实验四 处理器调度

# 实验目的

* 理解进程调度的过程。
* 掌握各种进程调度算法的实现方法

通过实验比较各种进程调度算法的优劣。

# 实验内容

随机给出一个进程调度实例，如：

进程 到达时间 服务时间

A 0 3

B 2 6

C 4 4

D 6 5

E 8 2

模拟进程调度，给出按照算法先来先服务FCFS、轮转RR（q=1）、最短进程优先SPN、最短剩余时间SRT、最高响应比优先HRRN进行调度各进程的完成时间、周转时间、响应比的值。

# 实验环境

* Windows 7 + Devcpp

# 遇到的问题及其解决方式

起初，SRT和HRRN中未考虑到达时间一致时的情况，没有按照要求进行排序。之后增加了判断，假如到达时间一致，再根据剩余时间/响应比进行排序。

# 源代码

## 先来先服务FCFS：

#include<stdio.h>

#include<stdlib.h>

#include<time.h>

typedef struct process\_FCFS{

char name;//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;//响应比

struct process\_FCFS \*link;//结构体指针

}FCFS;

FCFS \*p,\*q,\*head=NULL;

struct process\_FCFS a[100];

FCFS inital(struct process\_FCFS a[],int n);

void print(struct process\_FCFS a[],int n);

void Fcfs(struct process\_FCFS a[],int n);

struct process\_FCFS \*sortarrivetime(struct process\_FCFS a[],int n);

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

{

int i,j;

struct process\_FCFS t;

int flag;

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

{

flag=0;

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

{

if(a[j].arrivetime>a[j+1].arrivetime)

{

t=a[j];

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

a[j+1]=t;

flag=1;//交换

}

}

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;

}

void Fcfs(struct process\_FCFS a[],int n)

{

int i;

a[0].finishtime=a[0].arrivetime+a[0].servetime;

a[0].roundtime=a[0].finishtime-a[0].arrivetime;

a[0].daiquantime=a[0].roundtime/a[0].servetime;

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

{

if(a[i].arrivetime<a[i-1].finishtime)

{

a[i].finishtime=a[i-1].finishtime+a[i].servetime;

a[i].roundtime=a[i].finishtime-a[i].arrivetime;

a[i].daiquantime=a[i].roundtime/a[i].servetime;

}

else

{

a[i].finishtime=a[i].arrivetime+a[i].servetime;

a[i].roundtime=a[i].finishtime-a[i].arrivetime;

a[i].daiquantime=a[i].roundtime/a[i].servetime;

}

}

printf("先来先服务\n");

print(a,n);

}

void print(struct process\_FCFS a[],int n)

{

int i;

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

{

printf("进程名：%c ",a[i].name);

printf("到达时间：%f ",a[i].arrivetime);

printf("服务时间：%f ",a[i].servetime);

printf("完成时间：%f ",a[i].finishtime);

printf("周转时间：%f ",a[i].roundtime);

printf("响应比：%f ",a[i].daiquantime);

printf("\n");

}

}

//主函数

int main()

{

int n,i;

n=10;//进程数

srand(time(0));

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

{

a[i].name=65+i;

a[i].arrivetime=((float)(rand()%100)/10);

a[i].servetime=((float)(rand()%100)/10+rand()%10);

}

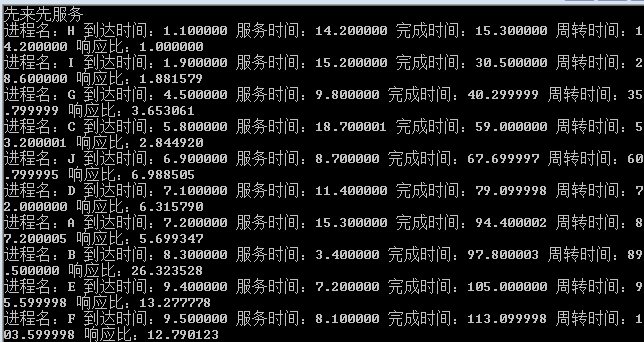
\*sortarrivetime(a,n);

Fcfs(a,n);

for(;;);

return 0;

}



## 轮转法（RR）：

#include<stdio.h>

#include<stdlib.h>

#include<time.h>

#include<malloc.h>

typedef int QElemType;

int MaxNume=100;

typedef struct QNode{

QElemType data;

struct QNode \*next;

}QNode,\*QueuePtr;

typedef struct{

QueuePtr front;

QueuePtr rear;

}LinkQueue;

LinkQueue Queue;

typedef struct process\_FCFS{

char name;//进程名

float arriveTIME;//到达时间

float serveTIME;//服务时间

float needTIME;//剩下需要的服务时间

float finishTIME;//完成时间

float roundTIME;//周转时间

float daiquanTIME;//带权周转时间

struct process\_FCFS \*link;//结构体指针

}FCFS;

FCFS \*p,\*q,\*head=NULL;

struct process\_FCFS a[100];

float pian=1;

float TIME;

int n;//进程个数

bool flags=true;//标志小于时间片的时间片有无用过

FCFS inital(struct process\_FCFS a[],int n);

void print(struct process\_FCFS a[],int n);

void Fcfs(struct process\_FCFS a[],int n);

struct process\_FCFS \*sortarriveTIME(struct process\_FCFS a[],int n);

bool finished(struct process\_FCFS a[],int n);

int InitQueue(LinkQueue &Q);

int DestroyQueue(LinkQueue &Q);

int EnQueue(LinkQueue &Q,QElemType e);

int RollQueue(LinkQueue &Q);

int getNextProcess(struct process\_FCFS a[],int n,float dTime);

int killPian(int t,float TIMERemaining,bool flag);

struct process\_FCFS \*sortRR(struct process\_FCFS a[],int n);

int InitQueue(LinkQueue &Q){

Q.front=Q.rear=(QueuePtr)malloc(sizeof(QNode));

Q.front->next=NULL;

return 1;

}

int DestroyQueue(LinkQueue &Q){

while(Q.front){

Q.rear=Q.front->next;

free(Q.front);

Q.front=Q.rear;

}

return 1;

}

int EnQueue(LinkQueue &Q,QElemType e){ //增加一个节点到队尾

QueuePtr p=(QueuePtr)malloc(sizeof(QNode));

if(Q.front ==Q.rear )

Q.front ->next =p;

p->data=e;

p->next=NULL;

Q.rear->next=p;

Q.rear=p;

return 1;

}

int RollQueue(LinkQueue &Q){

QueuePtr p;

QElemType e;

if(Q.front ==Q.rear )

return 0;//队列已空

p=Q.front->next ;

e=p->data ;

if(a[e].needTIME >0){

if(p==Q.rear )

return 1;

else{

Q.front->next =p->next ;

Q.rear ->next =p;

p->next =NULL;

Q.rear =p;

return 1;

}

}

else if(a[e].needTIME ==0){

if(p==Q.rear ){

Q.front =Q.rear ;

return 0;

}

else{

Q.front->next =p->next ;

free(p);

return 1;

}

}

}

//按到达时间进行冒泡排序

struct process\_FCFS \*sortarriveTIME(struct process\_FCFS a[],int n)

{

int i,j;

struct process\_FCFS t;

int flag;

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

{

flag=0;

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

{

if(a[j].arriveTIME>a[j+1].arriveTIME)

{

t=a[j];

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

a[j+1]=t;

flag=1;//交换

}

}

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;

}

bool finished(struct process\_FCFS a[],int n){

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

if(a[k].needTIME>0)

return false;

}

return true;

}

int killPian(int t,float TIMERemaining,bool flag){

if(pian<1&&pian>0){

flags=true;

}

if(a[t].needTIME>TIMERemaining){

a[t].needTIME=a[t].needTIME-TIMERemaining;

TIME=TIME+TIMERemaining;

if(flags)

pian=1;

if(flag)

t=getNextProcess(a,n,TIMERemaining);

return t;

}

else if(a[t].needTIME==TIMERemaining){

TIME=TIME+TIMERemaining;

a[t].needTIME=0;

a[t].finishTIME=TIME;

a[t].roundTIME=TIME-a[t].arriveTIME;

a[t].daiquanTIME=a[t].roundTIME/a[t].serveTIME;

if(flags)

pian=1;

if(flag)

t=getNextProcess(a,n,TIMERemaining);

return t;

}

else if(a[t].needTIME<TIMERemaining){

a[t].finishTIME=TIME+a[t].needTIME;

a[t].roundTIME=a[t].finishTIME-a[t].arriveTIME;

a[t].daiquanTIME=a[t].roundTIME/a[t].serveTIME;

TIMERemaining=TIMERemaining-a[t].needTIME;

TIME=TIME+a[t].needTIME;

t=getNextProcess(a,n,TIMERemaining+a[t].needTIME);

if(flags)

pian=1;

if(t==-1)

return -1;

else{

killPian(t,TIMERemaining,false);

}

}

}

int getNextProcess(struct process\_FCFS a[],int n,float dTime){ //返回下一个运行的进程号

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

if(a[l].arriveTIME>TIME-dTime&&a[l].arriveTIME<=TIME)

EnQueue(Queue,l);

}

int m=RollQueue(Queue);

if(m==0&&!finished(a,n)){ //越过长时间空白

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

if(a[r].arriveTIME>TIME){

for(int kk=0;kk>=0;kk++){

if(TIME+kk\*pian<=a[r].arriveTIME&&TIME+(kk+1)\*pian>a[r].arriveTIME){

break;

}

pian=pian-(a[r].arriveTIME-TIME-kk\*pian);

}

// pian=pian-(a[r].arriveTIME-TIME-kk\*pian);

TIME=a[r].arriveTIME;

EnQueue(Queue,r);

flags=false;

}

}

}

else if(m==0&&finished(a,n)){

return -1; //无剩余的需要执行的进程

}

return Queue.front->next->data;

}

struct process\_FCFS \*sortRR(struct process\_FCFS a[],int n){

TIME=a[0].arriveTIME;

int q=0;

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

if(a[pp].arriveTIME==a[0].arriveTIME ){

EnQueue(Queue,pp);

}

}

while(q!=-1){

q=killPian(q,pian,true);

}

return a;

}

void print(struct process\_FCFS a[],int n)

{

int i;

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

{

printf("here");

printf("进程名：%c",a[i].name);

printf("到达时间：%f",a[i].arriveTIME);

printf("服务时间：%f",a[i].serveTIME);

printf("完成时间：%f",a[i].finishTIME);

printf("周转时间：%f",a[i].roundTIME);

printf("带权周转时间：%f",a[i].daiquanTIME);

printf("\n");

}

}

//主函数

int main()

{

InitQueue(Queue);

n=5;//进程数

srand(time(0));

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

{

printf("print %d process name:",i+1);

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

printf("arrivetime");

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

printf("servetime");

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

a[i].needTIME=a[i].serveTIME ;

}

\*sortarriveTIME(a,n);

\*sortRR(a,n);

printf("时间片轮转算法\n");

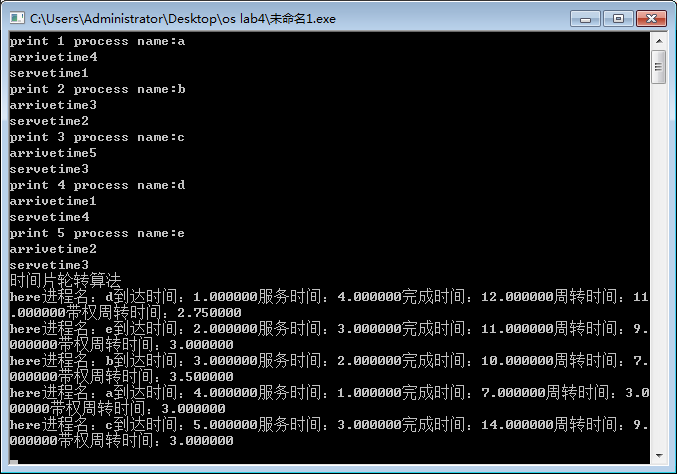
print(a,n);

DestroyQueue(Queue);

for(;;) ;

return 0;

}



## 最短时间优先（SPN）：

#include<iostream>

using namespace std;

#define MAX\_NUM 5 //进程数量

//定义进程控制块的结构

typedef struct pcb\_struct

{

char name[2]; //进程名称

char ProcState; //进程状态

int ArriveTime; //进程到达时间

int StartTime; //进程开始时间

int FinishTime; //进程结束时间

int ServeTime; //进程服务时间

int RunTime; //已经占用cpu时间

int MoreTime; //进程还需要运行时间

int TurnaroundTime; //进程周转时间

struct pcb\_struct \*next; //结构体指针

} pcb;

int time=0; //cpu时间

pcb\* head=NULL; //链表头指针

//pcb\* tail=NULL; //链表尾指针

//进程初始化，输入各进程的名称，到达时间，服务时间，

void FCFS\_initial()

{

cout<<"请输入"<<MAX\_NUM<<"个进程的进程名、达到时间、服务时间" <<endl;

for(int i=1;i<=MAX\_NUM;i++)

{

pcb\* new\_proc=NULL;

if((new\_proc=new pcb)==NULL)

{

cout<<"创建新进程失败"<<endl;

return;

}

cout<<"第"<<i<<"个进程:"<<endl;

cin>>new\_proc->name;

cin>>new\_proc->ArriveTime;

cin>>new\_proc->ServeTime;

new\_proc->StartTime=0; //开始时间置0

new\_proc->FinishTime=0; //结束时间置0

new\_proc->ProcState= 'w'; //进程状态置为等待

new\_proc->RunTime=0; //已经占用cpu时间置0

new\_proc->MoreTime=0; //进程还需要运行时间置0

new\_proc->TurnaroundTime=0; //进程的周转时间置为0

new\_proc->next=NULL; //新进程尾指针悬空

if(head==NULL) //输入第一个进程时

{

head=new\_proc;//头指针指向第一个进程

// tail=head;

}

else

{

if(head->ArriveTime>new\_proc->ArriveTime) //新进程的到达时间小于第一个进程的达到时间，置于第一个节点之前

{

new\_proc->next=head;

head=new\_proc;

}

else

{

pcb \*iterator\_proc=head; //用来遍历寻找的指针

while(iterator\_proc->next!=NULL&&iterator\_proc->next->ArriveTime<new\_proc->ArriveTime) //遍历链表，找到新进程的放置位置

{

iterator\_proc=iterator\_proc->next;

}

/\* if(iterator\_proc->next==NULL) //如果遍历到尾节点

{

tail=new\_proc; //改变尾指针

}\*/

new\_proc->next=iterator\_proc->next;

iterator\_proc->next=new\_proc;

}

}

}

}

pcb \*ArriveTime\_SortList(pcb \*head) //对链表中的进程按照到达时间进行排序

{

//链表快速排序

if(head == NULL || head->next == NULL) return head;

pcb \*p = NULL;

bool isChange = true;

while(p != head->next && isChange)

{

pcb\* q = head;

isChange = false;//标志当前这一轮中又没有发生元素交换，如果没有则表示数组已经有序

for(; q->next && q->next != p; q = q->next)

{

if(q->ArriveTime > q->next->ArriveTime)

{

//交换了节点名、到达时间、服务时间，相当于交换了两个节点本身

swap(q->name, q->next->name);

swap(q->ArriveTime, q->next->ArriveTime);

swap(q->ServeTime, q->next->ServeTime);

isChange = true;

}

}

p = q;

}

return head;

}

pcb \*SPN\_SortList(pcb \*head) //对链表中的进程按照服务时间进行排序

{

//链表快速排序

if(head == NULL || head->next == NULL) return head;

pcb \*p = NULL;

bool isChange = true;

while(p != head->next && isChange)

{

pcb\* q = head;

isChange = false;//标志当前这一轮中又没有发生元素交换，如果没有则表示数组已经有序

for(; q->next && q->next != p; q = q->next)

{

if(q->ServeTime > q->next->ServeTime)

{

//交换了节点名、到达时间、服务时间，相当于交换了两个节点本身

swap(q->name, q->next->name);

swap(q->ArriveTime, q->next->ArriveTime);

swap(q->ServeTime, q->next->ServeTime);

isChange = true;

}

}

p = q;

}

return head;

}

void SPN()

{

cout<<"最短进程优先:"<<endl;

SPN\_SortList(head); //对链表中的元素按照服务时间从小到大排序

pcb\* run\_proc=head; //遍历运行进程的指针

time=run\_proc->ArriveTime;

while(run\_proc)

{

//如果当前时刻短作业还没到达，cpu需要等待

run\_proc->ArriveTime>time?

(run\_proc->StartTime=run\_proc->ArriveTime,time=run\_proc->ArriveTime): run\_proc->StartTime=time;

cout<<endl;

cout<<run\_proc->StartTime<<"时刻，"<<run\_proc->name<<"进程正在执行"<<endl;

time+=run\_proc->ServeTime;

run\_proc->FinishTime=time;

run\_proc->TurnaroundTime=run\_proc->FinishTime-run\_proc->ArriveTime;//周转时间=进程完成时间-进程到达时间

cout<<run\_proc->name<<"完成时间:"<<run\_proc->FinishTime<<" "<<"周转时间："<<run\_proc->TurnaroundTime<<" "

<<"响应比："<< (1+(run\_proc->StartTime-run\_proc->ArriveTime)/run\_proc->ServeTime)<<endl;

cout<<endl;

run\_proc=run\_proc->next;

}

cout<<endl;

}

int main()

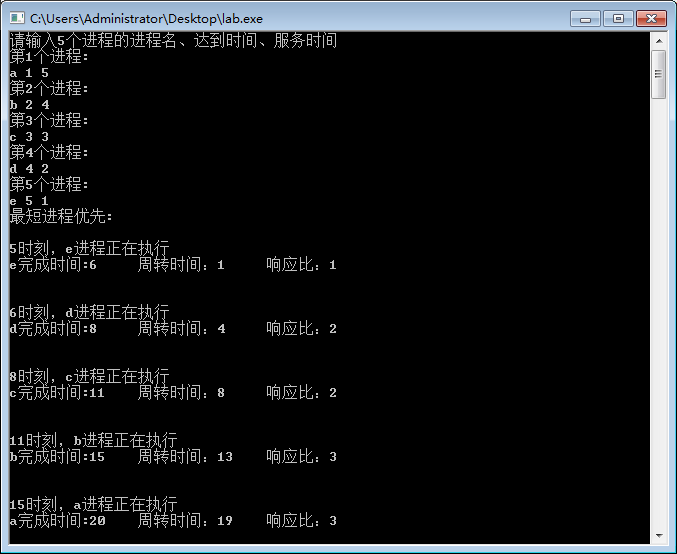
{

FCFS\_initial();

SPN();

return 0;

}



## 最短剩余时间(SRT)

#include<iostream>

using namespace std;

#define MAX\_NUM 3 //进程数量

#define INF 100000 //无穷大

//定义进程控制块的结构

typedef struct pcb\_struct

{

char name[2]; //进程名称

char ProcState; //进程状态

int ArriveTime; //进程到达时间

int StartTime; //进程开始时间

int FinishTime; //进程结束时间

int ServeTime; //进程服务时间

int RunTime; //已经占用cpu时间

int MoreTime; //进程还需要运行时间

int TurnaroundTime; //进程周转时间

float ResponseRatio; //响应比

struct pcb\_struct \*next; //结构体指针

} pcb;

int time=0; //cpu时间

pcb\* head=NULL; //链表头指针

//pcb\* tail=NULL; //链表尾指针

//进程初始化，输入各进程的名称，到达时间，服务时间，

void FCFS\_initial()

{

cout<<"请输入"<<MAX\_NUM<<"个进程的进程名、达到时间、服务时间" <<endl;

for(int i=1;i<=MAX\_NUM;i++)

{

pcb\* new\_proc=NULL;

if((new\_proc=new pcb)==NULL)

{

cout<<"创建新进程失败"<<endl;

return;

}

cout<<"第"<<i<<"个进程:"<<endl;

cin>>new\_proc->name;

cin>>new\_proc->ArriveTime;

cin>>new\_proc->ServeTime;

new\_proc->StartTime=0; //开始时间置0

new\_proc->FinishTime=0; //结束时间置0

new\_proc->ProcState= 'w'; //进程状态置为等待

new\_proc->RunTime=0; //已经占用cpu时间置0

new\_proc->MoreTime=0; //进程还需要运行时间置0

new\_proc->TurnaroundTime=0; //进程的周转时间置为0

new\_proc->next=NULL; //新进程尾指针悬空

if(head==NULL) //输入第一个进程时

{

head=new\_proc;//头指针指向第一个进程

// tail=head;

}

else

{

if(head->ArriveTime>new\_proc->ArriveTime) //新进程的到达时间小于第一个进程的达到时间，置于第一个节点之前

{

new\_proc->next=head;

head=new\_proc;

}

else

{

pcb \*iterator\_proc=head; //用来遍历寻找的指针

while(iterator\_proc->next!=NULL&&iterator\_proc->next->ArriveTime<new\_proc->ArriveTime) //遍历链表，找到新进程的放置位置

{

iterator\_proc=iterator\_proc->next;

}

/\* if(iterator\_proc->next==NULL) //如果遍历到尾节点

{

tail=new\_proc; //改变尾指针

}

\*/

new\_proc->next=iterator\_proc->next;

iterator\_proc->next=new\_proc;

}

}

}

}

pcb \*ArriveTime\_SortList(pcb \*head) //对链表中的进程按照到达时间进行排序

{

//链表快速排序

if(head == NULL || head->next == NULL) return head;

pcb \*p = NULL;

bool isChange = true;

while(p != head->next && isChange)

{

pcb\* q = head;

isChange = false;//标志当前这一轮中又没有发生元素交换，如果没有则表示数组已经有序

for(; q->next && q->next != p; q = q->next)

{

if(q->ArriveTime > q->next->ArriveTime)

{

//交换了节点名、到达时间、服务时间，相当于交换了两个节点本身

swap(q->name, q->next->name);

swap(q->ArriveTime, q->next->ArriveTime);

swap(q->ServeTime, q->next->ServeTime);

isChange = true;

}

}

p = q;

}

return head;

}

pcb \*MoreTime\_SortList(pcb \*head) //对链表中的进程按照剩余运行时间进行排序

{

//链表快速排序

if(head == NULL || head->next == NULL) return head;

pcb \*p = NULL;

bool isChange = true;

while(p != head->next && isChange)

{

pcb\* q = head;

isChange = false;//标志当前这一轮中又没有发生元素交换，如果没有则表示数组已经有序

for(; q->next && q->next != p; q = q->next)

{

if(q->MoreTime > q->next->MoreTime)

{

swap(q->name, q->next->name);

swap(q->ArriveTime, q->next->ArriveTime);

swap(q->ServeTime, q->next->ServeTime);

swap(q->MoreTime, q->next->MoreTime);

swap(q->ProcState, q->next->ProcState);

swap(q->RunTime, q->next->RunTime);

swap(q->ResponseRatio, q->next->ResponseRatio);

isChange = true;

}

}

p = q;

}

return head;

}

void SRT()

{

cout<<"SRT:"<<endl;

ArriveTime\_SortList(head); //对链表中的元素按照到达时间从小到大排序

time=0; //cpu时间置0

pcb\* run\_proc=head; //遍历运行进程的指针

pcb\* new\_proc=head; //遍历修改PCB的状态的指针

for(int i=1;i<=MAX\_NUM;i++)

{

new\_proc->StartTime=0; //开始时间置0

new\_proc->FinishTime=0; //结束时间置0

new\_proc->ProcState= 'w'; //进程状态置为等待

new\_proc->RunTime=0; //已经占用cpu时间置0

new\_proc->MoreTime=new\_proc->ServeTime; //进程还需要运行时间初始值与服务时间相等

new\_proc->TurnaroundTime=0; //进程的周转时间置为0

new\_proc=new\_proc->next;

}

while(run\_proc->MoreTime != INF)

{

if(run\_proc->ProcState != 'f')

{

//如果当前最短剩余时间进程还没到达，cpu需要等待

run\_proc->ArriveTime>time?

(run\_proc->StartTime=run\_proc->ArriveTime,time=run\_proc->ArriveTime): run\_proc->StartTime=time;

cout<<time<<"时刻 ，"<<run\_proc->name<<"正在执行"<<endl;

// run\_proc->StartTime=time; //修改开始运行时间

//思路是每运行一个时间刻度，进行一次检查

run\_proc->RunTime++;

time++;

run\_proc->MoreTime=run\_proc->ServeTime-run\_proc->RunTime;

if(run\_proc->MoreTime <= 0)

{

cout<<time<<"时刻，"<<run\_proc->name<<"运行结束"<<endl;

run\_proc->FinishTime=time;

run\_proc->TurnaroundTime=run\_proc->FinishTime-run\_proc->ArriveTime;//周转时间=进程完成时间-进程到达时间

cout<<run\_proc->name<<"完成时间:"<<run\_proc->FinishTime<<" "<<"周转时间："<<run\_proc->TurnaroundTime<<" "

<<"响应比："<< (1+(run\_proc->StartTime-run\_proc->ArriveTime)/run\_proc->ServeTime)<<endl;

run\_proc->ProcState = 'f' ; //状态修改

run\_proc->MoreTime= INF ;//剩余运行时间置为无穷大

}

else

{

cout<<time<<"时刻 ，"<<run\_proc->name<<" 决策运行 " <<endl;

cout<<"已经运行"<<run\_proc->RunTime<<"还需要执行"<<run\_proc->MoreTime<<endl;

}

MoreTime\_SortList(head); //对剩余运行时间进行排序

run\_proc=head;

for( ; run\_proc->MoreTime!=INF ; )

{

if(run\_proc->ArriveTime>time && run\_proc->next->MoreTime!=INF) //当前进程还未到达 && 下一个进程还未运行结束

run\_proc=run\_proc->next; //则搜索下一个进程

else

break; //当前进程是整个任务线中还未运行完的最后一个进程 ||当前进程是可行进程中最短运行时间的

}

}

}

}

int main()

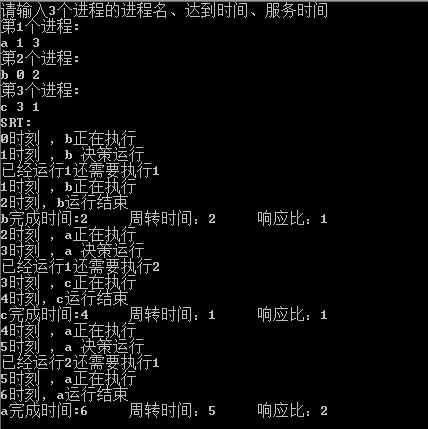
{

FCFS\_initial();

SRT();

return 0;

}



## 最高响应比优先HRRN

#include<iostream>

using namespace std;

#define MAX\_NUM 3 //进程数量

#define INF 100000 //无穷大

//定义进程控制块的结构

typedef struct pcb\_struct

{

char name[2]; //进程名称

char ProcState; //进程状态

int ArriveTime; //进程到达时间

int StartTime; //进程开始时间

int FinishTime; //进程结束时间

int ServeTime; //进程服务时间

int RunTime; //已经占用cpu时间

int MoreTime; //进程还需要运行时间

int TurnaroundTime; //进程周转时间

float ResponseRatio; //响应比

struct pcb\_struct \*next; //结构体指针

} pcb;

int time=0; //cpu时间

pcb\* head=NULL; //链表头指针

//pcb\* tail=NULL; //链表尾指针

//进程初始化，输入各进程的名称，到达时间，服务时间，

void FCFS\_initial()

{

cout<<"请输入"<<MAX\_NUM<<"个进程的进程名、达到时间、服务时间" <<endl;

for(int i=1;i<=MAX\_NUM;i++)

{

pcb\* new\_proc=NULL;

if((new\_proc=new pcb)==NULL)

{

cout<<"创建新进程失败"<<endl;

return;

}

cout<<"第"<<i<<"个进程:"<<endl;

cin>>new\_proc->name;

cin>>new\_proc->ArriveTime;

cin>>new\_proc->ServeTime;

new\_proc->StartTime=0; //开始时间置0

new\_proc->FinishTime=0; //结束时间置0

new\_proc->ProcState= 'w'; //进程状态置为等待

new\_proc->RunTime=0; //已经占用cpu时间置0

new\_proc->MoreTime=0; //进程还需要运行时间置0

new\_proc->TurnaroundTime=0; //进程的周转时间置为0

new\_proc->next=NULL; //新进程尾指针悬空

if(head==NULL) //输入第一个进程时

{

head=new\_proc;//头指针指向第一个进程

// tail=head;

}

else

{

if(head->ArriveTime>new\_proc->ArriveTime) //新进程的到达时间小于第一个进程的达到时间，置于第一个节点之前

{

new\_proc->next=head;

head=new\_proc;

}

else

{

pcb \*iterator\_proc=head; //用来遍历寻找的指针

while(iterator\_proc->next!=NULL&&iterator\_proc->next->ArriveTime<new\_proc->ArriveTime) //遍历链表，找到新进程的放置位置

{

iterator\_proc=iterator\_proc->next;

}

/\* if(iterator\_proc->next==NULL) //如果遍历到尾节点

{

tail=new\_proc; //改变尾指针

}

\*/

new\_proc->next=iterator\_proc->next;

iterator\_proc->next=new\_proc;

}

}

}

}

pcb \*ArriveTime\_SortList(pcb \*head) //对链表中的进程按照到达时间进行排序

{

//链表快速排序

if(head == NULL || head->next == NULL) return head;

pcb \*p = NULL;

bool isChange = true;

while(p != head->next && isChange)

{

pcb\* q = head;

isChange = false;//标志当前这一轮中又没有发生元素交换，如果没有则表示数组已经有序

for(; q->next && q->next != p; q = q->next)

{

if(q->ArriveTime > q->next->ArriveTime)

{

//交换了节点名、到达时间、服务时间，相当于交换了两个节点本身

swap(q->name, q->next->name);

swap(q->ArriveTime, q->next->ArriveTime);

swap(q->ServeTime, q->next->ServeTime);

isChange = true;

}

}

p = q;

}

return head;

}

pcb \*HRRN\_SortList(pcb \*head) //对链表中的进程按照响应比从大到小排序

{

//链表快速排序

if(head == NULL || head->next == NULL) return head;

//计算响应比

pcb\* iterator\_proc=head;

for(int i=1;i<=MAX\_NUM;i++)

{

if(iterator\_proc->ArriveTime<=time&&iterator\_proc->ProcState!='f') //进程此刻已经到来，并且还未运行结束

iterator\_proc->ResponseRatio=(1.0+(time-iterator\_proc->ArriveTime)/iterator\_proc->ServeTime);

else

iterator\_proc->ResponseRatio=-1;

iterator\_proc=iterator\_proc->next;

}

pcb \*p = NULL;

bool isChange = true;

while(p != head->next && isChange)

{

pcb\* q = head;

isChange = false;//标志当前这一轮中又没有发生元素交换，如果没有则表示数组已经有序

for(; q->next && q->next != p; q = q->next)

{

if(q->ResponseRatio < q->next->ResponseRatio)

{

swap(q->name, q->next->name);

swap(q->ArriveTime, q->next->ArriveTime);

swap(q->ServeTime, q->next->ServeTime);

swap(q->MoreTime, q->next->MoreTime);

swap(q->ProcState, q->next->ProcState);

swap(q->RunTime, q->next->RunTime);

swap(q->ResponseRatio, q->next->ResponseRatio);

isChange = true;

}

}

p = q;

}

iterator\_proc=head;

for(int i=1;i<=MAX\_NUM;i++)

{

iterator\_proc=iterator\_proc->next;

}

return head;

}

void HRRN()

{

cout<<"HRRN:"<<endl;

ArriveTime\_SortList(head); //对链表中的元素按照到达时间（此时等于剩余时间）从小到大排序

time=0; //cpu时间置0

pcb\* run\_proc=head; //遍历运行进程的指针

pcb\* new\_proc=head; //遍历修改PCB的状态的指针

int process\_count=0;

for(int i=1;i<=MAX\_NUM;i++)

{

new\_proc->StartTime=0; //开始时间置0

new\_proc->FinishTime=0; //结束时间置0

new\_proc->RunTime=0; //已经占用cpu时间置0

new\_proc->ProcState='w'; //进程状态置为等待

new\_proc->MoreTime=new\_proc->ServeTime; //进程还需要运行时间初始值与服务时间相等

new\_proc->TurnaroundTime=0; //进程的周转时间置为0

new\_proc->ResponseRatio=-1.0; //响应比设置为-1

new\_proc=new\_proc->next;

}

while(process\_count < MAX\_NUM)

{

cout<<time<<"时刻 ，"<<run\_proc->name<<"进程正在执行"<<endl;

run\_proc->StartTime=time; //修改开始运行时间

//思路是每运行一个时间刻度，进行一次检查

run\_proc->RunTime++;

time++;

run\_proc->MoreTime=run\_proc->ServeTime-run\_proc->RunTime;

if(run\_proc->MoreTime <= 0)

{

cout<<time<<"时刻，"<<run\_proc->name<<"进程运行结束"<<endl;

run\_proc->FinishTime=time;

run\_proc->TurnaroundTime=run\_proc->FinishTime-run\_proc->ArriveTime;//周转时间=进程完成时间-进程到达时间

cout<<run\_proc->name<<"完成时间:"<<run\_proc->FinishTime<<" "<<"周转时间："<<run\_proc->TurnaroundTime<<" "

<<"响应比："<< (1+(run\_proc->StartTime-run\_proc->ArriveTime)/run\_proc->ServeTime)<<endl;

run\_proc->ProcState = 'f' ; //状态修改

run\_proc->ResponseRatio=-2 ; //响应比设置为-2

process\_count++; //完成进程数+1

}

else

{

cout<<time<<"时刻 ，"<<run\_proc->name<<" 进入运行判定 " <<endl;

cout<<"已经运行了"<<run\_proc->RunTime<<"还需要执行"<<run\_proc->MoreTime<<endl;

}

if(run\_proc->ProcState=='f') //当前进程运行完成，则进入下一个进程

{

if(process\_count < MAX\_NUM) //判断是否还有进程待运行

run\_proc=run\_proc->next;

else

break;

if (run\_proc->ProcState=='w' ) //还未结束的进程

{

if(time<run\_proc->ArriveTime) //如果进程当前还没到来

{

time=run\_proc->ArriveTime;

}

}

else

{

for( ; run\_proc->ProcState=='f' && run\_proc->next!=NULL ; )

run\_proc=run\_proc->next; //搜索还未结束的下一个进程

}

}

else //否则根据响应比从大到小排序

{

HRRN\_SortList(head);

run\_proc=head;

if(run\_proc->ArriveTime>time&&run\_proc->ProcState=='w') //如果当前进程还未到来

{

time=run\_proc->ArriveTime;

}

}

}

}

int main()

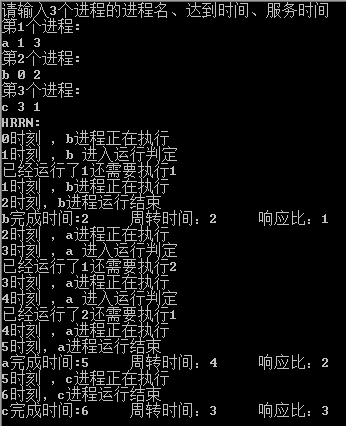
{

FCFS\_initial();

HRRN();

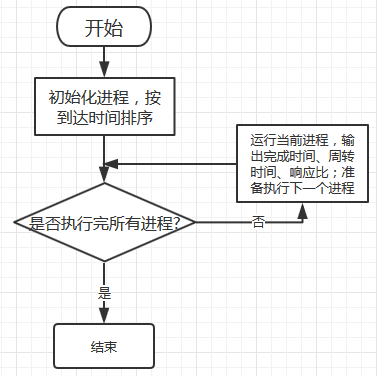
return 0;

}

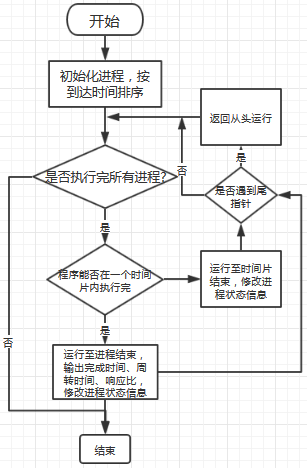


# 程序流程图

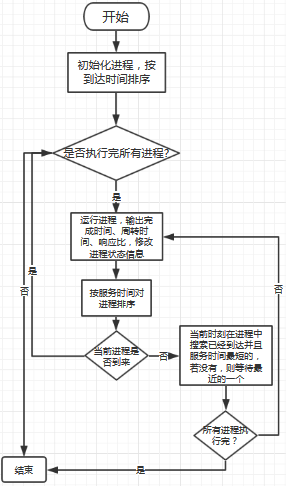
FCFS:



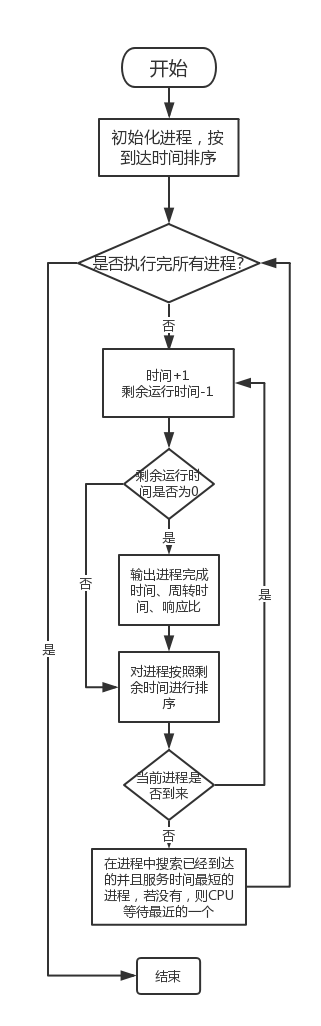
RR:



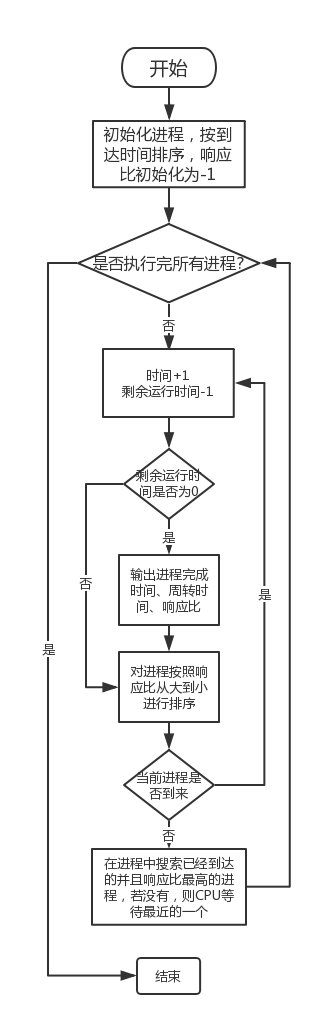
SPN:



SRT:



HRRN:



# 实验总结

通过本次实验，对比了几种调度算法。FCFS对执行长进程比执行短进程更好；RR可以减少FCFS对短进程不利的情况；SPN是非抢占策略，也减少了FCFS对长进程的偏向；HRRN下，长进程被饿死的可能性得以降低。