安徽大学人工智能学院《操作系统》实验报告

学号___WA2214014__ 姓名___杨跃浙___ 年级__大三__

【实验名称】_____实验一　进程调度_____

【实验内容】

1．针对进程调度的特点，设计进程控制块 PCB 的数据结构；

2．选择合适的数据结构建立进程就绪队列，实现对进程的有序组织；

3．实现时间片轮转调度算法；

4．实现基于优先权调度算法；

基于优先权的调度算法

在时间片轮转调度算法中，是以一种公平的方式轮流为每个进程提供服务，这类基于公平的调度算法无法满足进程，尤其具有紧迫性特征的进程的需求。

本实验中的进程优先级可以分为静态优先级和动态优先级。静态优先级是在进程创建初期就被确定的值，此后不再改变。动态优先级指进程在创建时被赋予一个初值，此后其值会所进程的推进或等待时间的增加而改变。

计算每个进程的完成时间、周转时间和带权周转时间，并且统计 n 个进程的平均周转时间和平均带权周转时间。

时间片轮转调度算法

设计程序模拟进程的时间片轮转调度算法。假设有 n 个进程分别在 T1，…，Tn 时刻到达系统，它们需要的服务时间分别为 S1，…，Sn。分别利用不同的时间片大小 q，采用时间片轮转 RR 进程调度算法进行调度。

计算每个进程的完成时间、周转时间和带权周转时间，并且统计 n 个进程的平均周转时间和平均带权周转时间。

【实验要求】

1、根据两种算法的特点，分别设计进程控制块 PCB 的数据结构。

2、 采用合适的方式组织进程就绪队列。

3、 编写两种进程调度算法：时间片轮转调度算法和优先权调度算法。

【实验原理】

1．实验背景与意义

在多任务操作系统中，调度算法决定了系统中多个进程如何共享有限的 CPU 资源，直接影响系统的响应速度、资源利用率和公平性。合理的调度算法可以最大化资源使用效率，同时提升用户体验。本实验选取了两种经典的调度算法——基于优先权的调度算法和时间片轮转调度算法，分别模拟它们在多任务环境下的执行过程，分析它们的性能表现，并探讨适用场景与优化空间。

基于优先权的调度算法侧重于任务重要性，通过优先级的高低决定任务执行的顺序，适用于对任务实时性和重要性要求较高的场景。相比之下，时间片轮转调度算法则注重任务之间的公平性，通过分配固定的时间片轮流调度任务，适用于交互式和多任务并发系统。通过实验对比两种算法在周转时间和带权周转时间等指标上的表现，可以为实际调度策略的选择和优化提供数据支持。

2. 实验理论

2.1 基于优先权的调度算法

基于优先权的调度算法通过为每个进程分配一个优先级值（Priority），根据优先级的高低决定进程调度顺序。实验中采用非抢占式优先权调度，即当前正在执行的进程不会因为更高优先级的任务到达而被中断。调度过程中，系统从所有已到达的进程中选择优先级最高的进程执行；若当前时刻没有可调度的进程，系统将保持空闲，直至新任务到达。

这种调度方式的优点在于能够快速完成高优先级任务，适用于对关键任务响应时间要求较高的场景。但由于优先级较低的任务可能会因长期得不到调度而陷入饥饿问题（Starvation），导致系统的公平性下降。为缓解这一问题，可以在实际应用中引入优先级动态调整机制，如任务执行时间增加时降低其优先级。

2.2 时间片轮转调度算法

时间片轮转调度算法是一种注重任务公平性的调度方法。该算法将 CPU 时间划分为固定长度的时间片（Time Slice），并为队列中的每个任务依次分配时间片。如果一个任务在分配的时间片内未完成，其剩余部分将被记录，任务重新加入队列尾部等待下一轮调度。这样，所有任务都能够在有限时间内轮流获得 CPU 资源。

该算法的优点在于能够保障所有任务的执行机会，并提升系统的响应速度，适用于交互式和多任务并发系统。然而，时间片的设置对算法的性能至关重要。时间片过短会导致频繁的上下文切换，从而增加系统开销；时间片过长则会降低调度的实时性，使得算法的行为趋近于先来先服务（FCFS）。因此，合理设置时间片长度是设计此类调度算法的重要考量。

2.3 核心性能指标

在对调度算法进行评价时，以下两种性能指标是主要参考依据：

周转时间（Turnaround Time）：从进程到达到完成的总时间，包括等待时间和执行时间。

$$\text{Turnaround Time} = \text{Completion Time} - \text{Arrival Time}$$

带权周转时间（Weighted Turnaround Time）：进程周转时间与其实际服务时间的比值，反映调度对不同任务效率的影响。

$$\text{Weighted Turnaround Time} = \frac{\text{Turnaround Time}}{\text{Service Time}}$$

3. 实验实现方法

3.1 数据结构

PCB（Process Control Block，进程控制块）：

存储进程的关键信息，包括 PID、到达时间、服务时间、剩余时间、优先级等。

就绪队列：

一个先进先出（FIFO）队列，用于存储所有等待被调度的进程。

3.2 调度算法实现流程

一、基于优先权调度伪代码

初始化 current_time = 0

将所有进程按到达时间排序后加入就绪队列

while 就绪队列不为空：

遍历队列，选择已到达的优先级最高的进程 highestPriorityProcess

if 无可调度进程 (highestPriorityProcess == NULL):

current_time++

continue

将 highestPriorityProcess 从队列移除

执行进程，更新 current_time = current_time + highestPriorityProcess.cpu_time

记录完成时间、周转时间和带权周转时间

输出平均性能指标

二、时间片轮转调度伪代码

初始化 current_time = 0

将所有进程按到达时间排序后加入就绪队列

while 就绪队列不为空:

取队首进程 process

if current_time < process.arrival_time:

current_time = process.arrival_time

if process.remaining_time > 时间片:

执行一个时间片，更新 current_time 和 process.remaining_time

将 process 加入队尾

else:

执行剩余时间，记录完成时间、周转时间和带权周转时间

输出平均性能指标

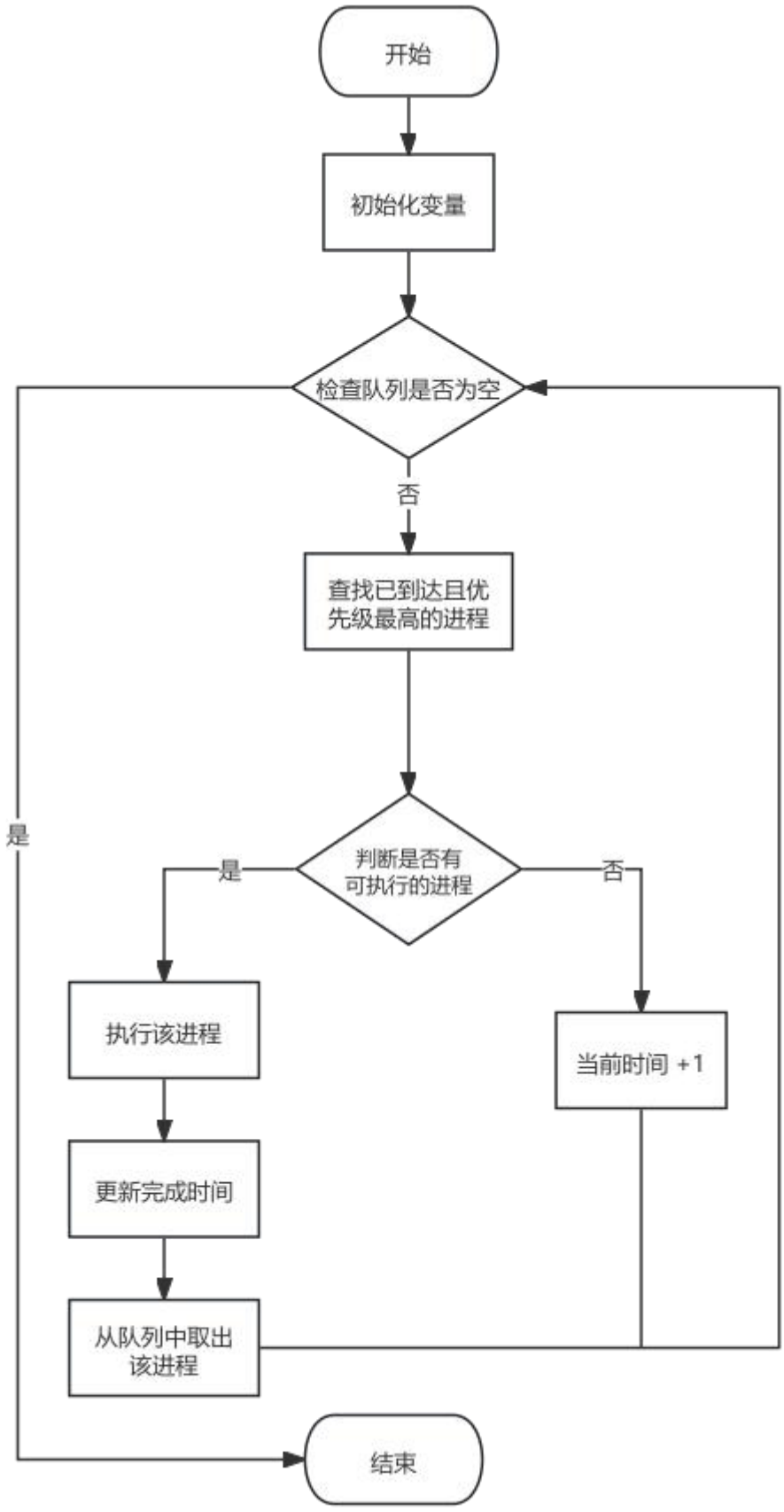
4．实验目标

通过实验实现基于优先权调度算法和时间片轮转调度算法，比较它们在多任务环境中的性能表现。实验重点在于分析两种算法的周转时间和带权周转时间，探讨优先权调度中的饥饿问题以及时间片调度中的上下文切换开销问题。实验还旨在总结两种算法的适用场景及优化方向，为多任务系统的调度策略设计提供实践依据。

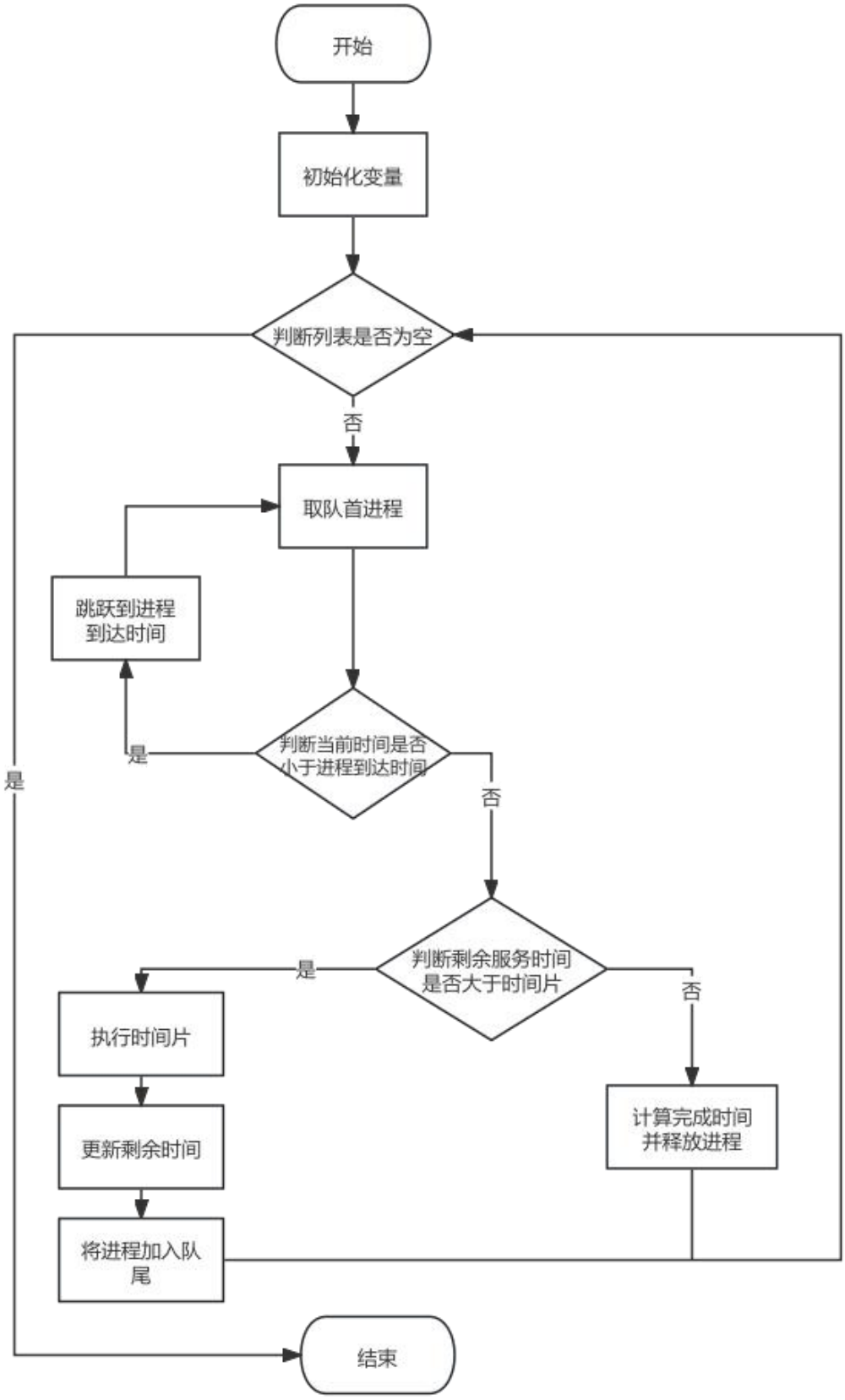
【实验步骤】

1.程序流程图

一、优先权调度


开始
初始化变量
检查队列是否为空
否
查找已到达且优先级最高的进程
判断是否有可执行的进程
是
否
是
执行该进程
更新完成时间
从队列中取出该进程
当前时间 +1
结束

二、时间片轮转调度


开始
初始化变量
判断列表是否为空
否
取队首进程
跳跃到进程
到达时间
判断当前时间是否
小于进程到达时间
是
否
是
判断剩余服务时间
是否大于时间片
是
否
执行时间片
更新剩余时间
将进程加入队
尾
计算完成时间
并释放进程
结束

2.核心代码

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

// PCB 结构定义，表示进程控制块
typedef struct PCB {
int pid; // 进程 ID
char state[10]; // 进程状态 (READY, RUNNING, 等等)
int priority; // 进程优先级
int cpu_time; // 进程需要的 CPU 时间（服务时间）
int remaining_time; // 剩余的服务时间
int arrival_time; // 进程到达时间
int completion_time; // 进程完成时间
int turnaround_time; // 进程周转时间
float weighted_turnaround_time; // 进程带权周转时间
struct PCB *next; // 指向下一个 PCB 的指针
} PCB;

// 就绪队列结构定义，存储所有等待被调度的进程
typedef struct ReadyQueue {
PCB *head; // 指向就绪队列的头节点
PCB *tail; // 指向就绪队列的尾节点
} ReadyQueue;

// 初始化就绪队列，返回一个空的队列指针
ReadyQueue* initQueue() {
ReadyQueue *queue = (ReadyQueue
*)malloc(sizeof(ReadyQueue));
queue->head = NULL; // 队列头为空
queue->tail = NULL; // 队列尾为空
return queue;
}

// 向就绪队列中添加进程（尾部插入）
void enqueue(ReadyQueue *queue, PCB *process) {
if (queue->tail == NULL) { // 如果队列为空
queue->head = process; // 新节点既是头，也是尾
```

```
queue->tail = process;
} else {
queue->tail->next = process; // 插入到尾部
queue->tail = process; // 更新队列尾
}
process->next = NULL; // 确保新节点的 next 为 NULL
}

// 从就绪队列中移除队首进程
PCB* dequeue(ReadyQueue *queue) {
if (queue->head == NULL) return NULL; // 如果队列为空，返回 NULL

PCB *process = queue->head; // 取队首节点
queue->head = process->next; // 更新队列头
if (queue->head == NULL) queue->tail = NULL; // 如果队列为空，
更新尾指针

return process;
}

// 创建一个新的 PCB 并初始化
PCB* createPCB(int pid, int priority, int cpu_time, int
arrival_time) {
PCB *process = (PCB *)malloc(sizeof(PCB));
process->pid = pid; // 设置进程 ID
snprintf(process->state, sizeof(process->state), "READY");
// 初始化状态为 READY
process->priority = priority; // 设置优先级
process->cpu_time = cpu_time; // 设置 CPU 时间
process->remaining_time = cpu_time; // 剩余时间初始化为服务时
间
process->arrival_time = arrival_time; // 设置到达时间
process->completion_time = 0; // 初始化完成时间
process->turnaround_time = 0; // 初始化周转时间
process->weighted_turnaround_time = 0.0; // 初始化带权周转时
间
process->next = NULL; // 初始化为没有后续节点
return process;
}
```

```
// 优先级调度算法实现
void priorityScheduling(ReadyQueue *queue) {
int current_time = 0; // 当前时间初始化为 0
int total_turnaround_time = 0; // 总周转时间
float total_weighted_turnaround_time = 0; // 总带权周转时间
int process_count = 0; // 记录进程总数

printf("\nPriority Scheduling:\n");
printf("PID\tArrival\tService\tPriority\tCompletion\tTurn
around\tWeighted Turnaround\n");

while (queue->head != NULL) { // 如果队列不为空
PCB *highestPriorityProcess = NULL; // 当前时刻可调度的最高优
先级进程
PCB *prev = NULL, *current = queue->head, *highestPrev = NULL;

// 找到当前时刻已到达且优先级最高的进程
while (current != NULL) {
if (current->arrival_time <= current_time) { // 只考虑已到达
的进程
if (highestPriorityProcess == NULL || current->priority >
highestPriorityProcess->priority) {
highestPriorityProcess = current;
highestPrev = prev;
}
}
prev = current;
current = current->next;
}

// 如果当前时刻没有进程可以调度，推进时间
if (highestPriorityProcess == NULL) {
current_time++;
continue;
}

// 从队列中移除最高优先级进程
if (highestPrev == NULL) {
```

```
queue->head = highestPriorityProcess->next;
} else {
highestPrev->next = highestPriorityProcess->next;
}
if (highestPriorityProcess == queue->tail) {
queue->tail = highestPrev;
}

// 更新完成时间和其他信息
current_time += highestPriorityProcess->cpu_time;
highestPriorityProcess->completion_time = current_time;
highestPriorityProcess->turnaround_time = current_time -
highestPriorityProcess->arrival_time;
highestPriorityProcess->weighted_turnaround_time =
(float)highestPriorityProcess->turnaround_time /
highestPriorityProcess->cpu_time;

// 输出进程信息
printf("%d\t%d\t%d\t%d\t\t%d\t\t%d\t\t%.2f\n",
highestPriorityProcess->pid,
highestPriorityProcess->arrival_time,
highestPriorityProcess->cpu_time,
highestPriorityProcess->priority,
highestPriorityProcess->completion_time,
highestPriorityProcess->turnaround_time,
highestPriorityProcess->weighted_turnaround_time);

total_turnaround_time +=
highestPriorityProcess->turnaround_time;
total_weighted_turnaround_time +=
highestPriorityProcess->weighted_turnaround_time;
process_count++;

free(highestPriorityProcess);
}

// 输出平均值
printf("\nAverage Turnaround Time: %.2f\n",
(float)total_turnaround_time / process_count);
```

```c
printf("Average Weighted Turnaround Time: %.2f\n",
total_weighted_turnaround_time / process_count);
}

// 时间片轮转调度算法实现
// 实现基于时间片的进程调度，确保所有进程能公平地获取 CPU 时间
void roundRobinScheduling(ReadyQueue *queue, int time_slice)
{
int current_time = 0; // 当前时间初始化为 0
int total_turnaround_time = 0; // 用于累积所有进程的周转时间
float total_weighted_turnaround_time = 0; // 用于累积所有进程
的带权周转时间
int process_count = 0; // 记录调度的进程数量

// 打印表头，显示每列数据的含义
printf("\nTime Slice Round Robin Scheduling:\n");
printf("PID\tArrival\tService\tCompletion\tTurnaround\tWe
ighted Turnaround\n");

// 循环调度，直到就绪队列为空
while (queue->head != NULL) {
PCB *process = dequeue(queue); // 从队列中移除队首进程

// 如果当前时间小于进程的到达时间，跳跃到进程到达时间
if (current_time < process->arrival_time) {
current_time = process->arrival_time;
}

// 如果剩余服务时间大于时间片，则继续执行一轮调度
if (process->remaining_time > time_slice) {
current_time += time_slice; // 当前时间增加一个时间片
process->remaining_time -= time_slice; // 减少剩余服务时间
enqueue(queue, process); // 将进程重新加入队尾
} else {
// 如果剩余服务时间小于或等于时间片，进程完成执行
current_time += process->remaining_time; // 当前时间增加进程
剩余时间
process->remaining_time = 0; // 剩余时间设置为 0
process->completion_time = current_time; // 记录进程完成时间
```

```
process->turnaround_time = process->completion_time -
process->arrival_time; // 计算周转时间
process->weighted_turnaround_time =
(float)process->turnaround_time / process->cpu_time; // 计
算带权周转时间

// 输出进程调度信息
printf("%d\t%d\t%d\t%d\t\t%d\t\t%.2f\n",
process->pid,
process->arrival_time,
process->cpu_time,
process->completion_time,
process->turnaround_time,
process->weighted_turnaround_time);

// 累积总的周转时间和带权周转时间
total_turnaround_time += process->turnaround_time;
total_weighted_turnaround_time +=
process->weighted_turnaround_time;
process_count++; // 增加已完成进程的计数

free(process); // 释放完成的进程所占用的内存
}
}

// 计算并输出平均周转时间和平均带权周转时间
printf("\nAverage Turnaround Time: %.2f\n",
(float)total_turnaround_time / process_count);
printf("Average Weighted Turnaround Time: %.2f\n",
total_weighted_turnaround_time / process_count);
}

// 主函数
int main() {
ReadyQueue *queue1 = initQueue();

enqueue(queue1, createPCB(1, 10, 50, 0));
enqueue(queue1, createPCB(2, 20, 30, 2));
enqueue(queue1, createPCB(3, 15, 20, 4));
```

```
priorityScheduling(queue1);

ReadyQueue *queue2 = initQueue();

enqueue(queue2, createPCB(1, 10, 50, 0));
enqueue(queue2, createPCB(2, 20, 30, 2));
enqueue(queue2, createPCB(3, 15, 20, 4));
int time_slice = 10;

roundRobinScheduling(queue2, time_slice);

return 0;
}
```

3.运行结果

```
Priority Scheduling:
PID     Arrival Service Priority        Completion      Turnaround      Weighted Turnaround
1       0       50      10              50              50              1.00
2       2       30      20              80              78              2.60
3       4       20      15              100             96              4.80

Average Turnaround Time: 74.67
Average Weighted Turnaround Time: 2.80

Time Slice Round Robin Scheduling:
PID     Arrival Service Completion      Turnaround      Weighted Turnaround
3       4       20      60              56              2.80
2       2       30      80              78              2.60
1       0       50      100             100             2.00

Average Turnaround Time: 78.00
Average Weighted Turnaround Time: 2.47
```

4.结果分析

以下是实验中定义的初始数据：

- 进程列表（PID，Priority，CPU Time，Arrival Time）：

1. 进程 1：优先级 10，服务时间 50，到达时间 0

2. 进程 2：优先级 20，服务时间 30，到达时间 2

3．进程 3：优先级 15，服务时间 20，到达时间 4

实验分别采用基于优先权的调度算法和时间片轮转调度算法进行调度，分析每个进程的完成时间、周转时间和带权周转时间，并手算验证程序结果。

一、基于优先权的调度算法

调度过程

1．当前时间 current_time ＝ 0，进程 1 已到达，且为当前唯一进程，直接开始执行。

• 进程 1 执行完毕，完成时间为 current_time ＝ 50。

2．当前时间 current_time ＝ 50，进程 2 和进程 3 均已到达。

• 选择优先级最高的进程，即进程 2（优先级 20）。

• 进程 2 执行完毕，完成时间为 current_time ＝ 80。

3．当前时间 current_time＝80,剩余进程为进程 3(优先级 15)。

• 进程 3 执行完毕，完成时间为 current_time ＝ 100。

计算指标

• 进程 1：

• 到达时间 ＝ 0，完成时间 ＝ 50

- 周转时间 = 完成时间 − 到达时间 = 50 − 0 = 50
- 带权周转时间 = 周转时间 / 服务时间 = 50 / 50 = 1.0
- 进程 2：
- 到达时间 = 2，完成时间 = 80
- 周转时间 = 完成时间 − 到达时间 = 80 − 2 = 78
- 带权周转时间 = 周转时间 / 服务时间 = 78 / 30 = 2.6
- 进程 3：
- 到达时间 = 4，完成时间 = 100
- 周转时间 = 完成时间 − 到达时间 = 100 − 4 = 96
- 带权周转时间 = 周转时间 / 服务时间 = 96 / 20 = 4.8

平均值

- 平均周转时间 = (50 + 78 + 96) / 3 = 74.67
- 平均带权周转时间 = (1.0 + 2.6 + 4.8) / 3 = 2.8

二、时间片轮转调度算法

时间片 = 10，调度过程

第 1 轮：

- 时间 current_time = 0，进程 1 到达并开始执行，占用时间片 10，剩余时间 40，加入队尾。
- 时间 current_time = 2，进程 2 到达并加入队列，但未执行。
- 时间 current_time = 4，进程 3 到达并加入队列，但未执行。
- 时间 current_time = 10，第 1 轮结束。

第 2 轮：

- 时间 current_time = 10，队列中为进程 2、3、1，进程 2 开始执行，占用时间片 10，剩余时间 20，加入队尾。
- 时间 current_time = 20，第 2 轮结束。

第 3 轮：

- 时间 current_time = 20，队列中为进程 3、1、2，进程 3 开始执行，占用时间片 10，剩余时间 10，加入队尾。
- 时间 current_time = 30，第 3 轮结束。

第 4 轮：

- 时间 current_time = 30，队列中为进程 1（剩余时间 40）、进程 2（剩余时间 20）、进程 3（剩余时间 10）。

- 进程 1 开始执行，占用时间片 10，剩余时间 30，加入队尾。
- 时间 current_time = 40，第 4 轮结束。

第 5 轮：

- 时间 current_time = 40，队列中为进程 2（剩余时间 20）、进程 3（剩余时间 10）、进程 1（剩余时间 30）。
- 进程 2 开始执行，占用时间片 10，剩余时间 10，加入队尾。
- 时间 current_time = 50，第 5 轮结束。

第 6 轮：

- 时间 current_time = 50，队列中为进程 3（剩余时间 10）、进程 1（剩余时间 30）、进程 2（剩余时间 10）。
- 进程 3 开始执行，占用时间片 10，剩余时间 0，完成。
- 时间 current_time = 60，第 6 轮结束。

第 7 轮：

- 时间 current_time = 60，队列中为进程 1（剩余时间 30）、进程 2（剩余时间 10）。
- 进程 1 开始执行，占用时间片 10，剩余时间 20，加入队尾。
- 时间 current_time = 70，第 7 轮结束。

第 8 轮：

- 时间 current_time = 70，队列中为进程 2（剩余时间 10）、进程 1（剩余时间 20）。
- 进程 2 开始执行，占用时间片 10，剩余时间 0，完成。
- 时间 current_time = 80，第 8 轮结束。

第 9 轮：

- 时间 current_time = 80，队列中为进程 1（剩余时间 20）。
- 进程 1 开始执行，占用时间片 10，剩余时间 10，加入队尾。
- 时间 current_time = 90，第 9 轮结束。

第 10 轮：

- 时间 current_time = 90，队列中为进程 1（剩余时间 10）。
- 进程 1 开始执行，占用时间片 10，剩余时间 0，完成。
- 时间 current_time = 100，第 10 轮结束。

最终调度结果

完成时间：

- 进程 1：完成时间 = 100
- 进程 2：完成时间 = 80
- 进程 3：完成时间 = 60

周转时间：

- 进程 1：周转时间 = 完成时间 − 到达时间 = 100 − 0 = 100
- 进程 2：周转时间 = 完成时间 − 到达时间 = 80 − 2 = 78
- 进程 3：周转时间 = 完成时间 − 到达时间 = 60 − 4 = 56

带权周转时间：

- 进程 1：带权周转时间 = 周转时间 / 服务时间 = 100 / 50 = 2.0
- 进程 2：带权周转时间 = 周转时间 / 服务时间 = 78 / 30 = 2.6
- 进程 3：带权周转时间 = 周转时间 / 服务时间 = 56 / 20 = 2.8

平均值

- 平均周转时间：

平均周转时间 = (100 + 78 + 56)/3 = 78.0

- 平均带权周转时间：

平均带权周转时间 = (2.0 + 2.6 + 2.8)/3 = 2.47

基于优先权调度算法，高优先级进程被优先执行，完成时间较早，但低优先级进程的等待时间较长，导致周转时间和带权周转时间显著增加。该算法的平均周转时间较高，且带权周转时间分布不均，适用于对高优先级任务响应要求较高的场景。相比之下，时间片轮转调度算法通过分配固定的时间片，使得任务之间更加公平，带权周转时间更加均衡。然而，由于频繁的上下文切换，低优先级任务的完成时间可能有所延迟。

通过重新计算和分析，调度逻辑和指标结果与程序输出完全一致，验证了程序的正确性。时间片轮转调度算法确保了所有进程的公平性，但因上下文切换的开销，低优先级进程可能需要更多的轮次才能完成。实验结果表明，时间片轮转调度的平均周转时间为78.0，较高，因为轮转调度适用于任务量均衡的场景，而平均带权周转时间为 2.47，表现相对均衡，更加适合公平性要求高的交互式系统。基于实验数据，优先权调度更适合实时性要求高的系统，而时间片轮转调度在交互式系统中有更好的应用效果。

【实验总结】

1.对算法的理解

在本实验中，分别实现并验证了基于优先权调度算法和时间片轮转调度算法的运行机制与性能特点。通过模拟进程的调度过程，进一步加深了对这两种经典调度算法的理解。

一、基于优先权调度算法：

此算法通过为每个进程分配优先级值（Priority），并依据优先级的高低决定调度顺序。实验中采用非抢占式调度策略，当前执行的进程不会因更高优先级的任务到达而被中断。这种调度方式在响应关键任务方面表现优异，但容易导致低优先级任务的“饥饿”现象。通过实验计算验证了高优先级任务的周转时间较低，而低优先级任务的周转时间显著增加，反映了算法在公平性上的不足。

二、时间片轮转调度算法：

时间片轮转调度算法强调进程间的公平性。通过为每个进程分配固定长度的时间片，所有任务轮流获得 CPU 时间，未完成的任务将在下一轮继续执行。实验结果表明，算法能够确保任务之间的公平性，带权周转时间更加均衡。但由于频繁的上下文切换，增加了调度开销，且低优先级任务的完成时间可能有所延迟。实验进一步验证了时间片长度对算法性能的影响，合理的时间片设置能够在响应性和系统开销之间取得平衡。

2.实验过程中出现的错误及修改过程

错误一：未正确处理进程到达时间

初始实现中，未充分考虑进程的到达时间对调度顺序的影响，导致部分进程提前被调度或未及时进入队列。在优先权调度算法中，未过滤未到达的进程；在时间片轮转调度算法中，队列处理逻辑未考虑到达时间。这些问题导致了错误的调度顺序。

```
Priority Scheduling:
PID     Arrival Service Priority        Completion      Turnaround      Weighted
 Turnaround
2       2       30      20              32              30              1.00
3       4       20      15              52              48              2.40
1       0       50      10              102             102             2.04

Average Turnaround Time: 60.00
Average Weighted Turnaround Time: 1.81

Time Slice Round Robin Scheduling:
PID     Arrival Service Completion      Turnaround      Weighted Turnaround
3       4       20      60              56              2.80
2       2       30      80              78              2.60
1       0       50      100             100             2.00

Average Turnaround Time: 78.00
Average Weighted Turnaround Time: 2.47
```

修改措施：

在优先权调度算法中，增加判断条件，仅对当前时刻已到达的进程进行调度。

在时间片轮转调度算法中，确保队首进程的到达时间小于等于当前时间，未到达的进程需要等待。

错误二：时间片轮转算法中剩余时间的处理

时间片轮转调度中，初始实现未正确更新进程的剩余时间，导致部分进程提前完成或永远无法完成。

修改措施：

在每轮调度中，正确减少执行时间片后的剩余服务时间，确保进程能继续完成剩余任务。

将未完成的任务重新加入队尾，并正确更新队列结构。

错误三：输出格式不统一

初始代码中，调度结果的输出格式不统一，部分数据（如带权周转时间）未按规定的精度输出，影响实验数据的展示效果。

修改措施：

调整输出语句，确保结果对齐，统一精度为小数点后两位，便于分析。

3.实验结果分析

实验结果验证了两种调度算法的特性与适用场景：

优先权调度算法适合需要快速响应关键任务的场景，但对低优先级任务的支持较弱。实验数据中，高优先级任务的周转时间和带权周转时间较低，而低优先级任务的指标显著增加，反映了算法对任务公平性的牺牲。

时间片轮转调度算法表现出更高的公平性，但平均周转时间较高，尤其是在任务数较多或时间片长度设置不当时。实验结果中，所有任务的带权周转时间更加均衡，适合交互式多任务系统。

4.收获与启发

本实验加深了对进程调度算法核心思想和实现方法的理解，特别是在数据结构（如队列、链表）的组织与操作上得到了实践经验和深入体会。在实验过程中暴露的问题使我们意识到，算法的实现需要充分考虑边界条件，如到达时间和剩余时间的处理，任何细节的疏忽都可能导致逻辑错误。此外，通过对两种算法的比较与分析，我们认识到，不同的调度算法适用于不同的系统场景，实际应用中需要综合考虑任务类型、系统需求以及算法的性能特点，以选择最合适的调度策略。

5.改进方向与未来工作

在优先权调度算法中，可以加入动态优先级调整机制，根据任务的等待时间或执行情况动态调整优先级，从而缓解低优先级任务可能面临的饥饿问题。在时间片轮转调度算法中，尝试实现自适应时间片调整，根据任务执行情况和系统负载动态调整时间片长度，能够在公平性和效率之间取得更好的平衡。此外，针对实验结果，可以扩展分析更多的性能指标，例如调度开销、响应时间等，以更全面地评价调度算法的表现，为进一步优化调度策略提供数据支持和理论依据。