**课题二：**

**多级反馈队列调度算法**

1. **背景需求**

#### 1.1 内容

多级反馈队列调度算法（Multilevel Feedback Queue Scheduling, MFQ）是一种综合了多种调度算法优点的高级调度算法，旨在解决操作系统中进程调度的多样化需求。不同于传统的单一调度策略，多级反馈队列算法通过设置多个优先级队列，依据进程的执行历史动态调整其优先级，能够有效地满足不同类型进程的调度需求。

在这种算法中，进程根据其响应时间和执行状态在多个队列之间流动。例如，当进程在某个队列中未能在规定的时间片内完成时，它会被移到优先级较低的队列，并采用不同的调度策略。

#### 1.2 功能

· 进程调度优先级动态调整：根据进程的执行情况，动态调整进程的优先级，避免长期等待的进程一直处于低优先级队列中，导致系统的响应时间过长。

· 多级队列管理：算法通过多个就绪队列，每个队列有不同的调度策略。低优先级队列使用轮转算法（RR），而高优先级队列则使用先来先服务（FCFS）。

· 时间片管理：每个队列中的进程都有一个时间片，如果在时间片内进程未能完成，则会被移动到下一个优先级较低的队列，以保证CPU资源的合理分配。

· 进程迁移机制：通过动态的进程迁移机制，保证高优先级进程的及时处理，同时防止低优先级进程长时间得不到调度。

#### 1.3 意义

多级反馈队列调度算法能够保证响应时间较短的交互式进程得到优先处理，从而提高系统的响应性和用户体验，特别适用于处理交互式任务和批处理任务的混合环境。通过设置多个优先级队列，系统能够公平地分配CPU时间给不同类型的进程，避免某些进程因优先级较低而长时间等待，导致系统不公平现象。

1. **系统设计**

#### 2.1 架构设计

本系统实现了一个模拟多级反馈队列调度算法（MFQ）的程序。该系统的架构设计采用模块化方式，主要由以下几个部分组成：

**（1）初始化模块**：负责初始化系统参数和进程信息。根据用户选择的方式（随机生成或键盘输入），生成或读取进程信息以及相关的队列设置。初始化函数init完成了进程的创建、到达时间和运行时间的分配，以及就绪队列和时间片大小的初始化。

**（2）调度模块**：多级反馈队列调度算法通过MFQ函数实现。此模块负责模拟不同优先级队列之间的调度过程，处理时间片轮转、进程迁移等。系统有多个就绪队列，每个队列对应不同的优先级，采用不同的调度策略（例如，低优先级队列使用轮转算法RR，高优先级队列使用先来先服务FCFS）。

**（3）打印模块**：print函数负责输出调度结果，包括每个进程的开始时间、运行时间、完成时间、周转时间和带权周转时间，以及平均周转时间和平均带权周转时间。

#### 2.2 数据结构设计

（1）进程信息结构体（Process）：

- id：进程的唯一标识符。

- arrival：进程的到达时间。

- burst：进程的运行时间。

- remaining：进程剩余运行时间（用于进程调度过程中追踪进程状态）。

- start\_time：进程开始执行的时间。

- finish\_time：进程完成执行的时间。

- turnaround\_time：周转时间，计算为finish\_time - arrival。

- weighted\_turnaround\_time：带权周转时间，计算为turnaround\_time / burst。

全局变量：

- num\_queues：就绪队列的数量。

- num\_processes：进程的总数。

- time\_quantum[MAX\_QUEUES]：每个队列的时间片大小。

- ready\_queues[MAX\_QUEUES]：多个就绪队列的实现，使用队列（std::queue）存储进程。

- all\_processes：存储所有进程的信息。

- ump：一个映射，用于存储进程ID与其在all\_processes向量中的索引对应关系，便于访问。

#### 2.3 编程语言

本系统使用C++编写，使用C++标准库中STL简化代码编写。

- std::queue：用于存储每个队列中的进程。

- std::vector：用于存储所有进程信息。

- std::map：用于存储进程ID与其对应在进程列表中的索引，方便调度时对进程进行快速访问。

- std::set：用于记录已用过的到达时间，确保进程的到达时间唯一。

1. **详细设计**
   1. 算法流程

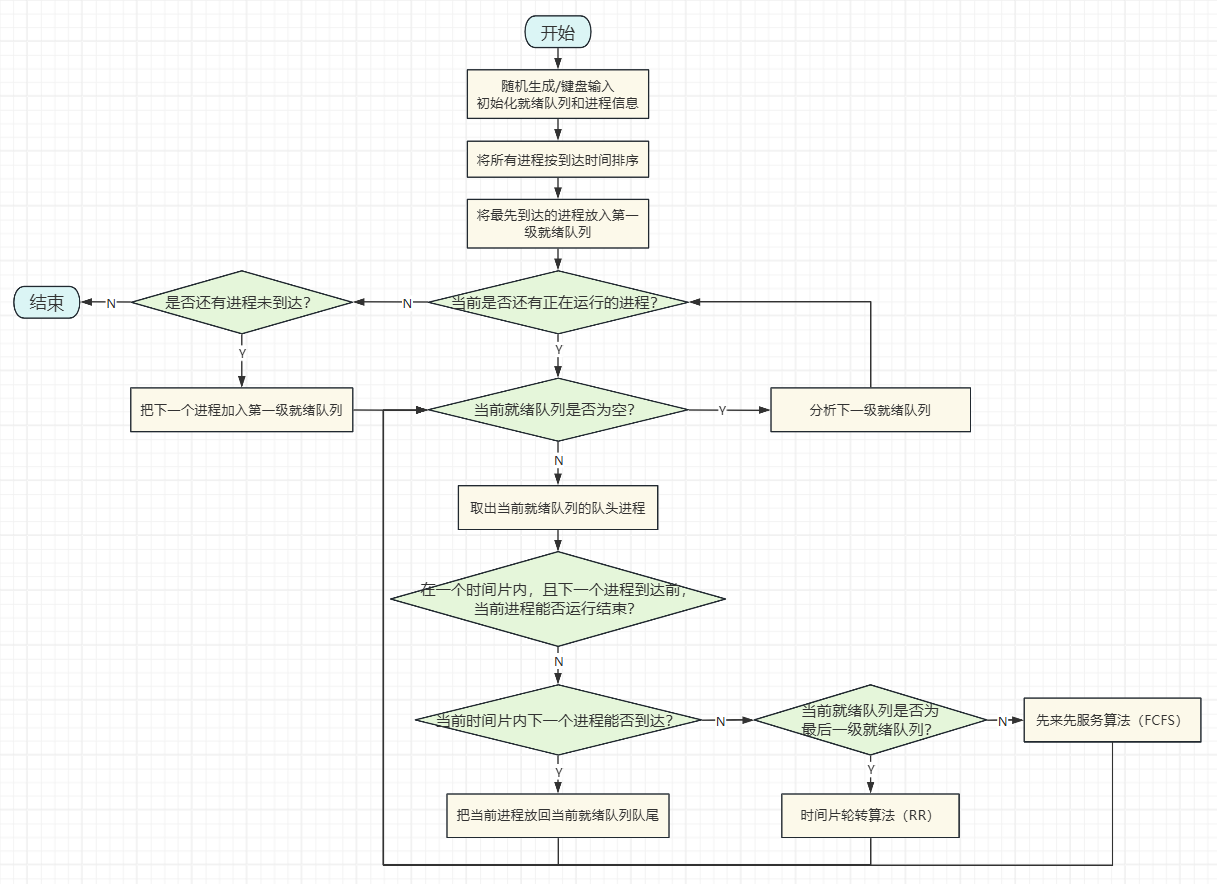
该算法通过多个优先级队列来管理进程，根据进程的执行情况动态调整其所在队列的优先级。每个队列有一个不同的时间片大小，通常高优先级队列采用较短的时间片，低优先级队列采用较长的时间片。算法的核心思想是：

（1）高优先级队列：处理时间较短的进程，采用先来先服务（FCFS）调度策略。

（2）低优先级队列：处理时间较长的进程，采用时间片轮转（RR）调度策略。

（3）动态调整：进程开始时处于高优先级队列，执行一定时间后如果未完成，便移到低优先级队列。当低优先级队列的进程占用完时间片后，如果它未完成，则会被进一步移到更低优先级队列。

（4）进程的迁移：如果新的进程到达，系统会中断当前执行的进程，将其放置原队列尾部，运行新进程。



* 1. 函数功能描述

|  |  |
| --- | --- |
| 函数 | 功能描述 |
| init(int op) | 初始化函数，用于生成进程信息或从用户输入获取进程数据。根据op值选择是否随机生成进程数据或者从键盘输入。 |
| MFQ() | 多级反馈队列调度的核心函数。该函数通过多级队列来执行进程调度，采用时间片轮转（RR）算法、先来先服务（FCFS）算法在不同优先级队列之间切换。 |
| print() | 输出调度结果，包括每个进程的开始时间、结束时间、周转时间等指标。 |
| nextProcess() | 用于从当前队列中选出下一个进程进行调度，并将其加入到相应队列。 |
| show\_processes() | 用于显示当前系统中所有进程的信息，方便用户查看进程的基本属性（如ID、到达时间、执行时间等）。 |

1. **编码**

#include <set>

#include <map>

#include <queue>

#include <time.h>

#include <vector>

#include <stdio.h>

#include <iomanip>

#include <iostream>

#include <stdlib.h>

#include <stdbool.h>

#include <algorithm>

#define MAX\_PROCESSES 100

#define MAX\_QUEUES 5

typedef struct

{

int id; // 进程ID

int arrival; // 到达时间

int burst; // 运行时间

int remaining; // 剩余运行时间

int start\_time; // 开始执行时间

int finish\_time; // 完成时间

int turnaround\_time; // 周转时间

double weighted\_turnaround\_time; // 带权周转时间

} Process;

int num\_queues; // 就绪队列数量

int num\_processes; // 进程数量

int time\_quantum[MAX\_QUEUES]; // 每个队列的时间片大小

std::queue<Process> ready\_queues[MAX\_QUEUES]; // 多级就绪队列

std::vector<Process> all\_processes; // 全部进程信息

std::map<int, int> ump; // 排序后映射

int total\_turnaround\_time; // 周转总时间

double total\_weighted\_turnaround\_time; // 带权周转总时间

void init(int op);

void MFQ();

void print();

void show\_processes();

signed main()

{

srand(time(0));

while (1)

{

std::cout << "请选择测试样例生成方式：\n1. 随机生成\n2. 键盘输入\n3. 退出\n";

int opt; std::cin >> opt;

if (opt == 1)

{

std::cout << "随机生成: \n";

init(1);

}

else if (opt == 2)

{

std::cout << "键盘输入: \n";

init(2);

}

else exit(0);

MFQ();

print();

}

}

void show\_processes()

{

std::cout << "就绪队列总数：" << num\_queues << '\n';

std::cout << "第一个就绪队列时间片大小：" << time\_quantum[0] << '\n';

std::cout << "进程共 " << num\_processes << " 项：\n";

for (int i = 0; i < num\_processes; i ++ )

{

std::cout << "进程 " << all\_processes[i].id << " : ";

std::cout << "到达时间：" << all\_processes[i].arrival << ' ';

std::cout << "运行时间：" << all\_processes[i].burst << '\n';

}

}

void init(int op)

{

std::set<int> st; // 记录已用过的开始时间

if (op == 1) // 随机生成

{

// 就绪队列初始化

num\_queues = rand() % 5 + 1; // 就绪队列总数

time\_quantum[0] = rand() % 10 + 1; // 第一个就序队列的时间片大小

for (int i = 1; i < num\_queues; i ++ ) time\_quantum[i] = time\_quantum[i - 1] \* 2;

// 进程初始化

num\_processes = rand() % 10 + 1;

all\_processes.resize(num\_processes);

for (int i = 0; i < num\_processes; i ++ )

{

all\_processes[i].id = i + 1;

all\_processes[i].arrival = std::max(rand() % 20 + 1, num\_processes + 10);

while (st.count(all\_processes[i].arrival)) all\_processes[i].arrival = rand() % std::max(20, num\_processes + 10) + 1;

st.insert(all\_processes[i].arrival);

all\_processes[i].burst = rand() % 50 + 1;

all\_processes[i].remaining = all\_processes[i].burst;

all\_processes[i].start\_time = all\_processes[i].finish\_time = -1;

all\_processes[i].turnaround\_time = all\_processes[i].weighted\_turnaround\_time = -1;

}

}

else // 键盘输入

{

// 就绪队列初始化

std::cout << "请输入就绪队列总数(最大为5): ";

std::cin >> num\_queues;

std::cout << "请输入第一个就绪队列时间片大小: ";

std::cin >> time\_quantum[0];

for (int i = 1; i < num\_queues; i ++ ) time\_quantum[i] = time\_quantum[i - 1] \* 2;

// 进程初始化

std::cout << "请输入进程总数: ";

std::cin >> num\_processes;

all\_processes.resize(num\_processes);

for (int i = 0; i < num\_processes; i ++ )

{

all\_processes[i].id = i + 1;

std::cout << "请输入第 " << i + 1 << " 个进程的到达时间: ";

std::cin >> all\_processes[i].arrival;

while (st.count(all\_processes[i].arrival))

{

std::cout << "该时间已分配给其他进程, 请重新输入: ";

std::cin >> all\_processes[i].arrival;

}

st.insert(all\_processes[i].arrival);

std::cout << "请输入第 " << i + 1 << " 个进程的运行时间: ";

std::cin >> all\_processes[i].burst;

all\_processes[i].remaining = all\_processes[i].burst;

all\_processes[i].start\_time = all\_processes[i].finish\_time = -1;

all\_processes[i].turnaround\_time = all\_processes[i].weighted\_turnaround\_time = -1;

}

}

show\_processes();

auto cmp = [&](Process a, Process b)

{

return a.arrival < b.arrival;

};

sort(all\_processes.begin(), all\_processes.end(), cmp);

for (int i = 0; i < num\_processes; i ++ ) ump[all\_processes[i].id] = i;

}

// 多级反馈队列调度函数

void MFQ()

{

int current\_queue = 0; // 当前所在队列

int current\_process = 0; // 当前进程

int current\_time = all\_processes[0].arrival; // 当前时间

int process\_running = 1; // 当前正在进行几项进程

// 把下一个进程加进就绪队列

auto nextProcess = [&]()

{

current\_process ++ ;

ready\_queues[0].push(all\_processes[current\_process]);

current\_time = all\_processes[current\_process].arrival;

current\_queue = 0;

process\_running ++ ;

};

// 第一个进程放进第一个队列

ready\_queues[0].push(all\_processes[0]);

while (1)

{

if (process\_running == 0)

{

if (current\_process == num\_processes - 1) break; // 所有进程运行完毕

else nextProcess(); // 把下一个进程加进就绪队列

}

while (!ready\_queues[current\_queue].empty())

{

Process now = ready\_queues[current\_queue].front();

ready\_queues[current\_queue].pop();

int tmp = time\_quantum[current\_queue];

if (current\_process + 1 < num\_processes) tmp = std::min(tmp, all\_processes[current\_process + 1].arrival - current\_time);

if (now.remaining <= tmp) // 当前进程先运行完

{

current\_time += now.remaining;

all\_processes[ump[now.id]].finish\_time = current\_time;

process\_running -- ;

}

else if (current\_process + 1 >= num\_processes || (time\_quantum[current\_queue] <= all\_processes[current\_process + 1].arrival - current\_time)) // 时间片先花完

{

current\_time += time\_quantum[current\_queue];

now.remaining -= time\_quantum[current\_queue];

if (current\_queue == num\_queues - 1) ready\_queues[current\_queue].push(now); // 最后一级队列 RR

else ready\_queues[current\_queue + 1].push(now); // FCFS

}

else // 时间片还没花完 下一个进程先来了

{

now.remaining -= (all\_processes[current\_process + 1].arrival - current\_time);

ready\_queues[current\_queue].push(now);

nextProcess(); // 把下一个进程加进就绪队列

}

}

if (current\_queue + 1 < num\_queues) current\_queue ++ ;

}

}

// 打印结果函数

void print()

{

std::cout << std::fixed << std::setprecision(2);

std::cout << "\n进程调度结果:\n";

std::cout << "进程ID\t开始时间\t运行时间\t完成时间\t周转时间\t带权周转时间\n";

for (int j = 0; j < num\_processes; j ++ )

{

int i = ump[j + 1];

all\_processes[i].turnaround\_time = all\_processes[i].finish\_time - all\_processes[i].arrival;

all\_processes[i].weighted\_turnaround\_time = 1.0 \* all\_processes[i].turnaround\_time / all\_processes[i].burst;

std::cout << all\_processes[i].id << '\t' << all\_processes[i].arrival << "\t\t" << all\_processes[i].burst << "\t\t" << all\_processes[i].finish\_time << "\t\t" << all\_processes[i].turnaround\_time << "\t\t" << all\_processes[i].weighted\_turnaround\_time << '\n';

total\_turnaround\_time += all\_processes[i].turnaround\_time;

total\_weighted\_turnaround\_time += all\_processes[i].weighted\_turnaround\_time;

}

std::cout << "平均周转时间: " << (double)total\_turnaround\_time / num\_processes << '\n';

std::cout << "平均带权周转时间: " << total\_weighted\_turnaround\_time / num\_processes << "\n\n";

}

1. **测试**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 序号 | 条件 | | 测试结果 | 测试过程 |
| num\_queues | num\_processes |
| 1 | 4 | 5 | CORRECT | 图5-1 |
| 2 | 1 | 8 | CORRECT | 图5-2 |
| 3 | 5 | 7 | CORRECT | 图5-3 |
| 4 | 3 | 8 | CORRECT | 图5-4 |
| 5 | 2 | 6 | CORRECT | 图5-5 |

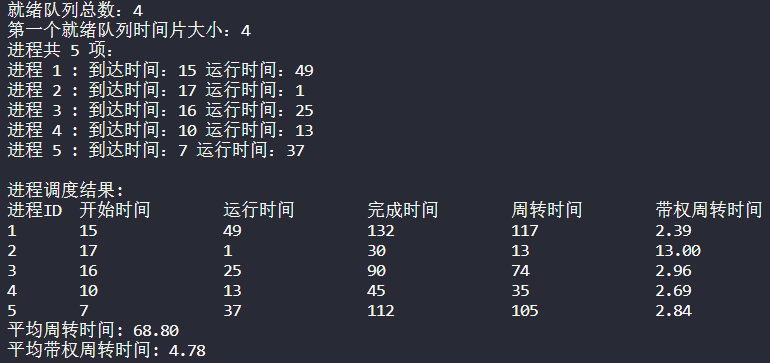


图5-1

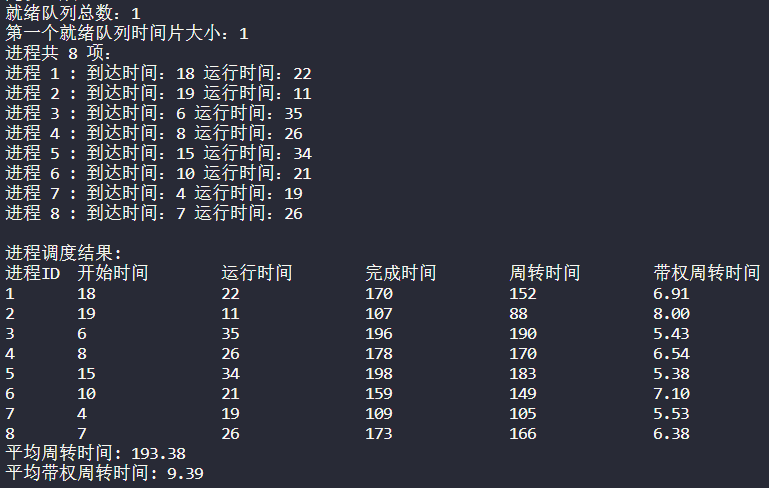


图5-2

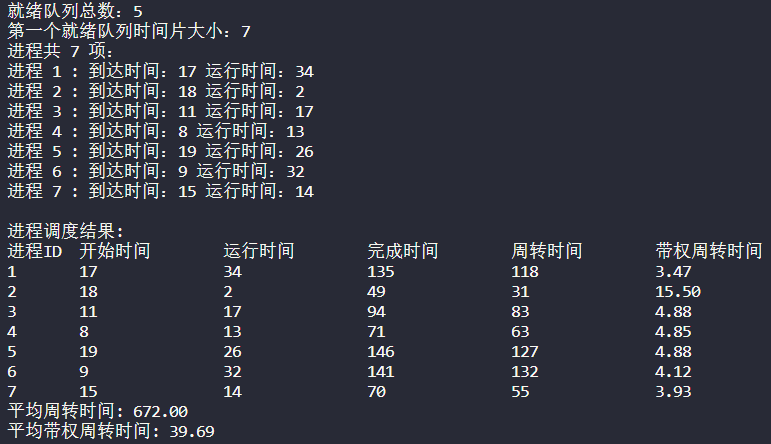


图5-3

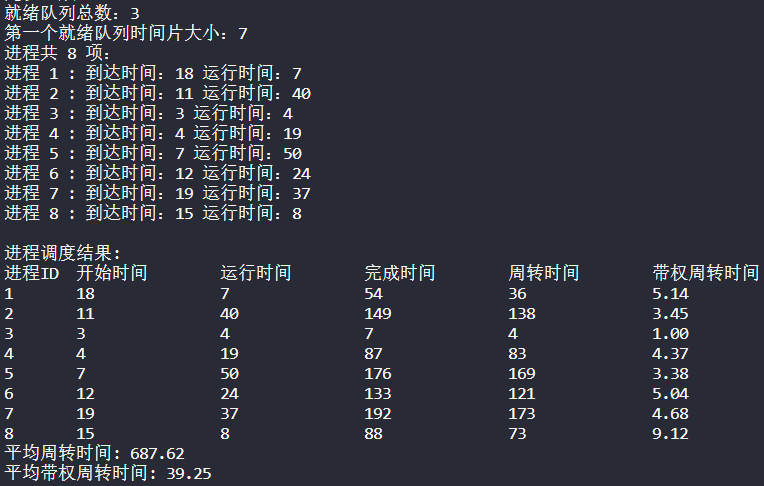


图5-4

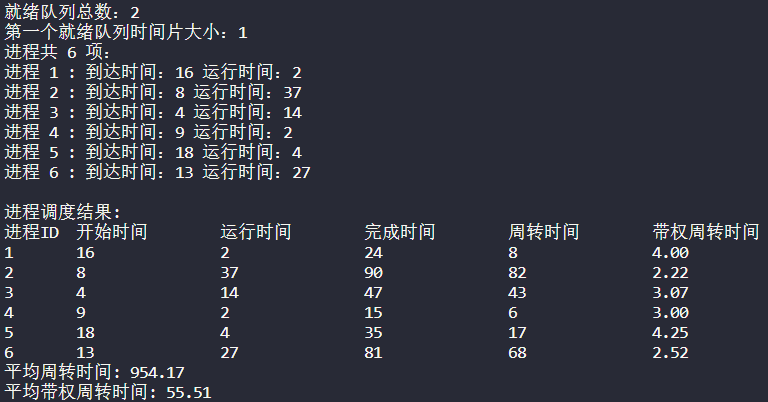


图5-5

1. **总结**

#### 通过此次实验，我深入理解了多级反馈队列（MFQ）调度算法的原理和实现方式。实验中，我通过模拟不同进程的调度，使用多级队列和时间片轮转策略，实现了对进程的动态调度和优先级调整。这不仅加深了我对操作系统调度算法的理解，也让我熟悉了如何通过编程实现复杂的调度逻辑和如何处理进程的时间管理。实验中对进程周转时间和带权周转时间的计算，让我更加清楚了调度算法对系统性能的影响。总的来说，本次实验提高了我在操作系统和进程调度方面的实践能力，对优化和改进调度策略也有了更深的认识。