过实现交互式进程调度模拟器，深入理解进程调度原理，掌握基于优先级的时间片轮转和多级反馈队列轮转算法，通过可视化展示直观比较不同算法的优劣。

2. 系统设计

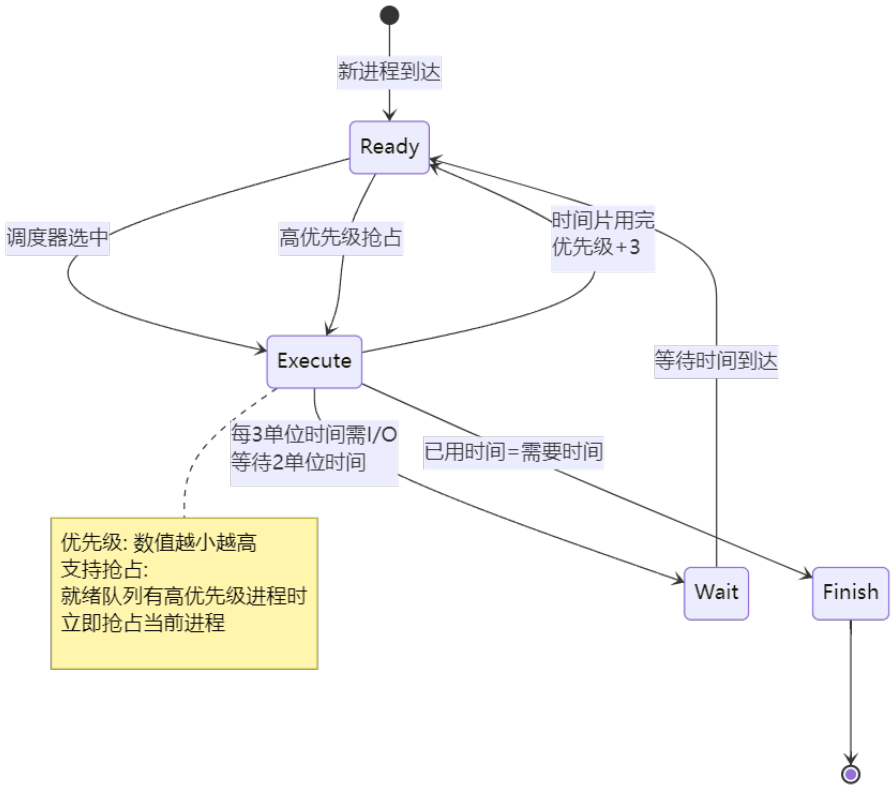
2.1 系统架构



2.2 核心数据结构

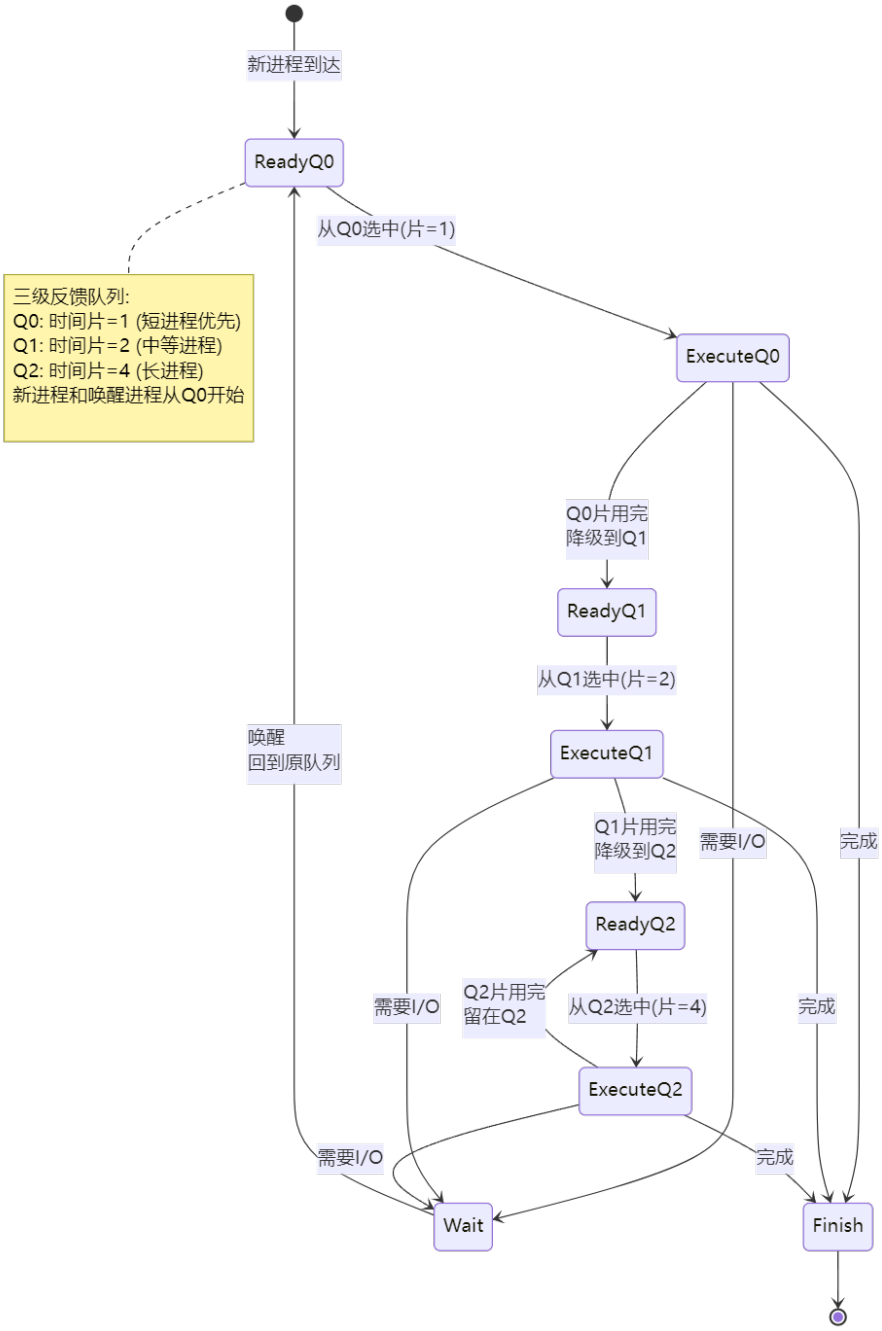
```
type PCB struct {
    Name, Priority, ArrivalTime, NeedTime int/string
    UsedTime, State, StartTime, FinishTime int/string
    QueueLevel, WaitUntil, SliceUsedTime int
}

type Snapshot struct {
    Time int
    Processes []PCB
    Running string
    Queues [][]string
    Event string
}
```

2.3 优先级时间片轮转算法流程



系统首先根据进程的优先级选择待执行任务，然后按照设定的时间片（Quantum）进行轮转（RR）执行，直到当前时间片耗尽或任务完成，再进行上下文切换，重复此流程。

2.4 多级反馈队列算法流程



新进程从最高优先级队列开始执行；若进程长时间占用 CPU，则被降级至低优先级队列；若长时间等待，则可能被提升优先级。

3. 系统实现

3.1 优先级时间片轮转核心代码

```
func SimulatePriorityRR(processes []*PCB, timeSlice int) SimResult {
    var readyQueue []*PCB
    time := 0
    for time <= maxTime {
        // 唤醒等待进程，处理新到达进程
```

```
        for _, p := range procs {
            if p.State == StateWait && p.WaitUntil <= time {
                p.State = StateReady; readyQueue = append(readyQueue, p)
            }
        }
        // 抢占检查：高优先级进程抢占当前进程
        if running != nil {
            for _, p := range readyQueue {
                if p.Priority < running.Priority {
                    running.State = StateReady
                    readyQueue = append(readyQueue, running); running = nil
                }
            }
        }
        // 按优先级选择进程执行
        if running == nil && len(readyQueue) > 0 {
            sort.Slice(readyQueue, ...) // 优先级排序
            running = readyQueue[0]; readyQueue = readyQueue[1:]
        }
        // 执行进程，记录快照
        running.UsedTime++; running.SliceUsedTime++
        if running.UsedTime >= running.NeedTime {
            running.State = StateFinish // 进程完成
        } else if running.SliceUsedTime >= timeSlice {
            running.Priority += 3 // 优先级降低
            if running.UsedTime%3 == 0 {
                running.State = StateWait; running.WaitUntil = time + 2 // I/O 等待
            }
        }
        time++
    }
}
```

3.2 多级反馈队列核心代码

```
func SimulateMLFQ(processes []*PCB, queueTimeSlices [3]int) SimResult {
    queues := [3][]*PCB{} // Q0:片=1, Q1:片=2, Q2:片=4
    time := 0
    for time <= maxTime {
        // 唤醒等待进程，新进程从 Q0 开始
        for _, p := range procs {
            if p.State == StateWait && p.WaitUntil <= time {
                p.State = StateReady; queues[p.QueueLevel] = append(..., p)
            }
            if p.ArrivalTime == time && p.State == StateReady {
                p.QueueLevel = 0; queues[0] = append(queues[0], p)
            }
        }
        // 从最高优先级非空队列选择进程
        running = selectFromMLFQ(queues)
        if running.StartTime == -1 { running.StartTime = time }
        // 执行进程，记录快照
        running.UsedTime++; running.SliceUsedTime++
```

```
        if running.UsedTime >= running.NeedTime {
            running.State = StateFinish // 进程完成
        } else if running.SliceUsedTime >= queueTimeSlices[running.QueueLevel] {
            if running.QueueLevel < 2 {
                running.QueueLevel++ // 降级到下一级队列
            }
            running.State = StateReady; running.SliceUsedTime = 0
        }
        time++
    }
}
```

3.3 前端模拟请求

```
async function doSim(algo, ts, procs) {
  const resp = await fetch('/api/simulate', {
    method:'POST',
    headers:{'Content-Type':'application/json'},
    body:JSON.stringify({algorithm:algo, timeSlice:ts, processes:procs})
  });
  return resp.json();
}
```

3.4 调度过程渲染

```
function renderStep(key) {
  const snap = state[key].data.snapshots[state[key].step];
  // 更新当前执行进程、就绪队列、状态表
  if (snap.running) {
    const rp = snap.processes.find(p=>p.name===snap.running);
    document.getElementById('running-'+key).innerHTML =
      `${rp.name} 优先级=${rp.priority} 已用=${rp.usedTime}`;
  }
  updateGantt(key, state[key].step); // 更新甘特图
}
```

4. 系统运行效果

4.1 调度流程对比

示例进程配置：P0(优先级 3/到达 0/需 5), P1(优先级 8/到达 1/需 3), P2(优先级 5/到达 2/需 6), 时间片=2

优先级时间片轮转调度序列:

时间	当前执行	就绪队列	事件
t=0	P0(3)	-	P0 开始
t=2	P2(5)	P0(6)	P0 时间片用完，被 P2 抢占
t=4	P0(6)	P2(8)	P2 时间片用完，P0 继续
t=6	P1(8)	P0(9),P2(11)	P0 再次时间片用完，P1 执行
...			

多级反馈队列调度序列:

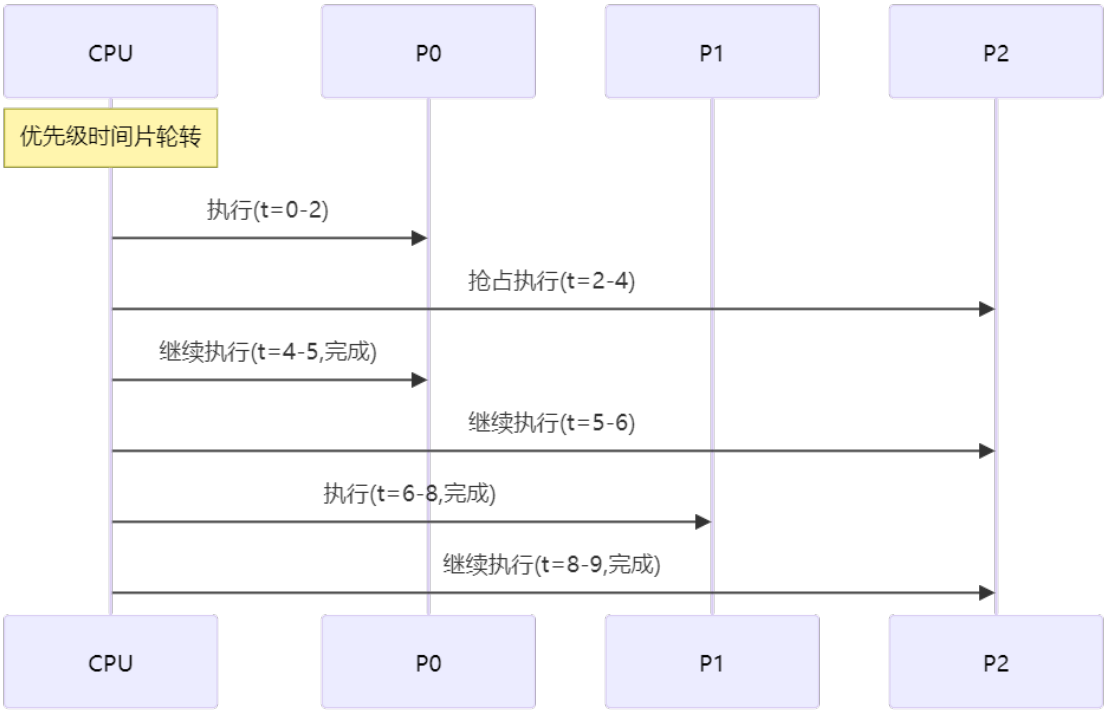
```
时间   当前执行   队列状态(Q0/Q1/Q2)        事件
t=0    P0          [P0]/[]/[]              P0 从 Q0 开始
t=1    P0(Q1)      []/[P0]/[]              P0 用完 Q0 片，降级到 Q1
t=3    P2(Q0)      [P2]/[P0]/[]            P2 新进程从 Q0 开始
t=5    P0(Q2)      []/[P2]/[P0]            P0 降级到 Q2
...
```

4.2 可视化展示

系统提供四个主要展示界面：

- **参数设置**：配置算法类型、时间片大小、进程参数
- **PRR 展示**：动态展示优先级时间片轮转调度过程，包括就绪队列、状态表、甘特图
- **MLFQ 展示**：动态展示三级队列调度过程，突出队列级别变化
- **算法对比**：比较两种算法的平均周转时间和性能特点

4.3 典型调度时序



5. 设计总结

5.1 关键问题解决

抢占逻辑：在时间循环中检查就绪队列，高优先级进程立即抢占当前进程，确保优先级高的进程及时执行。

I/O 等待机制：PCB 增加 WaitUntil 字段，进程每执行 3 单位时间后等待 I/O 2 单位时间，循环中检查唤醒条件。

死锁检测：就绪队列为空且所有进程在等待状态且无进程会唤醒时判定死锁。

5.2 算法特点对比

特点	优先级时间片轮转	多级反馈队列轮转
队列管理	单一就绪队列	三级反馈队列
时间片	固定统一	按队列递增(1,2,4)
优先级调整	时间片后+3	基于队列级别动态变化
短进程响应	较慢	快速(Q0 短片)
长进程公平性	可能饥饿	Q2 长片保证
实现复杂度	较低	较高

5.3 经验体会

通过本次课程设计，深入理解了进程调度的理论与实践结合。多级反馈队列算法的"反馈"机制体现在进程的 CPU 使用行为会动态调整其队列位置和时间片长度。可视化展示使抽象算法原理变得直观，体现了可视化工具在教学中的价值。

5.4 改进方向

- 支持更多调度算法（最短作业优先、优先级抢占式等）
- 增加资源管理模拟（内存分配、设备管理）
- 优化大量进程时的渲染性能
- 添加预设测试用例库和统计分析功能