

**HUBEI UNIVERSITY OF AUTOMOTIVE TECHNOLOGY**



**操作系统原理**

**实 验 报 告**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 实验项目 | 进程调度实验 | | | |
| 学生姓名 | 章崇文 | 学生学号 | 202202296 | |
| 学生班级 | 计算机222 | 完成日期 | 2024.5.22 | |
| 实验成绩 |  | 评阅日期 | |  |
| 评阅教师 |  | | | |

**湖北汽车工业学院实验报告**

班 号 学 号   姓 名

选课班中的序号  完成日期 年 月 日 至 节

# 实验四 进程调度实验

一、实验目的

1、了解操作系统CPU管理的主要内容。

2、加深理解操作系统管理控制进程的数据结构--PCB。

3、掌握几种常见的CPU调度算法（FCFS、SJF、HRRF、HPF、RR）的基本思想和实现过程。

4、用C语言模拟实现CPU调度算法。

5、掌握CPU调度算法性能评价指标的计算方法。

6、通过对进程调度算法的模拟加深对进程概念和进程调度算法的理解。

**二、实验内容**

1、用C语言编写程序，模拟单处理器下先来先服务算法FCFS，要求显示各进程的到达时间、服务时间、完成时间，周转时间以及该算法的平均周转时间和平均带权周转时间。运行以下参考程序，给出结果截图并分析该算法的优缺点。（3分）

参考程序：

#include <stdio.h>

#include <stdlib.h>

struct PCB // 先来先服务FCFS

{

char name[10]; // 进程名

float arrivetime; // 到达时间

float servetime; // 服务时间

float finishtime; // 完成时间

float roundtime; // 周转时间

float daiquantime; // 带权周转时间

};

struct PCB a[50]; // 定义进程数组

struct PCB \*sortarrivetime(struct PCB a[], int n); // 声明到达时间冒泡排序函数

void FCFS(struct PCB a[], int n, float \*t1, float \*t2); // 先来先服务算法

// 按到达时间进行冒泡排序

struct PCB \*sortarrivetime(struct PCB a[], int n)

{

int i, j;

struct PCB t;

int flag; // 标志变量，记录在每一趟冒泡中是否有元素交换，没有交换则结束冒泡

for (i = 1; i < n; i++) // 外层循环控制比较趟数

{

flag = 0; // 初始值设置为0

for (j = 0; j < n - i; j++) // 内存循环控制每一趟的比较次数

{

if (a[j].arrivetime > a[j + 1].arrivetime) // 将到达时间短的交换到前边

{

t = a[j];

a[j] = a[j + 1];

a[j + 1] = t;

flag = 1; // 有交换，flag置1

}

}

if (flag == 0) // 如果一趟排序中没发生任何交换，则排序结束

{

break;

}

}

return a; // 返回排序后进程数组

}

// 先来先服务算法

void FCFS(struct PCB a[], int n, float \*t1, float \*t2)

{

int i;

a[0].finishtime = a[0].arrivetime + a[0].servetime; // 完成时间=到达时间+服务时间

a[0].roundtime = a[0].finishtime - a[0].arrivetime; // 周转时间=完成时间-到达时间

a[0].daiquantime = a[0].roundtime / a[0].servetime; // 带权时间=周转时间/服务时间

for (i = 1; i < n; i++)

{

if (a[i].arrivetime < a[i - 1].finishtime) // 当前到达时间在上一个作业结束时间之前

{

a[i].finishtime = a[i - 1].finishtime + a[i].servetime; // 完成时间=上一个完成时间+服务时间

a[i].roundtime = a[i].finishtime - a[i].arrivetime; // 周转时间=完成时间-到达时间

a[i].daiquantime = a[i].roundtime / a[i].servetime; // 带权时间=周转时间/服务时间

}

else // 当前到达时间在上一个作业结束时间之后

{

a[i].finishtime = a[i].arrivetime + a[i].servetime; // 完成时间=到达时间+服务时间

a[i].roundtime = a[i].finishtime - a[i].arrivetime; // 周转时间=完成时间-到达时间

a[i].daiquantime = a[i].roundtime / a[i].servetime; // 带权时间=周转时间/服务时间

}

}

printf("=============================================================\n");

printf("进程相关信息如下：\n\n");

printf("进程名 ");

printf("到达时间 ");

printf("服务时间 ");

printf("完成时间 ");

printf("周转时间 ");

printf("带权周转时间\n");

for (i = 0; i < n; i++)

{

printf("%-10s", a[i].name);

printf("%-10.0f", a[i].arrivetime);

printf("%-10.0f", a[i].servetime);

printf("%-10.0f", a[i].finishtime);

printf("%-10.0f", a[i].roundtime);

printf("%10.2f\n", a[i].daiquantime);

\*t1 += a[i].roundtime;

\*t2 += a[i].daiquantime;

}

}

int main()

{

float t1; // 总周转时间

float t2; // 总带权周转时间

float avr\_t1; // 平均周转时间

float avr\_t2; // 平均带权周转时间

int n, i;

char select = ' '; // 选择算法变量标识

while (select != '2') // 不为退出标识，保持循环

{

t1 = 0.0f;

t2 = 0.0f;

system("clear");

printf("\n请选择算法：1.先来先服务算法 2.退出程序\n\n请输入选择: ");

scanf("%c", &select);

if (select == '1') // 先来先服务算法

{

printf("\n=====================先来先服务算法FCFS=====================\n\n");

printf("请输入进程数：");

scanf("%d", &n);

for (i = 0; i < n; i++)

{

printf("%d 进程名:", i + 1);

scanf("%s", a[i].name);

printf("到达时间：");

scanf("%f", &a[i].arrivetime);

printf("服务时间：");

scanf("%f", &a[i].servetime);

}

getchar();

sortarrivetime(a, n); // 按到达时间先后进行冒泡排序

FCFS(a, n, &t1, &t2); // 先来先服务算法

avr\_t1 = t1 / n;

avr\_t2 = t2 / n;

printf("\n");

printf("平均周转时间为：%2.2f\n", avr\_t1);

printf("平均带权周转时间为：%2.2f\n", avr\_t2);

getchar();

}

else if (select == '2')

{

exit(0);

}

else

{

printf("please enter right choose!\n");

}

}

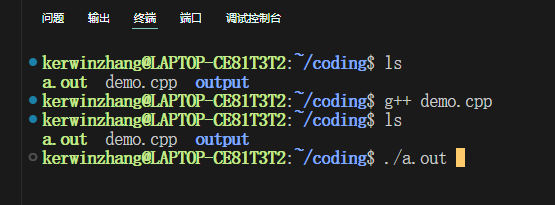
return 0;

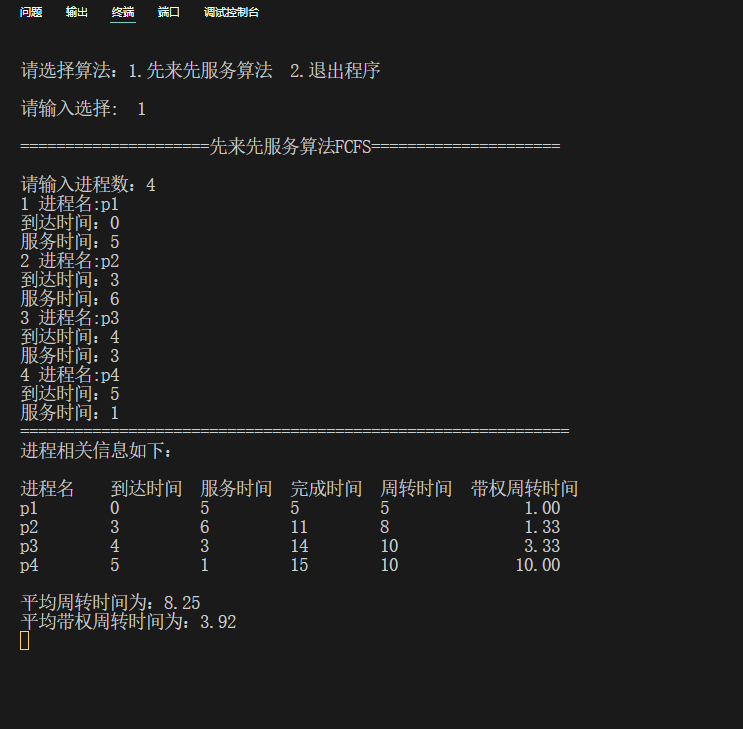
}

要求给出测试数据，如下所示（自行设定测试数据）：

|  |  |  |
| --- | --- | --- |
| 进程 | 到达时间 | 服务时间 |
| P1 | 0 | 5 |
| P2 | 3 | 6 |
| P3 | 4 | 3 |
| P4 | 5 | 1 |

要求给出编译及运行过程和运行结果：





分析该程序：

这个程序实现了 FCFS (First-Come, First-Served) 调度算法的模拟。

**数据结构:** 该程序定义了一个 PCB 结构体,用于存储每个进程的信息,包括进程名、到达时间、服务时间、完成时间、周转时间和带权周转时间。

**排序函数:** sortarrivetime() 函数实现了一个冒泡排序算法,按照进程的到达时间对进程数组进行排序。这样可以确保进程按照先到达的顺序进行调度。

**FCFS 算法实现:** FCFS() 函数实现了 FCFS 调度算法的核心逻辑。它根据进程的到达时间和服务时间,计算出每个进程的完成时间、周转时间和带权周转时间,并将结果输出到控制台。

**主程序逻辑:** main() 函数是程序的入口点。它提供了一个交互式的界面,允许用户选择 FCFS 算法或退出程序。当选择 FCFS 算法时,它会读取用户输入的进程信息,调用 sortarrivetime() 和 FCFS() 函数,并计算并输出平均周转时间和平均带权周转时间。

分析该算法优缺点：

**优点:**

1. 简单易实现: FCFS 算法是最基本的调度算法之一,实现起来非常简单,只需要维护一个先来先服务的队列即可。这种简单性使得它被广泛应用于各种操作系统和服务器中。
2. 公平性: FCFS 算法是一种先到先得的策略,每个任务都会得到公平的对待,不会出现某些任务被特别优先或者延迟的情况。这符合人们的直观感受。
3. 易于理解和分析: 由于 FCFS 算法的简单性,它的运行机制和性能分析都比较容易理解和推导。这有利于系统设计和性能优化。

**缺点:**

1. 平均响应时间长: FCFS 算法没有考虑任务的特性和优先级,因此可能会导致某些短时间内就能完成的任务被长时间任务阻塞,平均响应时间较长。
2. 不利于交互式任务: 对于需要快速响应的交互式任务,FCFS 算法可能会导致用户体验较差,因为这些任务需要尽快得到响应。
3. 缺乏灵活性: FCFS 算法无法根据任务的特性、优先级等因素进行调度,缺乏灵活性。在某些场景下可能无法满足系统的性能需求。
4. 饥饿问题: 如果存在大量的长时间任务不断进入系统,那么短时间任务可能会一直被饿死,无法得到及时的服务。

**说明：下面代码为了简便，取消了输入数据，改为写入vector里面，方便运行。**

1. 编程实现最短作业优先算法 SJF。要求给出编译，运行步骤和执行结果截图。（4分）

**代码：**

#include <iostream>

#include <vector>

#include <algorithm>

#include <string>

#include <climits>

using namespace std;

struct Process

{

string id;

int arrivalTime;

int burstTime;

int waitingTime;

int turnaroundTime;

float weightedTurnaroundTime;

};

bool compareArrivalTime(const Process &a, const Process &b)

{

return a.arrivalTime < b.arrivalTime;

}

bool compareBurstTime(const Process &a, const Process &b)

{

return a.burstTime < b.burstTime;

}

void sjfScheduling(vector<Process> &processes)

{

sort(processes.begin(), processes.end(), compareArrivalTime);

int currentTime = 0;

int totalProcesses = processes.size();

int completedProcesses = 0;

vector<bool> completed(totalProcesses, false);

while (completedProcesses < totalProcesses)

{

int shortestJobIndex = -1;

int shortestJobBurst = INT\_MAX;

for (int i = 0; i < totalProcesses; ++i)

{

if (!completed[i] && processes[i].arrivalTime <= currentTime && processes[i].burstTime < shortestJobBurst)

{

shortestJobIndex = i;

shortestJobBurst = processes[i].burstTime;

}

}

if (shortestJobIndex != -1)

{

cout << "在时间 " << currentTime << " 执行进程 " << processes[shortestJobIndex].id << endl;

processes[shortestJobIndex].waitingTime = currentTime - processes[shortestJobIndex].arrivalTime;

currentTime += processes[shortestJobIndex].burstTime;

processes[shortestJobIndex].turnaroundTime = currentTime - processes[shortestJobIndex].arrivalTime;

processes[shortestJobIndex].weightedTurnaroundTime = (float)processes[shortestJobIndex].turnaroundTime / processes[shortestJobIndex].burstTime;

completed[shortestJobIndex] = true;

++completedProcesses;

}

else

{

++currentTime;

}

}

}

int main()

{

**vector<Process> processes = {{"p1", 0, 5}, {"p2", 3, 6}, {"p3", 4, 3}, {"p4", 5, 1}};**

sjfScheduling(processes);

float totalWaitingTime = 0;

float totalTurnaroundTime = 0;

float totalWeightedTurnaroundTime = 0;

for (const auto &process : processes)

{

totalWaitingTime += process.waitingTime;

totalTurnaroundTime += process.turnaroundTime;

totalWeightedTurnaroundTime += process.weightedTurnaroundTime;

}

float avgWaitingTime = totalWaitingTime / processes.size();

float avgTurnaroundTime = totalTurnaroundTime / processes.size();

float avgWeightedTurnaroundTime = totalWeightedTurnaroundTime / processes.size();

cout << "平均等待时间: " << avgWaitingTime << endl;

cout << "平均周转时间: " << avgTurnaroundTime << endl;

cout << "平均带权周转时间: " << avgWeightedTurnaroundTime << endl;

return 0;

}

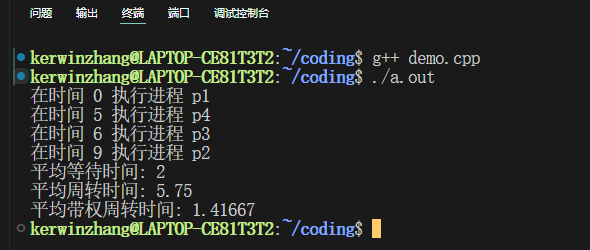
优点:

1. **最小平均等待时间**: SJF算法可以最小化所有进程的平均等待时间。这是因为短作业先执行可以减少总的等待时间。
2. **简单高效**: SJF算法的实现相对简单,只需要维护一个进程队列,并按照burst time从小到大的顺序执行即可。
3. **提高吞吐量**: 由于短作业先执行,可以增加CPU的利用率,从而提高整个系统的吞吐量。
4. **减少上下文切换**: 相比其他调度算法,SJF算法可以减少上下文切换的次数,从而降低系统开销。

缺点:

1. **不公平性**: SJF算法可能会导致一些长作业的进程长时间得不到执行,造成不公平的情况。这种情况被称为"饥饿"。
2. **无法预知burst time**: SJF算法需要提前知道每个进程的burst time,但在实际情况下,进程的burst time通常是难以预知的。
3. **缺乏实时性**: SJF算法主要关注最小化平均等待时间,而忽略了系统的实时性需求。对于一些实时性要求较高的应用程序,SJF算法可能不太适用。
4. **可能出现死锁**: 如果所有进程都有相同的burst time,那么SJF算法可能会陷入死锁的状态,无法继续执行。
5. **不适用于I/O密集型进程**: SJF算法主要针对CPU密集型进程,对于I/O密集型进程,其效果可能不太理想。

**运行截图：**



1. 编程实现最高响应比优先算法HRN，并分析算法的优缺点。（要求给出程序设计分析和调试通过的程序，并给出编译，运行步骤和执行结果截图。）（2分）

#include <iostream>

#include <vector>

#include <algorithm>

#include <iomanip>

using namespace std;

struct Process

{

string name;

int arrival\_time;

int burst\_time;

int waiting\_time;

int turnaround\_time;

double response\_ratio;

};

bool compareArrivalTime(const Process &a, const Process &b)

{

return a.arrival\_time < b.arrival\_time;

}

bool compareResponseRatio(const Process &a, const Process &b)

{

return a.response\_ratio > b.response\_ratio;

}

void calculateResponseRatio(vector<Process> &processes, int current\_time)

{

for (auto &process : processes)

{

if (process.arrival\_time <= current\_time && process.turnaround\_time == -1)

{

process.response\_ratio = (double)(current\_time - process.arrival\_time + process.burst\_time) / process.burst\_time;

}

}

}

void calculateWaitingTime(vector<Process> &processes)

{

for (size\_t i = 0; i < processes.size(); ++i)

{

processes[i].waiting\_time = processes[i].turnaround\_time - processes[i].burst\_time;

}

}

void calculateTurnaroundTime(vector<Process> &processes)

{

int current\_time = 0;

for (size\_t i = 0; i < processes.size(); ++i)

{

current\_time += processes[i].burst\_time;

processes[i].turnaround\_time = current\_time - processes[i].arrival\_time;

}

}

void scheduleProcesses(vector<Process> &processes)

{

sort(processes.begin(), processes.end(), compareArrivalTime);

int current\_time = 0;

for (size\_t i = 0; i < processes.size(); ++i)

{

calculateResponseRatio(processes, current\_time);

sort(processes.begin() + i, processes.end(), compareResponseRatio);

cout << "在时间 " << current\_time << " 执行进程 " << processes[i].name << endl;

current\_time += processes[i].burst\_time;

processes[i].turnaround\_time = current\_time - processes[i].arrival\_time;

}

calculateWaitingTime(processes);

calculateTurnaroundTime(processes);

}

int main()

{

**vector<Process> processes = {{"p1", 0, 4, -1, -1}, {"p2", 1, 6, -1, -1}, {"p3", 2, 3, -1, -1}, {"p4", 3, 2, -1, -1}};**

scheduleProcesses(processes);

float totalWaitingTime = 0;

float totalTurnaroundTime = 0;

cout << "进程\t到达时间\t执行时间\t等待时间\t周转时间" << endl;

for (const auto &process : processes)

{

cout << process.name << "\t\t" << process.arrival\_time << "\t\t" << process.burst\_time << "\t\t" << process.waiting\_time << "\t\t" << process.turnaround\_time << endl;

totalWaitingTime += process.waiting\_time;

totalTurnaroundTime += process.turnaround\_time;

}

float avgWaitingTime = totalWaitingTime / processes.size();

float avgTurnaroundTime = totalTurnaroundTime / processes.size();

cout << fixed << setprecision(2);

cout << "平均等待时间: " << avgWaitingTime << endl;

cout << "平均周转时间: " << avgTurnaroundTime << endl;

return 0;

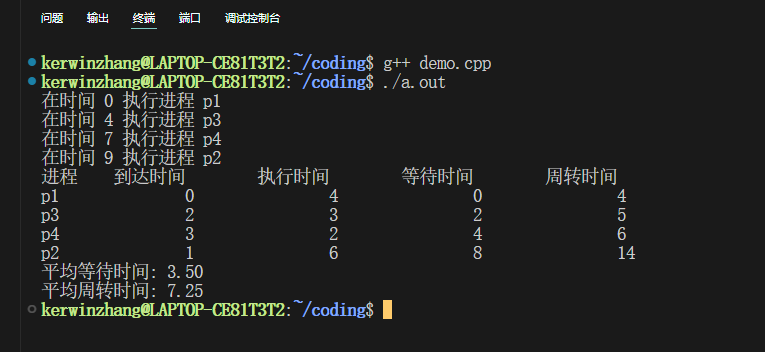
}

优点:

1. **动态调整优先级**: HRN算法会根据进程的等待时间和burst time动态调整进程的优先级,避免了固定优先级可能导致的饥饿问题。
2. **兼顾公平性和响应时间**: HRN算法通过计算响应比来确定执行顺序,既考虑了进程的优先级,也考虑了进程的等待时间,在一定程度上平衡了系统的公平性和响应时间。
3. **简单易实现**: HRN算法的实现相对简单,只需要计算每个进程的响应比并按照响应比从高到低的顺序执行即可。
4. **避免某些进程长期占用CPU**: 相比最高优先级算法,HRN算法可以避免某些高优先级进程长期独占CPU资源,从而提高了系统的整体吞吐量。

缺点:

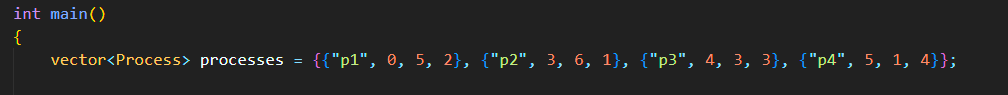
1. **无法完全保证公平性**: 虽然HRN算法考虑了等待时间因素,但对于一些长时间等待的低优先级进程而言,仍然可能面临一定的不公平问题。
2. **响应时间不可控**: 由于HRN算法会动态调整进程的优先级,对于一些实时性要求较强的应用程序而言,可能无法保证较为确定的响应时间。
3. **需要预先确定进程的burst time**: HRN算法需要知道每个进程的burst time才能计算响应比,但在实际应用中,进程的burst time可能难以预知。
4. **可能出现饥饿**: 虽然HRN算法可以减少饥饿的情况,但在某些特殊情况下,仍然可能出现某些低优先级进程长时间得不到执行的问题。
5. **不适用于I/O密集型进程**: HRN算法主要针对CPU密集型进程,对于I/O密集型进程,其效果可能不太理想。



5.编程实现更多算法，比如最高优先级算法、时间片轮换算法等，并分析算法的优缺点。（要求给出程序设计分析和调试通过的程序，并给出编译，运行步骤和执行结果截图。）（附加题）

**最高优先级算法：**

为了匹配算法，自行设定了优先级



每个 processes的参数为 进程 到达时间 服务时间 优先级、

**代码：**

**#include <iostream>**

**#include <vector>**

**#include <algorithm>**

**using namespace std;**

**struct Process**

**{**

**string name;**

**int arrival\_time;**

**int burst\_time;**

**int priority;**

**int waiting\_time;**

**int turnaround\_time;**

**};**

**bool comparePriority(const Process &a, const Process &b)**

**{**

**return a.priority < b.priority;**

**}**

**bool compareArrivalTime(const Process &a, const Process &b)**

**{**

**return a.arrival\_time < b.arrival\_time;**

**}**

**void priorityScheduling(vector<Process> &processes)**

**{**

**sort(processes.begin(), processes.end(), compareArrivalTime);**

**int current\_time = 0;**

**for (size\_t i = 0; i < processes.size(); ++i)**

**{**

**sort(processes.begin() + i, processes.end(), comparePriority);**

**cout << "在时间 " << current\_time << " 执行进程 " << processes[i].name << endl;**

**processes[i].turnaround\_time = current\_time + processes[i].burst\_time - processes[i].arrival\_time;**

**processes[i].waiting\_time = processes[i].turnaround\_time - processes[i].burst\_time;**

**current\_time += processes[i].burst\_time;**

**}**

**}**

**int main()**

**{**

**vector<Process> processes = {{"p1", 0, 5, 2}, {"p2", 3, 6, 1}, {"p3", 4, 3, 3}, {"p4", 5, 1, 4}};**

**priorityScheduling(processes);**

**float total\_waiting\_time = 0;**

**float total\_turnaround\_time = 0;**

**cout << "进程\t到达时间\t执行时间\t优先级\t等待时间\t周转时间" << endl;**

**for (const auto &process : processes)**

**{**

**cout << process.name << "\t\t" << process.arrival\_time << "\t\t" << process.burst\_time << "\t\t" << process.priority << "\t\t" << process.waiting\_time << "\t\t" << process.turnaround\_time << endl;**

**total\_waiting\_time += process.waiting\_time;**

**total\_turnaround\_time += process.turnaround\_time;**

**}**

**float avg\_waiting\_time = total\_waiting\_time / processes.size();**

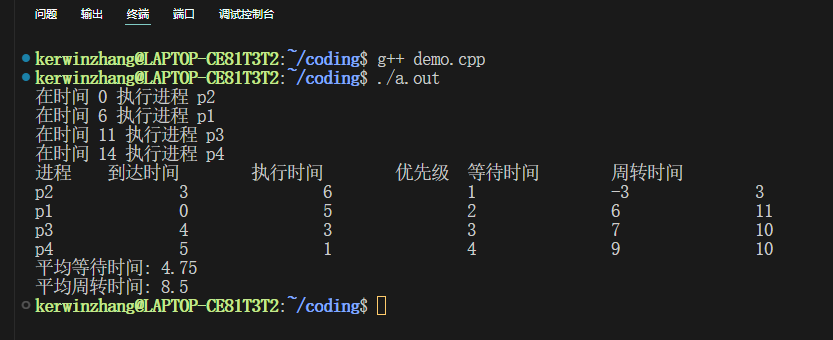
**float avg\_turnaround\_time = total\_turnaround\_time / processes.size();**

**cout << "平均等待时间: " << avg\_waiting\_time << endl;**

**cout << "平均周转时间: " << avg\_turnaround\_time << endl;**

**return 0;**

**}**



**优缺点：**

优点:

简单易实现: 最高优先级调度算法的实现比较简单,只需要根据进程的优先级进行排序即可。相比其他调度算法,它的实现复杂度较低。

有利于高优先级进程: 该算法会优先执行高优先级的进程,确保了高优先级进程能够尽快完成,满足了对高优先级任务的响应要求。

公平性: 从某种程度上来说,该算法是公平的,因为每个进程都会根据自己的优先级获得执行机会。

缺点:

低优先级进程可能饥饿: 如果持续有高优先级进程到来,那么低优先级进程可能会一直得不到执行机会,陷入饥饿状态。

难以确定合适的优先级: 在实际应用中,如何合理地确定进程的优先级是一个难题,需要考虑各种因素,如任务的重要性、deadline等。

响应时间不可控: 由于高优先级进程会一直抢占CPU,导致低优先级进程的响应时间难以保证。这可能会影响某些对响应时间有要求的应用程序。

吞吐量低: 由于高优先级进程会独占CPU资源,导致整体吞吐量可能会降低。

时间片轮换算法：

这里不给出源代码了，源码在我的github上[zhangjszs/HUAT-kerwin-labwork (github.com)](https://github.com/zhangjszs/HUAT-kerwin-labwork)

**优缺点：**

优点:

1. 公平性: 时间片轮换算法通过给每个进程分配固定的时间片,确保了进程能够得到公平的 CPU 时间,避免了某些进程独占 CPU 的情况。
2. 简单易实现: 时间片轮换算法的实现相对简单,只需要维护一个就绪队列并按照先进先出的顺序调度进程即可。
3. 良好的响应性: 由于时间片较短,进程切换频繁,可以提高系统的响应性,对于交互式应用程序较为友好。
4. 避免饥饿: 由于每个进程都会得到 CPU 时间,因此可以避免低优先级进程的饥饿问题。

缺点:

1. 时间片大小的选择: 时间片的大小是一个关键参数,如果设置过小,会增加进程切换的开销;如果设置过大,则可能导致高优先级进程的响应时间不佳。
2. CPU利用率不高: 由于需要频繁地进行进程切换,会增加系统开销,从而降低 CPU 的利用率。
3. 无法满足实时性要求: 对于一些实时性要求较高的应用程序,时间片轮换算法可能无法满足其响应时间的要求。
4. 不考虑进程的优先级: 时间片轮换算法是一种"一视同仁"的调度算法,不会根据进程的优先级做出调度决策,这可能会影响关键进程的执行。
5. 难以确定最佳时间片大小: 对于不同类型的进程,最佳的时间片大小可能会有所不同,这需要根据具体情况进行调整和优化。



三、实验总结和体会

这次 CPU 调度算法实验帮助我全面了解了操作系统 CPU 管理的核心内容。通过学习进程控制块 (PCB) 的数据结构,我深入掌握了进程管理的关键环节。同时,我熟练掌握了 FCFS、SJF、HRRF、HPF 和 RR 五种经典的 CPU 调度算法,不仅了解了它们的基本思想,还能利用 C 语言成功模拟实现。通过计算各算法的性能评价指标,我对它们的优缺点有了更清晰的认识。总的来说,这次实验达成了预期目标,让我对进程概念和进程调度算法有了更深刻的理解。

实验过程中的收获:

在实验过程中,我不仅深入理解了 PCB 在 CPU 调度中的重要作用,也掌握了 PCB 各个关键字段的含义。通过模拟实现五种调度算法,我熟练掌握了它们的基本思想和实现流程。此外,我还学会了根据作业到达时间、服务时间等指标计算各算法的性能评价指标,能够对比分析不同算法的优缺点。总的来说,这个实验过程极大地提高了我的编程能力和问题分析能力。

对未来工作的启发:

这次 CPU 调度算法实验为我今后从事操作系统开发或优化工作奠定了良好的基础。我已经掌握了常见调度算法的实现方法,未来可以根据不同场景选择合适的算法进行优化。同时,我对性能评价指标的计算和分析能力也将有助于我未来对操作系统进行性能评测和调优。总的来说,这次实验全面提升了我的操作系统知识和动手能力,为未来的工作打下了坚实的基础。