**实验一 处理机调度**

1. **实验目的**

多道程序设计中，经常是若干个进程同时处于就绪状态，为了使系统中的各进程有条不紊地运行，必须选择某种调度策略，以选择一个进程占用处理机。本次实验设计一个模拟单处理机调度的算法，以加深处理机调度的算法的理解。

1. **实验内容**

1. 按照轮转（时间片）算法设计模拟调度程序。

2. 输出进程的调度过程。

1. **源程序**

#include<stdio.h>

#include<string.h>

#include<iostream>

#include<malloc.h>

using namespace std;

#define slice\_time 10 //定义时间片的长度为10

//定义进程控制块PCB

struct pcb {

int id; //进程号

int status; //进程状态 0-Ready, 1-Run, 2-Finish

int arrive\_time; //进程到达时间

int time; //估计运行时间，服务时间

int run\_time; //已运行时间

int wait\_time; //等待时间

int priority; //优先级

struct pcb\* next; //链接指针

};

#define length sizeof(struct pcb)

int cur\_time = 0; //系统当前 运行 时间

int num = 0; //进程的个数

struct pcb \*ready\_head = NULL;

struct pcb \*pcb\_head = NULL;

struct pcb \*finish\_head = NULL;

/\*读文件数据,将文件中给出的进程放入PCB队列，以便后续使用

返回值： 0－失败 1－成功\*/

int readData() {

FILE \*fp;

char fname[200];

cout << "注意：文件中应包含以下信息：\n";

cout << "进程ID 到达时间 估计运行时间 优先级\n";

cout << "并且应按到达时间顺序排列!\n\n";

cout << "请输入进程流文件名:";

cin >> fname;

if ((fp = fopen(fname, "r")) == NULL) {

cout << "错误,文件打不开,请检查文件名" << endl;

return 0;

} else {

//建立PCB链表

struct pcb \*p1, \*p2;

pcb\_head = NULL;

p1 = p2 = (struct pcb\*)malloc(length);

fscanf(fp, "%d %d %d %d", &p1->id, &p1->arrive\_time, &p1->time, &p1->priority);

p1->status = 0; p1->run\_time = 0; p1->wait\_time = 0; //初始化

while (!feof(fp)) {

num++;

if (num == 1) {

pcb\_head = p1;

p1->next = NULL;

} else

p2->next = p1;

p2 = p1;

p1 = (struct pcb\*)malloc(length);

fscanf(fp, "%d %d %d %d", &p1->id, &p1->arrive\_time, &p1->time, &p1->priority);

p1->status = 0; p1->run\_time = 0; p1->wait\_time = 0;

}

p2->next = NULL;

free(p1);

fclose(fp);

return 1;

}

}

//建立就绪队列(前提：PCB队列是按到达时间排序的)

//返回值： 0－就绪队列空 1－就绪队列不空

int readyProcess() {

struct pcb \*p1, \*pready;

p1 = pcb\_head;

pready = ready\_head;

//将pready指向就绪队列最后一个结点

if (pready) {

while (pready->next) {

pready = pready->next;

}

}

/\*将新到达进程插入就绪队列末尾\*/

while (p1) {

if (p1->arrive\_time <= cur\_time) {

if (pready) {

pready->next = p1;

}

if (ready\_head == NULL) {

ready\_head = p1;

}

pready = p1;

p1 = p1->next;

pready->next = NULL;

} else {

pcb\_head = p1; //确定新的PCB队列头

if (ready\_head) {

return 1; //就绪队列中有元素

} else {

return 0;

}

}

}

if (p1 == NULL) //已经处理完PCB队列中的结点，将pcb队列头赋NULL

{

pcb\_head = NULL;

}

if (ready\_head) {

return 1; //就绪队列中有元素

} else {

return 0;

}

}

/\*轮转算法中时间片用完的进程插入就绪队列的末尾\*/

//参数：p-插入的进程

void insertReadyQueueByRR(struct pcb\* p) {

struct pcb \*pready;

readyProcess(); //若有新进程进入系统，先将其放入就绪队列

if (ready\_head == NULL) {

ready\_head = p;

return;

}

pready = ready\_head;

//将pready指向就绪队列最后一个结点

while (pready->next)

pready = pready->next;

pready->next = p;

p->next = NULL;

}

pcb\* getReadyProcessByFIFO() {

struct pcb \* p;

if (ready\_head) {

p = ready\_head;

ready\_head = ready\_head->next;

return p;

} else

return 0;

}

//时间片轮转算法

void runProcessByRR() {

struct pcb \*p, \*pFinish = NULL;

int t = 0, t\_sum = 0;//周转时间

float w = 0.0, w\_sum = 0.0;//带权周转时间

cout << endl << "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*" << endl;

cout << "RR算法执行流:" << endl;

cout << "进程名 等待时间 周转时间 带权周转时间" << endl;

ready\_head = NULL;

while (pcb\_head || ready\_head) {

while (1) {

if (readyProcess()) {

p = getReadyProcessByFIFO();

break; //!important

} else {

cur\_time++;

}

}

int past\_run\_time = p->run\_time;

while ((p->run\_time < p->time) && (p->run\_time - past\_run\_time) < slice\_time) {

p->status = 1;

p->run\_time++;

cur\_time++;

}

if (p->run\_time == p->time) {

p->status = 2;

t = cur\_time - p->arrive\_time;

t\_sum += t;

w = (float)t / p->time;

w\_sum += w;

p->wait\_time = t - p->time;

cout << p->id << "\t\t" << p->wait\_time << "\t\t" << t << "\t\t" << w << endl;

p = NULL;

past\_run\_time = NULL;

} else if ((p->run\_time - past\_run\_time) == slice\_time) {

p->status = 0;

insertReadyQueueByRR(p);

}

}

cout << "平均周转时间：" << (float)t\_sum / num << "\t\t" << "平均带权周转时间：" << (float)w\_sum / num << endl;

}

void main() {

if (readData()) {

runProcessByRR();

}

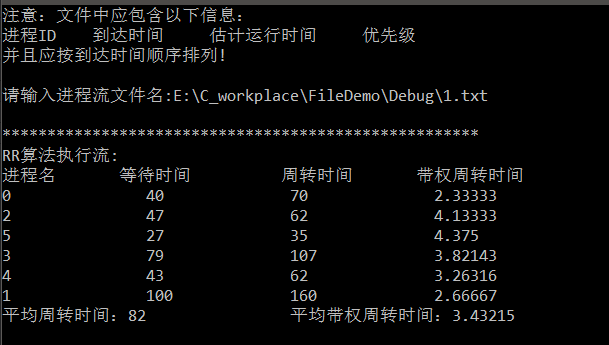
int a;

scanf("%d", &a);

}

1. **实验结果分析**

**（截取程序运行结果图并进行分析和说明）**

****

调度结果如图所示。经过课本例子验证结果正确，调度算法正确。

1. **实验总结**

**（通过实验，掌握和理解的知识；实验过程中获取的知识；实验中未能解决的问题等）**

在这次实验中，虽然代码中不是用多线程来实现的，但是事后我反而认为用多线程来写可能才是最优选，因为现实中可能无法提前预料有多少个进程将会到来，而且也无法提前预料进程将会在什么时候到来。在这个算法中，需要自增和时间有关的变量，容易出错，而在现实中时间本身就是个自增量，使用多线程来写就可以减少对时间因素的顾虑，把关注点主要放在链队的操作逻辑上就可以了。但考虑到该算法不是用于现实，而是主要用来模拟调度的过程，所以采用了这种带时间变量写法。困难就是如果不处理好条件，最后导致一层一层的嵌套代码，可读性实在差，可能上个 goto好一点。