《操作系统》课程

实验报告

实验名称：	实验三 处理器调度算法模拟实现与比较
姓　　名：	江家玮
学　　号：	**22281188**
日　　期：	**2024.11.01**

目录

1 实验目的

分析处理器实施进程调度的前提条件，理解并掌握各类处理器调度算法的设计原理和实现机制。

2 实验内容

分析和探索处理器实施进程调度的前提条件，理解并掌握处理器调度算法的设计原理和实现机制，随机发生和模拟进程创建及相关事件，编程实现基于特定处理器调度算法（三种以上，譬如先来先服务调度算法、短进程优先调度算法、高优先权优先调度算法、高响应比优先调度算法、时间片轮转调度算法、多级反馈队列调度算法等等）的系统调度处理过程，并加以测试验证。

3 实验代码分析

3.1 代码主要组成部分

该代码是一个操作系统进程调度模拟程序，允许用户输入进程信息并选择不同的调度算法来模拟进程的执行。代码包括以下主要组成部分：

(1)进程控制块（PCB）定义：保存进程相关信息。

(2)进程队列（PCBQueue）定义：用于管理进程调度的队列。

(3)调度算法的实现：支持多种经典的调度算法，如先来先服务（FCFS）、短作业优先（SJF）、高优先级调度（HPF）、高响应比优先（HRRN）和 时间片轮转（RR）。

(4)菜单系统：允许用户选择调度算法并进行模拟。

3.2 代码详解

3.2.1 PCB 结构体 —— 进程控制块

```
typedef struct PCB {
    char name[20];          // 进程名称
    int running_time;       // 进程运行时间，表示进程需要执行的时间
    int enter_time;         // 进程到达时间
    int priority;           // 进程优先级
    int done_time;          // 完成时间（主要用于时间片轮转）
    int copyRunning_time;   // 运行时间的副本，用于时间片轮转时保留原始的运行时间
    int start_time;         // 进程开始执行的时间
    struct PCB* next;       // 指向下一个进程的指针，用于链表结构
} PCB;
```

✱PCB 是操作系统中常用的数据结构，用于保存进程的状态信息。

✱name：存储进程的名称。

✱running_time：进程所需的运行时间。

✱enter_time：进程到达系统的时间，表示进程被调度的起始点。

✱priority：进程的优先级。

✱done_time：进程的完成时间，特别在时间片轮转中使用。

✱copyRunning_time：运行时间的副本，用于时间片轮转时记录剩余的运行时间。

✱start_time：进程真正开始运行的时间。

✱next：用于将多个 PCB 组成一个链表，形成一个进程队列。

3.2.2 PCBQueue 结构体 —— 进程队列

```
typedef struct PCBQueue {
    PCB* firstProg;         // 队列的第一个进程指针
    PCB* LastProg;          // 队列的最后一个进程指针
    int size;               // 队列中的进程数
} PCBQueue;
```

✱PCBQueue 用于管理进程调度中的进程队列，维护队列中的进程顺序。

✱firstProg：指向队列的第一个进程（即头部）。

✱LastProg：指向队列的最后一个进程（即尾部）。

✱size：记录队列中的进程数量。

3.2.3 队列的初始化和操作函数

3.2.3.1 Queueinit —— 初始化队列

```
void Queueinit(PCBQueue* queue) {
    if (queue == NULL) {
        return;
    }
    queue->size = 0;                          // 初始化队列大小为0
    queue->LastProg = (PCB*)malloc(sizeof(PCB)); // 为队列分配一个空的 `PCB` 作为尾节点
    queue->firstProg = queue->LastProg;       // 初始化时队头和队尾指向同一个空节点
}
```

Queueinit 函数用于初始化进程队列：

✱为队列的头部和尾部分配内存。

✱初始化时队列的 size 为 0，表示没有进程。

✱队列头部和尾部都指向同一个空的 PCB，此时队列为空。

3.2.3.2 EnterQueue —— 进程入队

```
void EnterQueue(PCBQueue* queue, PCB* pro) {
    queue->LastProg->next = (PCB*)malloc(sizeof(PCB)); // 为新进程分配内存
    queue->LastProg = queue->LastProg->next;          // 队尾指向新加入的进程
    queue->LastProg->enter_time = pro->enter_time;    // 复制进程的到达时间
    memcpy(queue->LastProg->name, pro->name, sizeof(pro->name)); // 复制进程的名称
    queue->LastProg->priority = pro->priority;        // 复制进程的优先级
    queue->LastProg->running_time = pro->running_time;// 复制进程的运行时间
    queue->LastProg->copyRunning_time = pro->copyRunning_time; // 复制运行时间副本
    queue->LastProg->start_time = pro->start_time;    // 复制进程开始时间
    queue->size++;                                    // 队列大小增加
}
```

EnterQueue 函数用于将新的进程 pro 加入队列：

✱为新的 PCB 分配内存，将其加入队列末尾。

✱复制进程的各个属性（如 name、priority 等），将其存储到新分配的节点中。

✱更新队列尾指针，并增加队列大小。

3.2.3.3 poll —— 进程出队

```
PCB* poll(PCBQueue* queue) {
    PCB* temp = queue->firstProg->next; // 从队列头部取出下一个进程
    if (temp == queue->LastProg) {      // 如果该进程是最后一个进程
        queue->LastProg = queue->firstProg; // 更新队列尾指针指向队头
        queue->size--;                  // 队列大小减小
        return temp;
    }
    queue->firstProg->next = queue->firstProg->next->next; // 移动队列头部指针
    queue->size--;                  // 队列大小减小
    return temp;
}
```

poll 函数用于从队列中取出第一个进程：

✱获取队列头部的下一个进程，并将头指针向后移动。

✱如果队列中只剩下最后一个进程，则将队列的尾指针更新为队头。

✱每次出队后，队列大小减 1。

3.2.4 进程输入和排序

3.2.4.1 inputPCB —— 输入进程信息

```
void inputPCB(PCB pro[], int num) {
    for (int i = 0; i < num; i++) {
```

```
            PCB prog;
            printf("请输入第%d 个进程的名字，到达时间 ，服务时间，优先级\n", i + 1);
            scanf("%s", prog.name);
            scanf("%d", &prog.enter_time);
            scanf("%d", &prog.running_time);
            prog.copyRunning_time = prog.running_time;
            scanf("%d", &prog.priority);
            pro[i] = prog;
        }
    }
```

inputPCB 函数用于输入一组进程的信息：

✱用户需要输入每个进程的 name（名字）、enter_time（到达时间）、running_time（运行时间）和 priority（优先级）。

✱将输入的信息存储到 pro[]数组中，供后续调度使用。

3.2.4.2 sortWithEnterTime —— 根据到达时间排序

```
void sortWithEnterTime(PCB pro[], int num) {
    for (int i = 1; i < num; i++) {
        for (int j = 0; j < num - i; j++) {
            if (pro[j].enter_time > pro[j + 1].enter_time) {
                PCB temp = pro[j];
                pro[j] = pro[j + 1];
                pro[j + 1] = temp;
            }
        }
    }
}
```

sortWithEnterTime 函数通过冒泡排序算法按照进程的 enter_time 对进程数组进行排序：

✱每次比较相邻的两个进程，如果前一个进程的到达时间较晚，就交换两个进程的顺序。

✱这样经过 n 次排序后，数组按照进程到达时间从小到大排列。

3.2.5 调度算法的实现

3.2.5.1 先来先服务调度（FCFS）

```
void FCFS(PCB pro[], int num) {
    printf("进程 到达时间  服务时间 开始时间 完成时间 周转时间 带权周转时间\n");
    sortWithEnterTime(pro, num);  // 按到达时间排序
    PCBQueue* queue = (PCBQueue*)malloc(sizeof(PCBQueue));
    Queueinit(queue);             // 初始化队列
    EnterQueue(queue, &pro[0]);   // 将第一个进程加入队列
    int time = pro[0].enter_time; // 当前时间初始化为第一个进程的到达时间
```

```
        int pronum = 1;               // 记录已处理进程数
        float sum_T_time = 0;         // 总周转时间
        float sum_QT_time = 0;        // 总带权周转时间

        while (queue->size > 0) {
            PCB* curpro = poll(queue);       // 从队列中取出第一个进程
            if (time < curpro->enter_time) time = curpro->enter_time; // 更新当前时间
            int done_time = time + curpro->running_time; // 计算完成时间
            int T_time = done_time - curpro->enter_time; // 周转时间
            float QT_time = T_time / (curpro->running_time + 0.0); // 带权周转时间
            sum_T_time += T_time;
            sum_QT_time += QT_time;

            for (int tt = time; tt <= done_time && pronum < num; tt++) {
                if (tt >= pro[pronum].enter_time) {
                    EnterQueue(queue, &pro[pronum]); // 将新的进程按到达时间加入队列
                    pronum++;
                }
            }

            printf("%s\t%d\t%d\t%d\t%d\t%.2f\n", curpro->name, curpro->enter_time, curpro->running
_time, time, done_time, T_time, QT_time);
            time += curpro->running_time; // 更新当前时间

            if (queue->size == 0 && pronum < num) {
                EnterQueue(queue, &pro[pronum]); // 如果队列为空且还有未处理的进程，则加入新进程
                pronum++;
            }
        }

        printf("平均周转时间为%.2f\t 平均带权周转时间为%.2f\n", sum_T_time / num, sum_QT_time / num);
    }
```

FCFS 的特点是按到达顺序进行调度，不考虑运行时间和优先级，先到的进程先执行。

✱使用 sortWithEnterTime 函数将进程按照到达时间进行排序。

✱将第一个进程入队，依次执行每个进程，并计算其完成时间、周转时间和带权周转时间。

✱输出每个进程的执行信息，并计算平均周转时间和带权平均周转时间。

3.2.5.2 短作业优先调度（SJF）

```
    void SJF(PCB pro[], int num) {
        printf("进程 到达时间  服务时间 开始时间 完成时间 周转时间 带权周转时间\n");
        sortWithEnterTime(pro, num);          // 按到达时间排序
```

```
4.          PCBQueue* queue = (PCBQueue*)malloc(sizeof(PCBQueue));
5.          Queueinit(queue);
6.          EnterQueue(queue, &pro[0]);           // 将第一个进程入队
7.          int time = pro[0].enter_time;         // 当前时间初始化为第一个进程的到达时间
8.          int pronum = 1;                       // 已处理的进程数
9.          float sum_T_time = 0, sum_QT_time = 0;// 总周转时间和总带权周转时间
10.
11.         while (queue->size > 0) {
12.             PCB* curpro = poll(queue);       // 取出队首进程
13.             if (time < curpro->enter_time) time = curpro->enter_time;
14.             int done_time = time + curpro->running_time; // 计算完成时间
15.             int T_time = done_time - curpro->enter_time; // 周转时间
16.             float QT_time = T_time / (curpro->running_time + 0.0); // 带权周转时间
17.             sum_T_time += T_time;
18.             sum_QT_time += QT_time;
19.
20.             int pre = pronum;  // 记录当前未处理进程的下标
21.             for (int tt = time; tt <= done_time && pronum < num; tt++) {
22.                 if (tt >= pro[pronum].enter_time) pronum++;
23.             }
24.
25.             sortWithLength(pro, pre, pronum); // 将到达的进程按运行时间排序，短作业优先
26.             for (int i = pre; i < pronum; i++) {
27.                 EnterQueue(queue, &pro[i]);   // 将按运行时间排序的进程入队
28.             }
29.
30.             printf("%s\t%d\t%d\t%d\t%d\t%.2f\n", curpro->name, curpro->enter_time, curpro->running
    _time, time, done_time, T_time, QT_time);
31.             time += curpro->running_time;
32.
33.             if (queue->size == 0 && pronum < num) {
34.                 EnterQueue(queue, &pro[pronum]);
35.                 pronum++;
36.             }
37.         }
38.
39.         printf("平均周转时间为%.2f\t 平均带权周转时间为%.2f\n", sum_T_time / num, sum_QT_time / num);
40.     }
```

SJF 算法的特点是选择运行时间最短进程优先执行，从而减少平均周转时间。

✱同样首先按照到达时间排序。

✱每次调度时，先取出当前的进程进行执行，同时将已经到达的进程按运行时间排序，并加入队列。

3.2.5.3 时间片轮转调度（RR）

```
1.  void RR(PCB pro[], int num) {
2.      printf("请输入时间片大小: ");
3.      int timeslice;
4.      scanf("%d", &timeslice);
5.      printf("进程 到达时间 服务时间 进入时间 完成时间 周转时间 带权周转时间\n");
6.      sortWithEnterTime(pro, num);
7.      PCBQueue* queue = (PCBQueue*)malloc(sizeof(PCBQueue));
8.      Queueinit(queue);
9.      pro[0].start_time = pro[0].enter_time;
10.     EnterQueue(queue, &pro[0]);
11.     int time = 0, pronum = 1;
12.     float sum_T_time = 0, sum_QT_time = 0;
13.
14.     while (queue->size > 0) {
15.         PCB* curpro = poll(queue);    // 从队列中取出进程
16.         if (time < curpro->enter_time) time = curpro->enter_time;
17.         if (timeslice >= curpro->running_time) { // 如果进程的运行时间小于时间片
18.             for (int tt = time; tt <= time + curpro->running_time && pronum < num; tt++) {
19.                 if (tt >= pro[pronum].enter_time) { // 进程到达则入队
20.                     pro[pronum].start_time = tt;
21.                     EnterQueue(queue, &pro[pronum]);
22.                     pronum++;
23.                 }
24.             }
25.             time += curpro->running_time;
26.             curpro->running_time = 0;
27.             curpro->done_time = time;
28.             int T_time = curpro->done_time - curpro->start_time;
29.             float QT_time = T_time / (curpro->copyRunning_time + 0.0);
30.             sum_T_time += T_time;
31.             sum_QT_time += QT_time;
32.             printf("%s\t%d\t%d\t  %d\t   %d\t %d\t  %.2f\n", curpro->name, curpro->enter_time, cur
    pro->copyRunning_time, curpro->start_time, curpro->done_time, T_time, QT_time);
33.         } else {
34.             for (int tt = time; tt <= time + timeslice && pronum < num; tt++) {
35.                 if (tt >= pro[pronum].enter_time) {
36.                     pro[pronum].start_time = tt;
37.                     EnterQueue(queue, &pro[pronum]);
38.                     pronum++;
39.                 }
40.             }
```

```
                time += timeslice;
                curpro->running_time -= timeslice;
                EnterQueue(queue, curpro);
            }

            if (queue->size == 0 && pronum < num) {
                pro[pronum].start_time = pro[pronum].enter_time;
                EnterQueue(queue, &pro[pronum]);
                pronum++;
            }
        }
        printf("平均周转时间为%.2f\t 平均带权周转时间为%.2f\n\n", sum_T_time / num, sum_QT_time / num);
    }
```

时间片轮转（RR）调度是基于固定时间片的调度算法：

✱每个进程最多只能运行一个时间片，如果未完成，则被重新加入队列，等待下一个时间片继续执行。

✱通过时间片的方式，保证所有进程能公平地获得 CPU 时间。

✱当一个进程完成后，计算其周转时间和带权周转时间，并输出结果。

4 实验结果展示

```
请输入进程的个数：4
请输入第1个进程的名字，到达时间 ，服务时间，优先级
1 4 2 3
请输入第2个进程的名字，到达时间 ，服务时间，优先级
2 3 2 1
请输入第3个进程的名字，到达时间 ，服务时间，优先级
3 1 5 3
请输入第4个进程的名字，到达时间 ，服务时间，优先级
4 2 3 2
请选择进程调度算法：

1.先来先服务算法
2.短进程优先算法
3.高优先级优先
4.时间片轮转算法
5.高响应比优先算法
6.退出
1
进程 到达时间  服务时间 开始时间 完成时间 周转时间 带权周转时间
3       1       5       1       6       5       1.00
4       2       3       6       9       7       2.33
2       3       2       9       11      8       4.00
1       4       2       11      13      9       4.50
平均周转时间为7.25      平均带权周转时间为2.96
```

图 1 先来先服务算法结果展示

```
请选择进程调度算法：

1.先来先服务算法
2.短进程优先算法
3.高优先级优先
4.时间片轮转算法
5.高响应比优先算法
6.退出
3
进程 到达时间  服务时间 开始时间 完成时间 周转时间 带权周转时间
3       1       5       1       6       5       1.00
2       3       2       6       8       5       2.50
4       2       3       8       11      9       3.00
1       4       2       11      13      9       4.50
平均周转时间为7.00      平均带权周转时间为2.75
```

图 2 短进程优先算法结果展示

```
请选择进程调度算法：

1.先来先服务算法
2.短进程优先算法
3.高优先级优先
4.时间片轮转算法
5.高响应比优先算法
6.退出
3
进程  到达时间   服务时间  开始时间  完成时间  周转时间  带权周转时间
3       1       5       1       6       5       1.00
2       3       2       6       8       5       2.50
4       2       3       8       11      9       3.00
1       4       2       11      13      9       4.50
平均周转时间为7.00      平均带权周转时间为2.75
```

图 3 高优先级优先算法结果展示

```
请选择进程调度算法：

1.先来先服务算法
2.短进程优先算法
3.高优先级优先
4.时间片轮转算法
5.高响应比优先算法
6.退出
4
请输入时间片大小10
进程  到达时间  服务时间  进入时间  完成时间  周转时间  带权周转时间
3       1       5       1       6       5       1.00
4       2       3       2       9       7       2.33
2       3       2       3       11      8       4.00
1       4       2       4       13      9       4.50
平均周转时间为7.25      平均带权周转时间为2.96
```

图 4 时间片轮转算法结果展示

```
请选择进程调度算法：

1.先来先服务算法
2.短进程优先算法
3.高优先级优先
4.时间片轮转算法
5.高响应比优先算法
6.退出
5
进程  到达时间   服务时间  开始时间  完成时间  周转时间  带权周转时间
3       1       5       1       6       5       1.00
4       2       3       6       9       7       2.33
2       3       2       9       11      8       4.00
1       4       2       11      13      9       4.50
平均周转时间为7.25      平均带权周转时间为2.96
```

图 5 高响应比优先算法结果展示

5 疑难解惑及经验教训

5.1 疑难解惑

（1）问题：时间片轮转调度（RR）如何处理运行时间大于时间片的进程？

✱解惑：在时间片轮转调度中，如果某个进程的运行时间大于分配的时间片，进程只会执行时间片长度的时间，剩余的执行时间将保留并在下一个时间片中继续运行。未完成的进程会被重新加入队列，等待下一个调度周期。这确保了每个进程都能公平地获得 CPU 时间，但如果时间片过短，会增加进程切换的开销，降低效率。

（2）问题：短作业优先（SJF）调度如何处理进程到达的时间不一致的情况？

✱解惑：短作业优先调度（SJF）在每次调度时会选取当前所有已到达的进程中运行时间最短的进程进行执行。如果在某个进程执行过程中，有新的进程到达，算法会将这些新到达的进程加入队列，并重新排序，确保下一个被调度的进程仍然是当前所有已到达进程中运行时间最短的。这种动态排序方式确保了调度的灵活性，但可能导致长作业的饥饿问题。

（3）问题：调度算法如何影响系统的平均周转时间和带权周转时间？

✱解惑：不同的调度算法对系统的平均周转时间和带权周转时间有不同的影响。先来先服务（FCFS）调度可能导致后来的进程等待时间较长，增加平均周转时间。短作业优先（SJF）通常能有效降低平均周转时间，因为它优先处理运行时间短的进程。时间片轮转（RR）则更适合交互式系统，能让所有进程公平分配 CPU 时间，但在某些情况下，可能增加整体的平均周转时间。因此，选择合适的调度算法取决于系统的性能需求和进程的特性。

5.2 经验教训

（1）教训：合理设置时间片长度至关重要

在时间片轮转调度中，时间片的长度对系统的性能有重大影响。如果时间片过短，系统会频繁进行进程切换，导致上下文切换开销过高；如果时间片过长，时间片轮转的公平性就会受到影响，可能导致部分进程得不到及时的响应。因此，合理设置时间片长度需要权衡系统的上下文切换开销与进程响应时间之间的关系。

（2）教训：SJF 虽然高效，但可能导致“长作业饥饿”

虽然短作业优先（SJF）调度能够优化系统的平均周转时间，但它可能导致长时间运行的进程长期无法被调度，出现“饥饿”现象。为了解决这个问题，可以采用一种变体——最高响应比优先（HRRN）调度，综合考虑等待时间和运行时间，避免长时间的作业被无限推迟。

（3）教训：进程到达的动态性需要灵活的调度策略

在设计和选择调度算法时，必须考虑进程到达的动态性。像 FCFS 这样简单的算法在处理随机到达的进程时可能导致次优的调度结果，而更复杂的算法如 SJF 和 RR 则能够更好地处理这种动态场景。经验表明，在多用户和多任务系统中，选择灵活性更高的调度算法（如 RR 或 HRRN）能够显著提升系统的响应速度和用户体验。

附录

```c
#include<stdio.h>
#include<malloc.h>
#include<string.h>
#include<stdlib.h>

typedef struct PCB{
    char name[20];
//  运行时间
    int running_time;
//   到达时间
    int enter_time;
//    优先级
    int priority;
//   完成时间
    int done_time;    //用于时间片轮转
    int copyRunning_time;  //用于时间片轮转
//  进程开始运行的时间
    int start_time;

```

```
    struct PCB* next;
} PCB;

typedef struct PCBQueue{
    PCB* firstProg;
    PCB* LastProg;
    int size;
} PCBQueue;

void Queueinit(PCBQueue* queue){
    if(queue==NULL){
        return;
    }
    queue->size = 0;
    queue->LastProg = (PCB*)malloc(sizeof(PCB));
    queue->firstProg = queue->LastProg;
}



void EnterQueue(PCBQueue* queue,PCB* pro){   //加入进程队列
    queue->LastProg->next = (PCB*)malloc(sizeof(PCB));
    queue->LastProg = queue->LastProg->next;
    queue->LastProg->enter_time = pro->enter_time;
//    将name复制给 LastProg
    memcpy(queue->LastProg->name,pro->name,sizeof(pro->name));
    queue->LastProg->priority = pro->priority;
    queue->LastProg->running_time = pro->running_time;
    queue->LastProg->copyRunning_time = pro->copyRunning_time;
    queue->LastProg->start_time = pro->start_time;
    queue->size++;
}
PCB* poll(PCBQueue* queue){
    PCB* temp = queue->firstProg->next;
    if(temp == queue->LastProg){
        queue->LastProg=queue->firstProg;
        queue->size--;
        return temp;
    }
    queue->firstProg->next = queue->firstProg->next->next;
    queue->size--;
    return temp;
}

```

```
64.  void inputPCB(PCB pro[],int num){
65.      for(int i=0;i<num;i++){
66.          PCB prog ;
67.          printf("请输入第%d 个进程的名字，到达时间 ，服务时间，优先级\n",i+1);
68.          scanf("%s",prog.name);
69.          scanf("%d",&prog.enter_time) ;
70.          scanf("%d",&prog.running_time);
71.          prog.copyRunning_time = prog.running_time;
72.          scanf("%d",&prog.priority);
73.          pro[i]=prog;
74.      }
75.  }
76.
77.  //冒泡排序算法（每次找到最大的放在末尾）
78.  void sortWithEnterTime(PCB pro[],int num){
79.      for(int i=1;i<num;i++){
80.          for(int j= 0;j<num-i;j++){
81.              if(pro[j].enter_time>pro[j+1].enter_time){
82.                  PCB temp = pro[j];
83.                  pro[j] = pro[j+1];
84.                  pro[j+1] = temp;
85.              }
86.          }
87.      }
88.  }
89.
90.  void FCFS(PCB pro[],int num){
91.      printf("进程 到达时间  服务时间 开始时间 完成时间 周转时间 带权周转时间\n");
92.      sortWithEnterTime(pro,num);    //按照进入顺序排序
93.      PCBQueue* queue = (PCBQueue*)malloc(sizeof(PCBQueue));
94.      Queueinit(queue);
95.      EnterQueue(queue,&pro[0]);
96.      int time = pro[0].enter_time;
97.      int pronum=1;    //记录当前的进程
98.      //平均周转时间
99.      float sum_T_time = 0;
100. //    带权平均周转时间
101.     float sum_QT_time = 0 ;
102.
103.     while(queue->size>0){
104.         PCB* curpro = poll(queue);   //从进程队列中取出进程
105.         if(time < curpro->enter_time)
106.             time =  curpro->enter_time;
107.         //完成时间
```

```
108.            int done_time = time+curpro->running_time;
109.            //周转时间（作业完成的时间-作业到达的时间）
110.            int T_time = done_time - curpro->enter_time;
111.            sum_T_time += T_time;
112.            // 带权周转时间（（作业完成的时间-作业到达的时间）/ 作业运行时间）
113.            float QT_time = T_time / (curpro->running_time+0.0) ;
114.            sum_QT_time += QT_time;
115.            for(int tt = time;tt<=done_time && pronum<num;tt++){    //模拟进程的执行过程
116.                if(tt >= pro[pronum].enter_time){
117.                    EnterQueue(queue,&pro[pronum]);
118.                    pronum++;
119.                }
120.            }
121.            printf("%s\t%d\t%d\t%d\t%d\t%.2f\n",curpro->name,curpro->enter_time,curpro->running_ti
    me,time,done_time
122.                   ,T_time,QT_time);
123.            time += curpro->running_time;
124.            if(queue->size==0 && pronum < num){   //防止出现前一个进程执行完到下一个进程到达之间无进程进入
125.                EnterQueue(queue,&pro[pronum]);
126.                pronum++;
127.            }
128.        }
129.        printf("平均周转时间为%.2f\t 平均带权周转时间
    为%.2f\n",sum_T_time/(num+0.0),sum_QT_time/(num+0.0));
130.    }
131.
132.    //按照运行时间排序
133.    void sortWithLongth(PCB pro[],int start,int end){
134.        int len = end - start;
135.        if(len == 1)
136.            return;
137.        for(int i=1;i<len;i++){
138.            for(int j= start;j<end-i;j++){
139.                if(pro[j].running_time>pro[j+1].running_time){
140.                    PCB temp = pro[j];
141.                    pro[j] = pro[j+1];
142.                    pro[j+1] = temp;
143.                }
144.            }
145.        }
146.    }
147.    void SJF(PCB pro[],int num) {
148.        printf("进程 到达时间  服务时间 开始时间 完成时间 周转时间 带权周转时间\n");
149.        sortWithEnterTime(pro,num);
```

```
150.      PCBQueue* queue = (PCBQueue*)malloc(sizeof(PCBQueue));;
151.      Queueinit(queue);
152.      EnterQueue(queue,&pro[0]);
153.      int time = pro[0].enter_time;
154.      int pronum=1;    //记录当前的进程
155.      float sum_T_time = 0,sum_QT_time = 0;
156.      while(queue->size>0){
157.          PCB* curpro = poll(queue);   //从进程队列中取出进程
158.          if(time <  curpro->enter_time)
159.              time =  curpro->enter_time;
160.          int done_time = time+curpro->running_time;
161.          int T_time = done_time - curpro->enter_time;
162.          float QT_time = T_time / (curpro->running_time+0.0) ;
163.          sum_T_time += T_time;
164.          sum_QT_time += QT_time;
165.          int pre = pronum;
166.          for(int tt = time;tt<=done_time&&pronum<num;tt++){    //模拟进程的执行过程
167.              if(tt>=pro[pronum].enter_time){ // 统计从此任务开始到结束之间有几个进程到达
168.                  pronum++;
169.              }
170.          }
171.          sortWithLength(pro,pre,pronum);//将到达的进程按照服务时间排序
172.          for(int i=pre;i<pronum;i++){    //将进程链入队列
173.              EnterQueue(queue,&pro[i]);
174.          }
175.          pre = pronum;
176.          printf("%s\t%d\t%d\t%d\t%d\t%.2f\n",curpro->name,curpro->enter_time,curpro->running_ti
    me,time,done_time
177.                 ,T_time,QT_time);
178.          time +=  curpro->running_time;
179.          if(queue->size==0&&pronum<num){   //防止出现前一个进程执行完到下一个进程到达之间无进程进入
180.              EnterQueue(queue,&pro[pronum]);
181.              pronum++;
182.          }
183.      }
184.      printf("平均周转时间为%.2f\t 平均带权周转时间为%.2f\n",sum_T_time/(num+0.0),sum_QT_time/num);
185.  }
186.  //按照响应比排序（倒序）
187.  void sortWithResponse(PCB pro[],int start,int end){
188.      int len = end - start;
189.      if(len == 1)
190.          return;
191.      for(int i=1;i<len;i++){
192.          for(int j= start;j<end-i;j++){
```

```
193.                //计算响应比
194.                float m = (pro[j].start_time-pro[j].enter_time+pro[j].running_time)/(pro[j].running_ti
    me+0.0);
195.                float n = (pro[j+1].start_time-pro[j+1].enter_time+pro[j+1].running_time)/(pro[j+1].ru
    nning_time+0.0);
196.                if(m < n){
197.                    PCB temp = pro[j];
198.                    pro[j] = pro[j+1];
199.                    pro[j+1] = temp;
200.                }
201.            }
202.        }
203.    }
204.    //高响应比优先
205.    void HRRN(PCB pro[],int num) {
206.        printf("进程 到达时间  服务时间 开始时间 完成时间 周转时间 带权周转时间\n");
207.        sortWithEnterTime(pro,num);
208.        PCBQueue* queue = (PCBQueue*)malloc(sizeof(PCBQueue));;
209.        Queueinit(queue);
210.        EnterQueue(queue,&pro[0]);
211.        int time = pro[0].enter_time;
212.        int pronum=1;    //记录当前的进程
213.        float sum_T_time = 0,sum_QT_time = 0;
214.        while(queue->size>0){
215.            PCB* curpro = poll(queue);   //从进程队列中取出进程
216.            if(time <  curpro->enter_time)
217.                time =  curpro->enter_time;
218.            int done_time = time+curpro->running_time;
219.            int T_time = done_time - curpro->enter_time;
220.            float QT_time = T_time / (curpro->running_time+0.0) ;
221.            sum_T_time += T_time;
222.            sum_QT_time += QT_time;
223.            int pre = pronum;
224.            for(int tt = time;tt<=done_time&&pronum<num;tt++){    //模拟进程的执行过程
225.                if(tt>=pro[pronum].enter_time){ // 统计从此任务开始到结束之间有几个进程到达
226.                    pronum++;
227.                }
228.            }
229.            sortWithResponse(pro,pre,pronum);//将到达的进程按照响应时间排序
230.            for(int i=pre;i<pronum;i++){    //将进程链入队列
231.                EnterQueue(queue,&pro[i]);
232.            }
233.            pre = pronum;
```

```
234.            printf("%s\t%d\t%d\t%d\t%d\t%.2f\n",curpro->name,curpro->enter_time,curpro->running_ti
    me,time,done_time
235.                    ,T_time,QT_time);
236.            time +=  curpro->running_time;
237.            if(queue->size==0&&pronum<num){   //防止出现前一个进程执行完到下一个进程到达之间无进程进入
238.                EnterQueue(queue,&pro[pronum]);
239.                pronum++;
240.            }
241.        }
242.        printf("平均周转时间为%.2f\t 平均带权周转时间为%.2f\n",sum_T_time/(num+0.0),sum_QT_time/num);
243.    }
244.
245.    //按权重排序
246.    void sortWithPriority(PCB pro[],int start,int end){
247.        int len = end - start;
248.        if(len == 1) return ;
249.        for(int i=1;i<len;i++){
250.            for(int j= start;j<end-i;j++){
251.                if(pro[j].priority>pro[j+1].priority){
252.                    PCB temp = pro[j];
253.                    pro[j] = pro[j+1];
254.                    pro[j+1] = temp;
255.                }
256.            }
257.        }
258.    }
259.    //优先级调度算法
260.    void HPF(PCB pro[],int num){
261.        printf("进程 到达时间  服务时间 开始时间 完成时间 周转时间 带权周转时间\n");
262.        sortWithEnterTime(pro,num);
263.        PCBQueue* queue = (PCBQueue*)malloc(sizeof(PCBQueue));;
264.        Queueinit(queue);
265.        EnterQueue(queue,&pro[0]);
266.        int time = pro[0].enter_time;
267.        int pronum=1;    //记录当前的进程
268.        float sum_T_time = 0,sum_QT_time = 0;
269.        while(queue->size>0){
270.            PCB* curpro = poll(queue);   //从进程队列中取出进程
271.            if(time<curpro->enter_time)
272.                time =  curpro->enter_time;
273.            int done_time = time+curpro->running_time;
274.            int T_time = done_time - curpro->enter_time;
275.            float QT_time = T_time / (curpro->running_time+0.0) ;
276.            sum_T_time += T_time;
```

```
277.            sum_QT_time += QT_time;
278.            int pre = pronum;
279.            for(int tt = time;tt<=done_time&&pronum<num;tt++){    //模拟进程的执行过程
280.                if(tt>=pro[pronum].enter_time){ // 统计从此任务开始到结束之间有几个进程到达
281.                    pronum++;
282.                }
283.            }
284.            sortWithPriority(pro,pre,pronum);//将到达的进程按照服务时间排序
285.            for(int i=pre;i<pronum;i++){    //将进程链入队列
286.                EnterQueue(queue,&pro[i]);
287.            }
288.            pre = pronum;
289.            printf("%s\t%d\t%d\t%d\t%d\t%.2f\n",curpro->name,curpro->enter_time,curpro->running_ti
    me,time,done_time
290.                   ,T_time,QT_time);
291.            time +=  curpro->running_time;
292.            if(queue->size==0&&pronum<num){   //防止出现前一个进程执行完到下一个进程到达之间无进程进入
293.                EnterQueue(queue,&pro[pronum]);
294.                pronum++;
295.            }
296.        }
297.        printf("平均周转时间为%.2f\t 平均带权周转时间
    为%.2f\n",sum_T_time/(num+0.0),sum_QT_time/(num+0.0));
298.    }
299.    //时间片轮转调度
300.    void RR(PCB pro[],int num){
301.        printf("请输入时间片大小");
302.        int timeslice;
303.        scanf("%d",&timeslice);
304.        printf("进程 到达时间 服务时间 进入时间 完成时间 周转时间 带权周转时间\n");
305.        sortWithEnterTime(pro,num);
306.        PCBQueue* queue = (PCBQueue*)malloc(sizeof(PCBQueue));;
307.        Queueinit(queue);
308.        //第一个进程开始运行的时间就是到达时间
309.        pro[0].start_time = pro[0].enter_time;
310.        EnterQueue(queue,&pro[0]);
311.        int time = 0;
312.        int pronum = 1;
313.        float sum_T_time = 0,sum_QT_time = 0;
314.        while(queue->size>0){
315.            PCB* curpro = poll(queue);    // 从队列中取出头节点
316.            if(time<curpro->enter_time)
317.                time = curpro->enter_time;
318.            if(timeslice >= curpro->running_time){   // 如果剩余时间小于时间片  则此任务完成
```

```
            for(int tt = time;tt<=time+curpro->running_time&&pronum<num;tt++){    // 模拟进程的执行过程
                if(tt>=pro[pronum].enter_time){ // 统计从此任务开始到结束之间有几个进程到达
                    pro[pronum].start_time = tt;
                    EnterQueue(queue,&pro[pronum]);
                    pronum++;
                }
            }
            time += curpro->running_time;
            curpro->running_time = 0;
            curpro->done_time = time;
            int T_time = curpro->done_time-curpro->start_time;
            float QT_time = T_time / (curpro->copyRunning_time+0.0);
            sum_T_time += T_time;
            sum_QT_time += QT_time;
            printf("%s\t%d\t%d\t  %d\t   %d\t %d\t  %.2f\n",curpro->name,curpro->enter_time,curpro->copyRunning_time,
                   curpro->start_time,curpro->done_time,T_time,QT_time);
            if(queue->size==0&&pronum<num){   //防止出现前一个进程执行完到下一个进程到达之间无进程进入
                pro[pronum].start_time = pro[pronum].enter_time;
                EnterQueue(queue,&pro[pronum]);
                pronum++;
            }
            continue;
        }
        // 运行时间大于时间片
        for(int tt = time;tt<=time+timeslice&&pronum<num;tt++){    //模拟进程的执行过程
            if(tt>=pro[pronum].enter_time){ // 统计从此任务开始到结束之间有几个进程到达
                pro[pronum].start_time = tt;
                EnterQueue(queue,&pro[pronum]);
                pronum++;
            }
        }
        time += timeslice;
        curpro->running_time -= timeslice;
        //当前程序未完成  继续添加到队列中
        EnterQueue(queue,curpro);
        if(queue->size==0&&pronum<num){   //防止出现前一个进程执行完到下一个进程到达之间无进程进入
            pro[pronum].start_time = pro[pronum].enter_time;
            EnterQueue(queue,&pro[pronum]);
            pronum++;
        }
    }
```

```
360.        printf("平均周转时间为%.2f\t 平均带权周转时间为%.2f\n\n",sum_T_time/(num+0.0),sum_QT_time/(num+0.0));
361.    }
362.    void choiceMenu(){
363.        printf("请选择进程调度算法：\n\n");
364.        printf("1.先来先服务算法\n2.短进程优先算法\n3.高优先级优先\n4.时间片轮转算法\n5.高响应比优先算法\n6.退出\n");
365.    }
366.    void menu(){
367.        int proNum;
368.        printf("请输入进程的个数：");
369.        scanf("%d",&proNum);
370.        PCB pro[proNum];
371.        inputPCB(pro,proNum);
372.        choiceMenu();
373.        int choice;
374.        while(1){
375.            scanf("%d",&choice);
376.            switch(choice){
377.                case 1:FCFS(pro,proNum);choiceMenu();break;
378.                case 2:SJF(pro,proNum);choiceMenu();break;
379.                case 3:HPF(pro,proNum);choiceMenu();break;
380.                case 4:RR(pro,proNum);choiceMenu();break;
381.                case 5:HRRN(pro,proNum);choiceMenu();break;
382.                case 6:return;
383.            }
384.        }
385.    }
386.    int main(){
387.        menu();
388.        return 0;
389.    }
```