**实验五 进程调度**

# 姓 名 吴梓弘 学 号 21013054 成绩

实验时间 12.7 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

（可另行加页打印）

FCFS:

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

typedef struct task{

char name[6];//名称

int roundtime;//到达时间

int servertime;//所需时间

int needtime;//在算法中不断更新的进程剩余所需时间

int leavetime;//进程退出时间-----

int count;//

int turnaround\_time;

int quanturnaround\_time;

char status;//状态

struct task \*next;

}PCB;

PCB \*head = NULL, \*tail = NULL;//全局的指向进程链表头部和尾部的指针

void addProcess(PCB \*p){

if(head==NULL){

head = p;

tail = p;

}else{

tail->next = p;

tail = p;

}

p->next = NULL;

}

PCB\* createProcess(char\* name, int roundTime, int serverTime) {

PCB \*newProcess = (PCB \*)malloc(sizeof(PCB));

strcpy(newProcess->name, name);

newProcess->roundtime = roundTime;

newProcess->servertime = serverTime;

newProcess->needtime = serverTime;

newProcess->count = 0;

newProcess->status = 'W';

return newProcess;

}

void fcfs(int n) {

int totaltime = 0;

int totalturnaroundtime = 0;

int totalquanturnaroundtime = 0;

while (head != NULL){

PCB \*current = head;

head = head->next;

if(current->roundtime>totaltime){

current->status = 'W';

addProcess(current);

totaltime++;

continue;

}

current->status = 'F';

totaltime = totaltime + current->servertime;

printf("\n进程名: %s\n", current->name);

printf("服务时间: %d\n", current->servertime);

printf("到达时间: %d\n", current->roundtime);

printf("状态: %c\n", current->status);

current->leavetime = totaltime;

current->turnaround\_time = current->leavetime - current->roundtime;

totalturnaroundtime += current->turnaround\_time;

current->quanturnaround\_time = (double)current->turnaround\_time / current->servertime;

totalquanturnaroundtime += current->quanturnaround\_time;

printf("进程 %s 已完成\n", current->name);

free(current);

}

printf("平均周转时间：%f\n", (double)totalturnaroundtime / (double)n);

printf("平均带权周转时间：%f\n", (double)totalquanturnaroundtime / (double)n);

}

int main() {

int n, i, roundTime, serverTime;

char name[6];

char a;

printf("输入进程个数: ");

scanf("%d", &n);

printf("输入进程: ");

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

scanf("%s %d %d", name, &roundTime, &serverTime);

addProcess(createProcess(name, roundTime, serverTime));

}

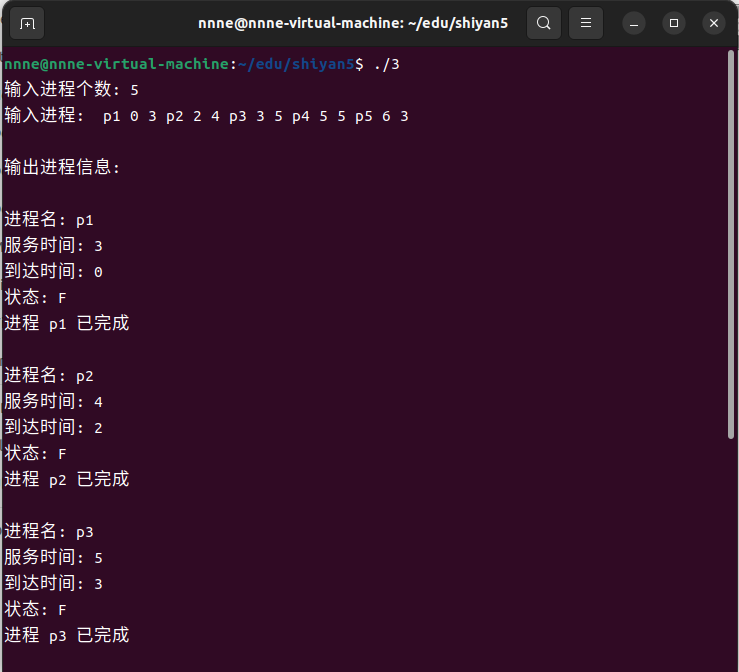
printf("\n输出进程信息:\n");

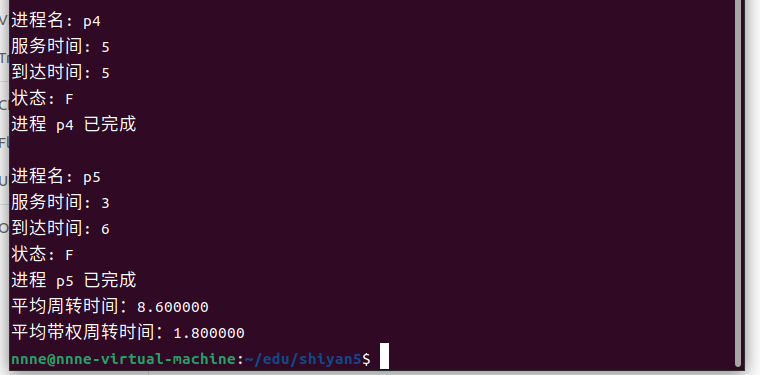
fcfs(n);

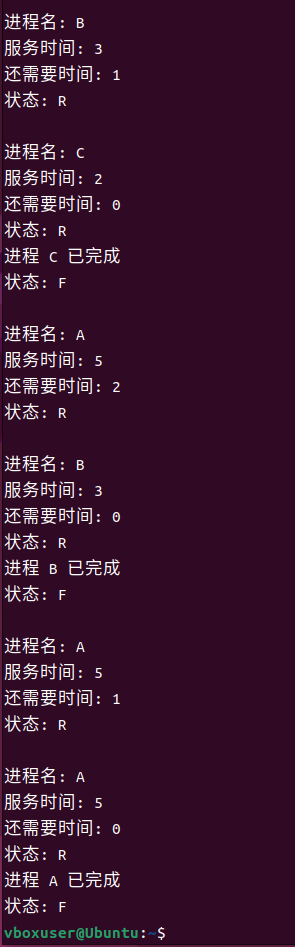
return 0;

}

运行结果（测试样例为五个进程，分别为：p1 0 3,p2 2 4,p3 3 5,p4 5 5）:







SPN核心代码：

void spn(int n) {

int totaltime = 0;

int totalturnaroundtime = 0;

int totalquanturnaroundtime = 0;

while (head != NULL) {

PCB \*current = head;

PCB \*shortest = NULL;

while (current != NULL) {

if (current->roundtime <= totaltime && (shortest == NULL || current->needtime < shortest->needtime)) {

shortest = current;

}

current = current->next;

}

if (shortest == NULL) {

totaltime++;

continue;

}

shortest->status = 'F';

totaltime += shortest->servertime;

printf("\n进程名: %s\n", shortest->name);

printf("服务时间: %d\n", shortest->servertime);

printf("到达时间: %d\n", shortest->roundtime);

printf("状态: %c\n", shortest->status);

shortest->leavetime = totaltime;

shortest->turnaround\_time = shortest->leavetime - shortest->roundtime;

totalturnaroundtime += shortest->turnaround\_time;

shortest->quanturnaround\_time = (double)shortest->turnaround\_time / shortest->servertime;

totalquanturnaroundtime += shortest->quanturnaround\_time;

printf("进程 %s 已完成\n", shortest->name);

if (shortest == head) {

head = head->next;

} else {

PCB \*prev = head;

while (prev->next != shortest) {

prev = prev->next;

}

prev->next = shortest->next;

}

free(shortest);

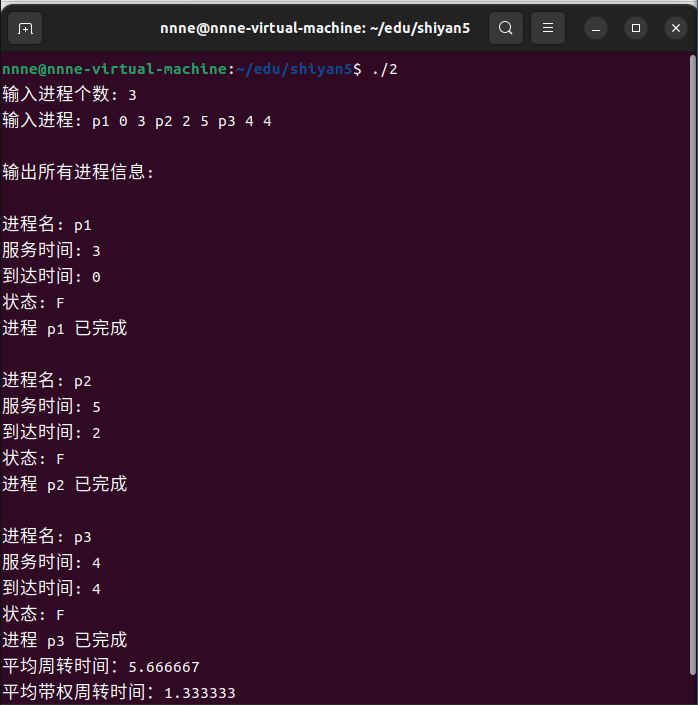
}

printf("平均周转时间：%f\n", (double)totalturnaroundtime / (double)n);

printf("平均带权周转时间：%f\n", (double)totalquanturnaroundtime / (double)n);

}

运行结果（测试样例为三个进程，分别为：p1 0 3,p2 2 5,p3 4 4）:



RR核心代码:

void rr(int n) {

int totaltime = 0;

int totalturnaroundtime = 0;

int totalquanturnaroundtime = 0;

while (head != NULL) {

PCB \*current = head;

head = head->next;

if (current->roundtime > totaltime) {

current->status = 'W';

addProcess(current);

continue;

}

current->status = 'R';

current->count += 1;

totaltime += 1;

current->needtime -= 1;

printf("\n进程名: %s\n", current->name);

printf("服务时间: %d\n", current->servertime);

printf("还需要时间: %d\n", current->needtime);

printf("已经经过的时间片: %d\n", current->count);

printf("到达时间: %d\n", current->roundtime);

printf("状态: %c\n", current->status);

if (current->needtime == 0) {

current->status = 'F';

current->leavetime = totaltime;

current->turnaround\_time = current->leavetime - current->roundtime;

totalturnaroundtime += current->turnaround\_time;

current->quanturnaround\_time = (double)current->turnaround\_time / current->servertime;

totalquanturnaroundtime += current->quanturnaround\_time;

printf("进程 %s 已完成\n", current->name);

free(current);

} else {

current->status = 'W';

addProcess(current);

}

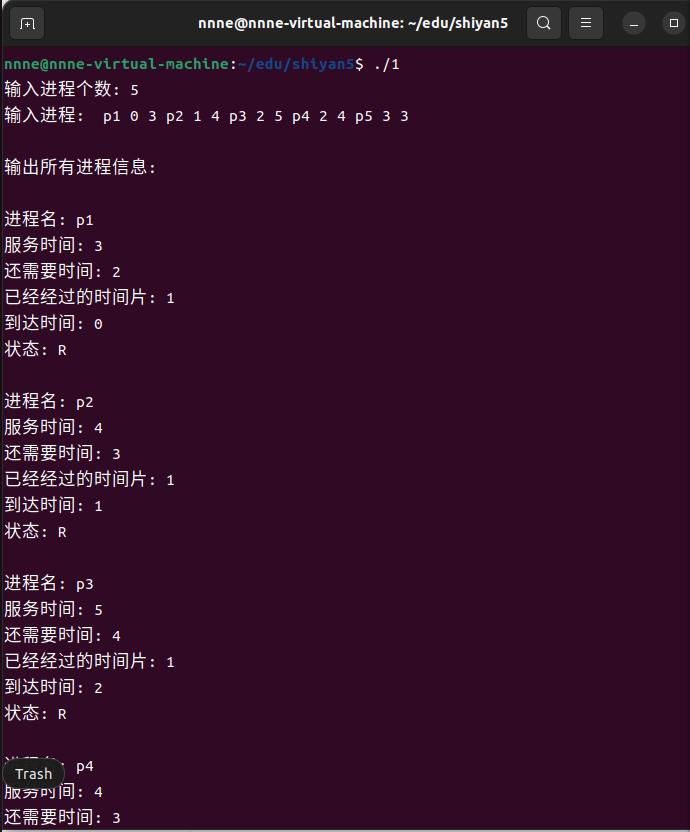
}

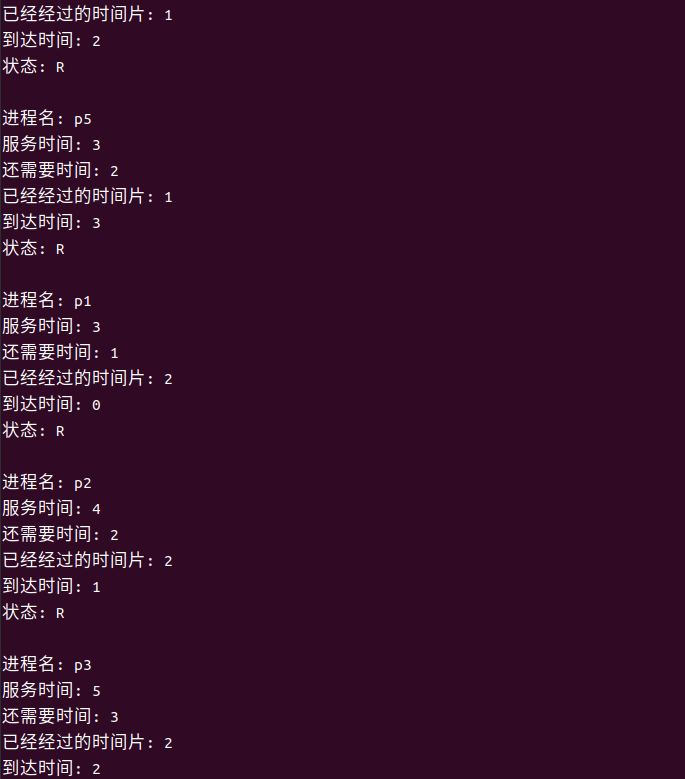
printf("平均周转时间：%f\n", (double)totalturnaroundtime / (double)n);

printf("平均带权周转时间：%f\n", (double)totalquanturnaroundtime / (double)n);

}

运行结果：（测试样例为两个进程，分别为p1 0 3,p2 1 4,p3 2 5,p4 2 4,p5 3 3）







# 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（Context Switch）： 切换到新的进程执行之前，需要保存当前正在执行进程的上下文信息（寄存器状态、程序计数器等），并加载即将执行的进程的上下文信息。这个过程称为上下文切换。上下文切换的开销对系统性能有影响，因此需要尽量减少上下文切换的次数

2.管理不同状态的进程队列，包括就绪队列、阻塞队列等。进程的状态会随着时间的推移而改变，需要将进程从一个队列移到另一个队列

3．处理硬件中断或软件中断

4．调度执行进程需要的资源，如处理器、内存等。

5.更新当前状况下的进程优先级

1. 几种进程调度算法有何区别与联系？

1、调度原则：

FCFS：按照进程到达的顺序进行调度，即先到达的进程先被执行。

SPN：选择服务时间最短的进程优先执行。

RR：将处理器分配给每个进程一个时间片，按照轮转的方式执行。

2、适用场景：

FCFS：适用于任务执行时间相对均匀，没有特别的优先级要求的场景。

SPN：适用于任务执行时间相差较大，需要尽量减小周转时间的场景。

RR：适用于任务执行时间差异较大，以及对于长任务不能等待时间过长的场景。

3、灵活性和开销：

FCFS：简单易实现，但可能不够灵活。

SPN：需要对所有就绪队列中的进程按照服务时间进行排序，可能增加调度开销。

RR：灵活，但需要处理时间片到期的情况，可能导致上下文切换增加。

4、优先级计算：

FCFS和RR： 通常没有显式的优先级计算。

SPN： 优先级是根据服务时间计算的，服务时间越短，优先级越高。

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？

当时间片大小设置得非常大时，进程在CPU上执行的时间变得很长。这可能导致其他就绪的进程长时间等待，造成整体系统的响应时间增加。

长任务需要等待更长时间才能获得CPU执行时间，这会导致对长任务的不公平性。

大的时间片会减少上下文切换的频率，减少上下文切换的开销

系统在灵活性上的下降，因为它无法快速地响应新的短任务。

大时间片可能导致处理器资源利用效率降低。进程可能会在等待更长的时间后才能再次获得CPU时间。