**实验五 进程调度**

# 姓 名郭宇哲 学 号22013229 成绩

实验时间 2024/11/26 指导教师(签名)

**（诚信声明：本实验报告内容，均由本人亲自上机完成。 签名：\_\_\_\_\_\_\_\_\_\_\_\_\_）**

一．实验目的

1.深入了解进程调度的策略和机制

2.掌握各种调度算法的基本原理和评价指标

3.实现几种常用的调度算法

二．实验工具与设备

装有 Linux 操作系统的计算机。

三．实验内容

1. 用 C 语言实现 FCFS、SPN 和 RR 算法

#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

#include <limits.h>

// 定义进程结构体

typedef struct {

int pid; // 进程编号

int arrival; // 到达时间

int burst; // 服务时间

int start; // 开始时间

int finish; // 完成时间

int remaining; // 剩余时间（RR 用）

bool completed; // 标记是否完成

} Process;

// 函数声明

void sortByArrival(Process processes[], int n); // 按到达时间排序

void FCFS(Process processes[], int n); // FCFS 调度算法

void SPN(Process processes[], int n); // SPN 调度算法

void RR(Process processes[], int n, int quantum); // RR 调度算法

void calculateAndPrintMetrics(Process processes[], int n); // 计算并打印平均周转时间和平均带权周转时间

**int main()** {

int n, quantum;

// 输入进程数量

printf("please enter the number of pid: ");

scanf("%d", &n);

// 初始化进程

Process processes[n];

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

printf("please enter the arrival and the burst of P%d: ", i + 1);

processes[i].pid = i + 1; // 设置进程编号

scanf("%d %d", &processes[i].arrival, &processes[i].burst);

processes[i].remaining = processes[i].burst; // 初始化剩余时间为服务时间

processes[i].completed = false; // 标记为未完成

}

// 输入时间片大小（用于 RR）

printf("please enter the size of time for RR: ");

scanf("%d", &quantum);

// FCFS 调度

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

FCFS(processes, n);

// SPN 调度

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

SPN(processes, n);

// RR 调度

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

RR(processes, n, quantum);

return 0;

}

**// 按到达时间排序（冒泡排序实现）**

**void sortByArrival(Process processes[], int n)** {

for (int i = 0; i < n - 1; i++) {

for (int j = 0; j < n - i - 1; j++) {

if (processes[j].arrival > processes[j + 1].arrival) {

// 交换两个进程的位置

Process temp = processes[j];

processes[j] = processes[j + 1];

processes[j + 1] = temp;

}

}

}

}

**// 计算并打印平均周转时间和平均带权周转时间**

**void calculateAndPrintMetrics(Process processes[], int n)** {

double total\_turnaround = 0, total\_weighted\_turnaround = 0;

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

// 计算周转时间和带权周转时间

int turnaround = processes[i].finish - processes[i].arrival; // 周转时间 = 完成时间 - 到达时间

double weighted\_turnaround = (double)turnaround / processes[i].burst; // 带权周转时间 = 周转时间 / 服务时间

total\_turnaround += turnaround; // 累加总周转时间

total\_weighted\_turnaround += weighted\_turnaround; // 累加总带权周转时间

// 输出每个进程的结果

printf("pid:P%d turnaround:%d weighted\_turnaround:%.2f\n", processes[i].pid, turnaround, weighted\_turnaround);

}

// 输出平均周转时间和平均带权周转时间

printf("average\_turnaround: %.2f\n", total\_turnaround / n);

printf("average\_weighted\_turnaround: %.2f\n", total\_weighted\_turnaround / n);

}

**// FCFS 调度**

**void FCFS(Process processes[], int n)** {

sortByArrival(processes, n); // 按到达时间排序

int current\_time = 0; // 当前时间

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

// 如果当前时间小于进程的到达时间，跳过到达前的空闲时间

if (current\_time < processes[i].arrival)

current\_time = processes[i].arrival;

// 设置开始时间和完成时间

processes[i].start = current\_time;

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

current\_time += processes[i].burst; // 更新当前时间

}

calculateAndPrintMetrics(processes, n); // 计算并输出指标

}

**// SPN 调度**

**void SPN(Process processes[], int n)** {

sortByArrival(processes, n); // 按到达时间排序

int completed = 0, current\_time = 0;

while (completed < n) {

int shortest = -1; // 当前找到的最短服务时间的进程索引

int shortest\_burst = INT\_MAX; // 最短服务时间初始化为无穷大

// 遍历所有进程，寻找可运行且服务时间最短的进程

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

if (!processes[i].completed && processes[i].arrival <= current\_time && processes[i].burst < shortest\_burst) {

shortest = i;

shortest\_burst = processes[i].burst;

}

}

// 如果没有可运行的进程，当前时间推进

if (shortest == -1) {

current\_time++;

} else {

// 运行找到的最短服务时间的进程

processes[shortest].start = current\_time;

processes[shortest].finish = current\_time + processes[shortest].burst;

current\_time += processes[shortest].burst; // 更新当前时间

processes[shortest].completed = true; // 标记为已完成

completed++; // 完成进程数 +1

}

}

calculateAndPrintMetrics(processes, n); // 计算并输出指标

}

**// RR 调度**

**void RR(Process processes[], int n, int quantum)** {

sortByArrival(processes, n); // 按到达时间排序

int current\_time = 0, completed = 0;

int queue[n], front = 0, rear = 0; // 定义队列

bool in\_queue[n];

for (int i = 0; i < n; i++) in\_queue[i] = false; // 初始化队列状态

// 将第一个到达的进程加入队列

queue[rear++] = 0;

in\_queue[0] = true;

while (completed < n) {

// 如果队列为空，推进当前时间，并检查新到达的进程

if (front == rear) {

current\_time++;

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

if (!in\_queue[i] && processes[i].arrival <= current\_time) {

queue[rear++] = i; // 加入队列

in\_queue[i] = true;

}

}

continue;

}

// 从队列中取出当前进程

int idx = queue[front++];

if (front == n) front = 0; // 循环队列

// 如果当前进程是首次运行，设置开始时间

if (processes[idx].remaining == processes[idx].burst) {

processes[idx].start = current\_time;

}

// 执行当前进程的时间片

if (processes[idx].remaining <= quantum) {

current\_time += processes[idx].remaining;

processes[idx].remaining = 0; // 进程已完成

processes[idx].finish = current\_time; // 设置完成时间

processes[idx].completed = true; // 标记为已完成

completed++;

} else {

current\_time += quantum;

processes[idx].remaining -= quantum; // 减少剩余时间

}

//重点：默认先检查是否新到达的队列，再将当前刚刚没运行完的进程加入队列

// 检查新到达的进程，并加入队列

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

if (!in\_queue[i] && processes[i].arrival <= current\_time && processes[i].remaining > 0) {

queue[rear++] = i;

if (rear == n) rear = 0; // 循环队列

in\_queue[i] = true;

}

}

// 如果当前进程未完成，将其重新加入队列

if (processes[idx].remaining > 0) {

queue[rear++] = idx;

if (rear == n) rear = 0; // 循环队列

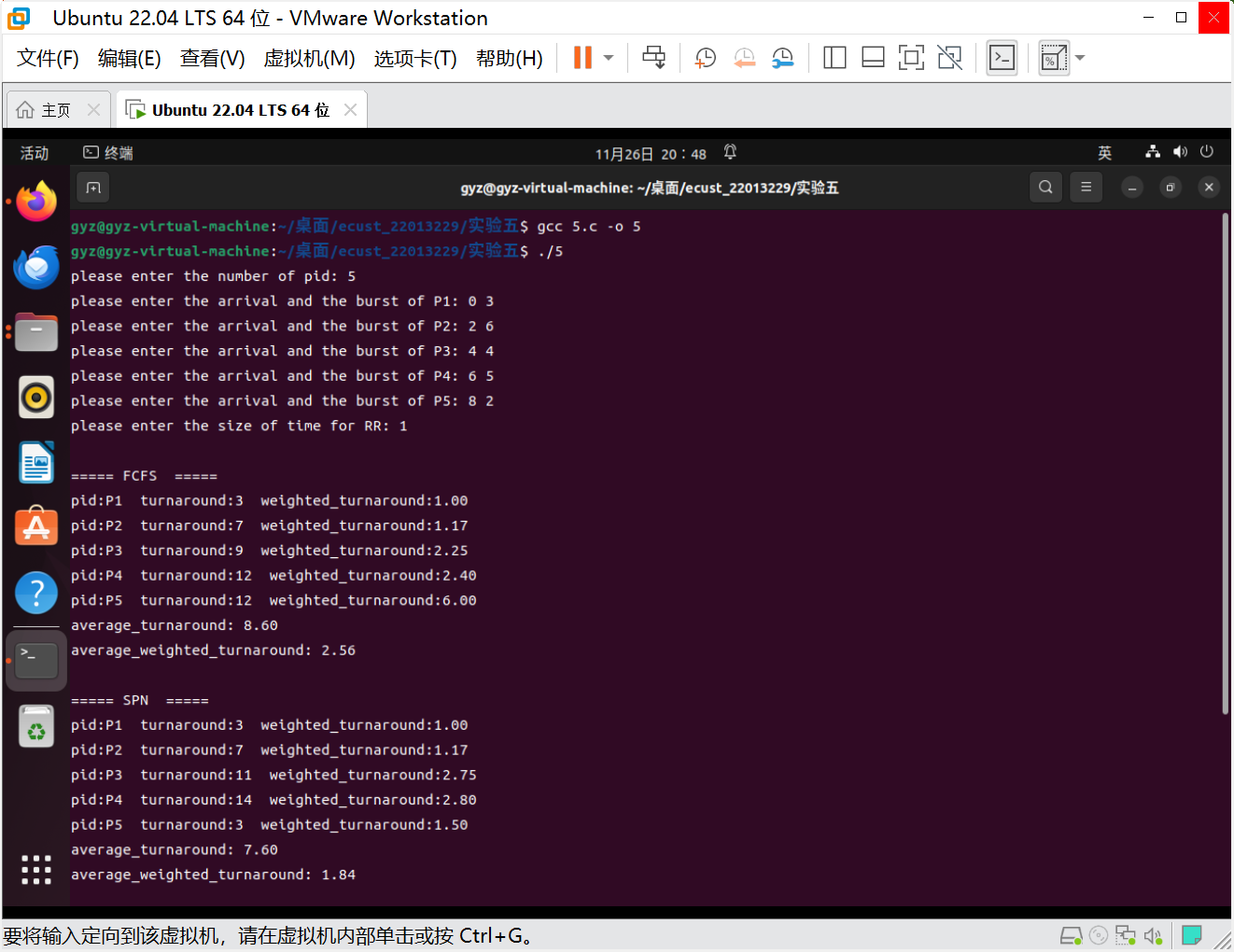
}

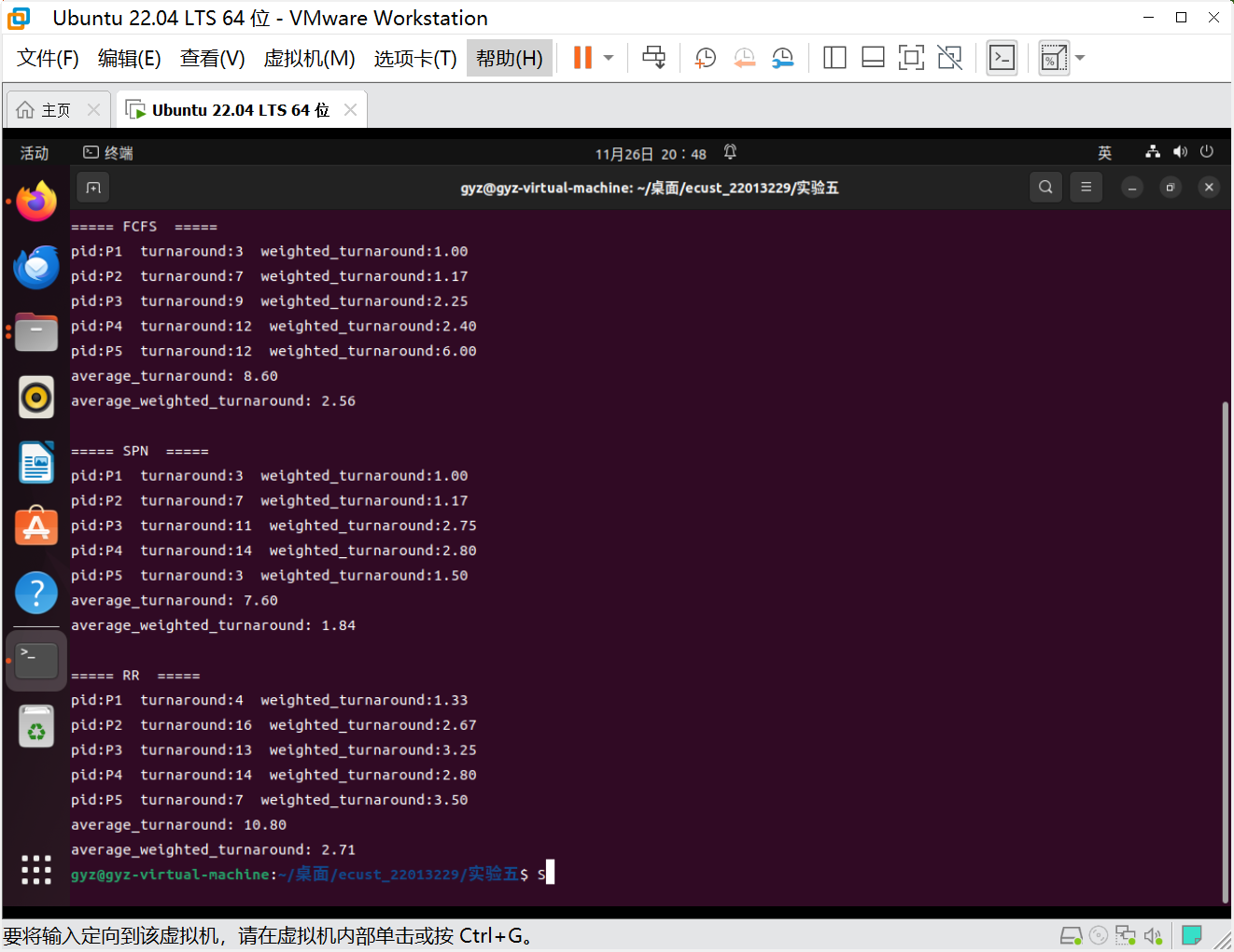
}

calculateAndPrintMetrics(processes, n); // 计算并输出指标

}

1. 计算每种算法下的平均周转时间、平均带权周转时间





# 四．思考题

1. 在实际的进程调度中，除了按调度算法选择下一个执行的进程外，还应该处理哪些工作？

①上下文切换：保存当前正在运行进程的状态，并恢复下一个被选中的进程的状态。上下文切换需要开销，包括保存/恢复寄存器、堆栈、程序计数器等。

②优先级调整：动态调整进程的优先级，例如通过老化机制防止某些低优先级进程长期被饿死。

③进程阻塞与唤醒：处理进程因 I/O 等事件而阻塞，以及当事件完成时的唤醒操作。

④资源管理与分配：确保进程调度与资源分配策略兼容，避免资源冲突和死锁。

⑤中断处理：响应硬件中断（如时钟中断）和软件中断（如系统调用），触发必要的调度。

⑥公平性和实时性保障：根据系统目标，确保不同进程获得公平的 CPU 使用机会或满足实时性需求。

1. 几种进程调度算法有何区别与联系？

（1）区别

①FCFS（先来先服务）：

按照进程到达的顺序调度。

实现简单，调度开销低。

缺点：可能导致“长进程占用”问题（即 convoy effect），导致系统响应时间不均衡。

②SPN（最短进程优先）：

优先调度预计运行时间最短的进程。

可以最小化平均周转时间。

缺点：不支持抢占，可能导致较长进程“饥饿”问题。

③RR（时间片轮转）：

通过为每个进程分配固定的时间片轮流调度。

支持抢占，适合交互式任务。

缺点：时间片过长会退化为 FCFS，时间片过短会增加上下文切换开销。

（2）联系

①都是用来管理 CPU 调度的基本算法。

②都涉及选择进程运行的顺序，并影响系统性能指标（如响应时间、吞吐量、等待时间等）。

③RR 可以看作对 FCFS 的改进，通过引入抢占和时间片提高交互性能。

④SPN 和 FCFS 的区别在于优先级定义方式不同，但都不支持抢占。

1. RR 调度算法中，如果将时间定量增长为一个任意大的数目，那么会产生什么影响？

①时间片无限大时，RR 退化为 FCFS：

若时间片无限大，当前进程在运行时不会被强制切换，直到其运行结束。

系统表现为先到先服务的调度策略（FCFS），长进程会占用 CPU 过久，导致短进程等待时间增加。

②影响系统响应时间：

系统响应时间变差，特别是对交互式任务（如用户输入）的响应显著延迟。

③减少上下文切换开销：

时间片增大导致上下文切换频率降低，从而减少切换开销，但以牺牲交互性能为代价。

④公平性下降：

时间片无限大时，CPU 时间的分配会显得不公平，可能导致某些进程的长时间等待，甚至产生饥饿。