**进程调度算法的模拟实现**

* **实验目的**

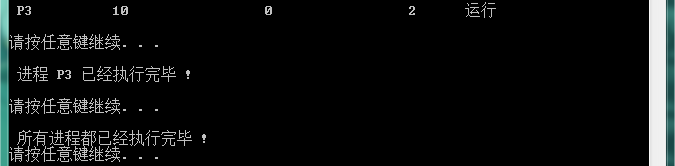
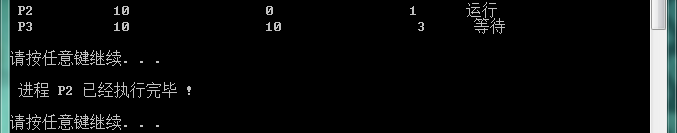
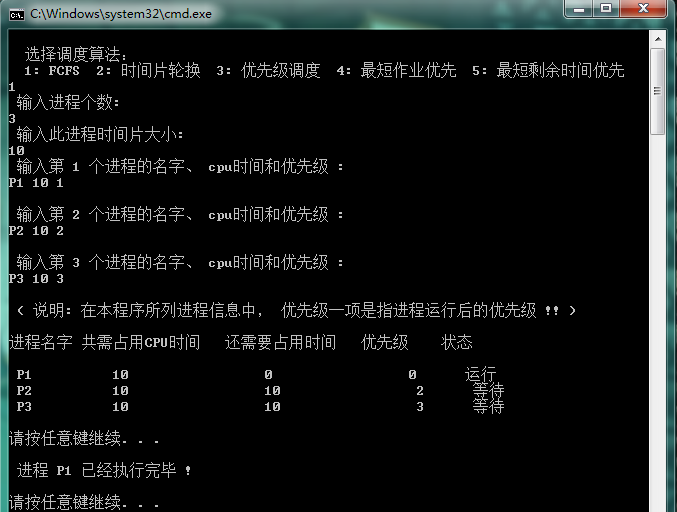
1．本实验模拟在单处理机情况下的处理机调度问题，加深对进程调度的理解。

2．利用程序设计语言编写算法，模拟实现先到先服务算法FCFS、轮转调度算法RR、最短作业优先算法SJF、优先级调度算法PRIOR、最短剩余时间优先算法SRTF。

3．进行算法评价，计算平均等待时间和平均周转时间。

* **实验内容及结果**

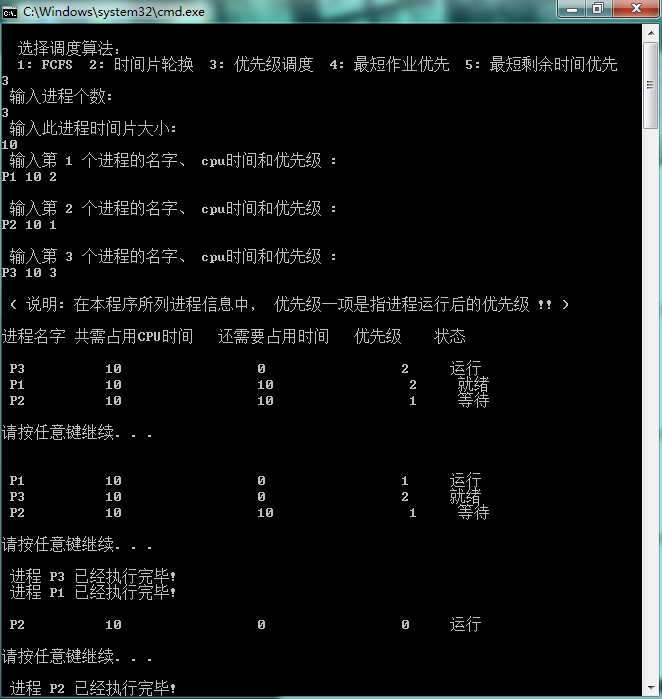
1．先来先服务算法



2．轮转调度算法



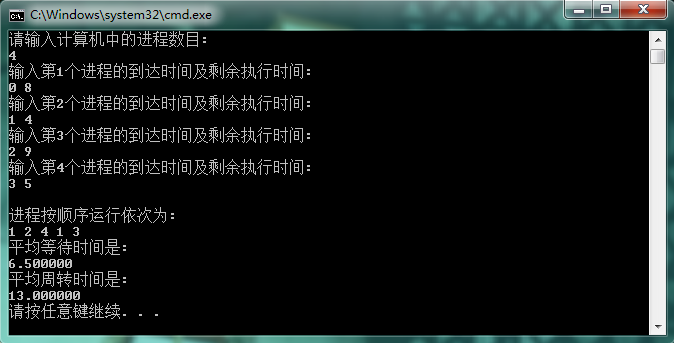
3. 优先级调度算法



4. 最短时间优先算法



5. 最短剩余时间优先算法



* **实验总结**

在此次模拟过程中，将SRTF单独拿了出来用指针表示，而其余均用数组表示。

* **完整代码**

【Srtf.cpp代码如下：】

//最短剩余时间优先算法的实现

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

typedef struct

{

int remain\_time; //进程剩余执行时间

int arrive\_time; //进程到达时间

int Tp; //进入就绪队列的时间

int Tc; //进入执行队列的时间

int To; //进程执行结束的时间

int number; //进程编号

}Process\_Block; //定义进程模块

typedef struct \_Queue

{

Process\_Block PB;

struct \_Queue \*next;

}\_Block,\*Process; //定义一个进程模块队列中结点

typedef struct

{

Process head; //队列头指针

Process end; //队列尾指针

}Process\_Queue; //进程队列

Process\_Queue PQ; //定义一个全局队列变量

int t; //全局时间

Process Run\_Now; //当前正在运行的进程，作为全局变量

void InitQueue(Process\_Queue PQ)

{

PQ.head ->next = NULL;

PQ.end ->next = PQ.head;

}/\*初始化队列\*/

int IsEmpty(Process\_Queue PQ)

{

if(PQ.end->next == PQ.head)

return 1; //队列空的条件为头指针指向尾指针并且尾指针指向头指针

else

return 0;

}/\*判定队列是否为空队列\*/

void EnQueue(Process\_Queue PQ,Process P)

{

Process temp =(Process)malloc(sizeof(\_Block));

temp = PQ.end;

temp->next->next = P;

PQ.end->next = P;

}/\*插入队列操作\*/

Process DeQueue(Process\_Queue PQ)

{

if(IsEmpty(PQ))

return NULL;

Process temp = PQ.head->next;

PQ.head->next= temp ->next;

if(PQ.end->next == temp)

PQ.end->next = PQ.head;

return temp;

}/\*出列操作\*/

Process ShortestProcess(Process\_Queue PQ)

{

if(IsEmpty(PQ)) //如果队列为空，返回

{

if(!Run\_Now)

return NULL;

else

return Run\_Now;

}

Process temp,shortest,prev;

int min\_time;

if(Run\_Now) //如果当前有进程正在执行，

{

shortest = Run\_Now; //那么最短进程初始化为当前正在执行的进程，

min\_time = Run\_Now->PB.remain\_time;

}

else //如果当前没有进程执行，

{

shortest = PQ.head->next; //则最短进程初始化为队列中第一个进程

min\_time = PQ.head->next->PB.remain\_time;

}

temp = PQ.head;

prev = temp;

while(temp->next)

{

if(temp->next->PB.remain\_time <min\_time) //如果当前进程的剩余时间比min\_time短，

{

shortest = temp->next; //则保存当前进程，

min\_time = shortest->PB.remain\_time;

prev=temp; //及其前驱

}

temp=temp->next;

}

if(shortest == PQ.end->next) //如果最短剩余时间进程是队列中最后一个进程，

PQ.end->next = prev; //则需要修改尾指针指向其前驱

prev->next = shortest->next; //修改指针将最短剩余时间进程插入到队头

return shortest;

}/\*调度最短剩余时间的进程至队头\*/

void Run()

{

Run\_Now->PB.remain\_time--; //某一时间运行它的剩余时间减

return;

}/\*运行函数\*/

void Wait()

{

return ;

}

int sum(int array[],int n)

{

int i,sum=0;

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

sum+=array[i];

return sum;

}

int main()

{

PQ.head = (Process)malloc(sizeof(\_Block));

PQ.end = (Process)malloc(sizeof(\_Block));

Run\_Now = (Process)malloc(sizeof(\_Block));

Run\_Now =NULL;

InitQueue(PQ);

int i,N,Total\_Time=0; //Total\_Time为所有进程的执行时间之和

printf("请输入计算机中的进程数目:\n");

scanf("%d",&N);

Process \*P,temp;

P = (Process\*)malloc(N\*sizeof(Process));

int \*wt,\*circle\_t;

wt =(int\*)malloc(N\*sizeof(int));

circle\_t =(int\*)malloc(N\*sizeof(int));

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

{

P[i] = (Process)malloc(sizeof(\_Block));

P[i]->PB.number =i+1;

P[i]->next =NULL;

wt[i] =0;

circle\_t[i] =0;

printf("输入第%d个进程的到达时间及剩余执行时间:\n",i+1);

scanf("%d %d",&P[i]->PB.arrive\_time,&P[i]->PB.remain\_time);

}

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

Total\_Time+=P[i]->PB.remain\_time;

printf("\n进程按顺序运行依次为:\n");

i=0;

int k=0;

for(t=0;;t++)

{

if(Run\_Now) //如果当前有进程正在执行

{

Run();

if(t == P[i]->PB.arrive\_time) //如果当前时间正好有进程进入

{

if(P[i]->PB.remain\_time < Run\_Now->PB.remain\_time)

{

temp = P[i];

P[i] = Run\_Now;

Run\_Now = temp; //则调度它至运行队列中，

Run\_Now->PB.Tp=t;

Run\_Now->PB.Tc=t;

wt[Run\_Now->PB.number-1]+=Run\_Now->PB.Tc-Run\_Now->PB.Tp;

printf("%d ",Run\_Now->PB.number);

}

EnQueue(PQ,P[i]); //并将当前运行进程重新插入队列中

P[i]->PB.Tp=t;

k++;

i=(i+1)>(N-1)?(N-1):(i+1);

}

if(Run\_Now->PB.remain\_time == 0) //如果当前进程运行结束，

{

Run\_Now->PB.To=t; //进程运行结束的时间

circle\_t[Run\_Now->PB.number-1] +=t-Run\_Now->PB.arrive\_time;

free(Run\_Now); //则将它所占资源释放掉，

Run\_Now =NULL; //并修改Run\_Now为NULL

Run\_Now = ShortestProcess(PQ); //从就绪队列中调出最短剩余时间进程至队头，

if(!Run\_Now) //如果队列为空，转为等待状态

{

if(IsEmpty(PQ) && k >= N) break;

Wait();

continue;

}

else

{

Run\_Now->PB.Tc=t;

wt[Run\_Now->PB.number-1]+=Run\_Now->PB.Tc-Run\_Now->PB.Tp;

printf("%d ",Run\_Now->PB.number);

}

}

}

else //如果当前运行进程为空，那么

{

if(t == P[i]->PB.arrive\_time) //如果正好这时有进程入队

{

k++;

EnQueue(PQ,P[i]);

Run\_Now = DeQueue(PQ); //则直接被调入运行队列中

Run\_Now->PB.Tp=t;

Run\_Now->PB.Tc=t;

printf("%d ",Run\_Now->PB.number);

i=(i+1)>(N-1)?(N-1):(i+1);

}

else

{

Wait();

continue;

}

}

}

printf("\n");

printf("平均等待时间是:\n%f\n",((float)sum(wt,N))/N);

printf("平均周转时间是:\n%f\n",((float)sum(circle\_t,N))/N);

return 0;

}

//////////////////////////////////////////////////////

【Process.cpp代码如下：】

#include<iostream>

#include<string>

using namespace std;

class Process

{

public:

string ProcessName; // 进程名字

int Time; // 进程需要时间

int leval; // 进程优先级

int LeftTime; // 进程运行一段时间后还需要的时间

};

void Copy ( Process proc1, Process proc2); // 把proc2赋值给proc1

void Sort( Process pr[], int size) ; // 此排序后按优先级从大到小排列

void sort1(Process pr[], int size) ; // 此排序后按需要的cpu时间从小到大排列

void Fcfs( Process pr[], int num, int Timepice); // 先来先服务算法

void TimeTurn( Process process[], int num, int Timepice); // 时间片轮转算法

void Priority( Process process[], int num, int Timepice); // 优先级算法

void main()

{

int a;

cout<<endl;

cout<<" 选择调度算法："<<endl;

cout<<" 1: FCFS 2: 时间片轮换 3: 优先级调度 4: 最短作业优先 5: 最短剩余时间优先"<<endl;

cin>>a;

const int Size =30;

Process process[Size] ;

int num;

int TimePice;

cout<<" 输入进程个数:"<<endl;

cin>>num;

cout<<" 输入此进程时间片大小: "<<endl;

cin>>TimePice;

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

{

string name;

int CpuTime;

int Leval;

cout<<" 输入第"<< i+1<<" 个进程的名字、cpu时间和优先级:"<<endl;

cin>>name;

cin>> CpuTime>>Leval;

process[i].ProcessName =name;

process[i].Time =CpuTime;

process[i].leval =Leval;

cout<<endl;

}

for ( int k=0;k<num;k++)

process[k].LeftTime=process[k].Time ;//对进程剩余时间初始化

cout<<" ( 说明: 在本程序所列进程信息中，优先级一项是指进程运行后的优先级!! )";

cout<<endl; cout<<endl;

cout<<"进程名字"<<"共需占用CPU时间 "<<" 还需要占用时间 "<<" 优先级"<<" 状态"<<endl;

if(a==1)

Fcfs(process,num,TimePice);

else if(a==2)

TimeTurn( process, num, TimePice);

else if(a==3)

{

Sort( process, num);

Priority( process , num, TimePice);

}

else // 最短作业算法，先按时间从小到大排序，再调用Fcfs算法即可

{

sort1(process,num);

Fcfs(process,num,TimePice);

}

}

void Copy ( Process proc1, Process proc2)

{

proc1.leval =proc2.leval ;

proc1.ProcessName =proc2.ProcessName ;

proc1.Time =proc2.Time ;

}

void Sort( Process pr[], int size) //以进程优先级高低排序

{// 直接插入排序

for( int i=1;i<size;i++)

{

Process temp;

temp = pr[i];

int j=i;

while(j>0 && temp.leval<pr[j-1].leval)

{

pr[j] = pr[j-1];

j--;

}

pr[j] = temp;

} // 直接插入排序后进程按优先级从小到大排列

for( int d=size-1;d>size/2;d--)

{

Process temp;

temp=pr [d];

pr [d] = pr [size-d-1];

pr [size-d-1]=temp;

} // 此排序后按优先级从大到小排列

}

/\* 最短作业优先算法的实现\*/

void sort1 ( Process pr[], int size) // 以进程时间从低到高排序

{// 直接插入排序

for( int i=1;i<size;i++)

{

Process temp;

temp = pr[i];

int j=i;

while(j>0 && temp.Time < pr[j-1].Time )

{

pr[j] = pr[j-1];

j--;

}

pr[j] = temp;

}

}

/\* 先来先服务算法的实现\*/

void Fcfs( Process process[], int num, int Timepice)

{ // process[] 是输入的进程，num是进程的数目，Timepice是时间片大小

while(true)

{

if(num==0)

{

cout<<" 所有进程都已经执行完毕!"<<endl;

exit(1);

}

if(process[0].LeftTime==0)

{

cout<<" 进程"<<process[0].ProcessName<< " 已经执行完毕!"<<endl;

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

process[i]=process[i+1];

num--;

}

else if(process[num-1].LeftTime==0)

{

cout<<" 进程"<<process[num-1].ProcessName<< " 已经执行完毕!"<<endl;

num--;

}

else

{

cout<<endl; //输出正在运行的进程

process[0].LeftTime=process[0].LeftTime- Timepice;

process[0].leval =process[0].leval-1;

cout<<" "<<process[0].ProcessName <<" "<<process[0].Time <<" ";

cout<<process[0].LeftTime <<" "<<process[0].leval<<" 运行";

cout<<endl;

for(int s=1;s<num;s++)

{

cout<<" "<<process[s].ProcessName <<" "<<process[s].Time <<" ";

cout<<process[s].LeftTime <<" "<<process[s].leval<<" 等待"<<endl; ;

}

} // else

cout<<endl;

system(" pause");

cout<<endl;

} // while

}

/\* 时间片轮转调度算法实现\*/

void TimeTurn( Process process[], int num, int Timepice)

{

while(true)

{

if(num==0)

{

cout<<" 所有进程都已经执行完毕!"<<endl;

exit(1);

}

if(process[0].LeftTime==0)

{

cout<<" 进程"<<process[0].ProcessName<< " 已经执行完毕!"<<endl;

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

process[i]=process[i+1];

num--;

}

if( process[num-1].LeftTime ==0 )

{

cout<<" 进程" << process[num-1].ProcessName <<" 已经执行完毕! "<<endl;

num--;

}

else if(process[0].LeftTime > 0)

{

cout<<endl; //输出正在运行的进程

process[0].LeftTime=process[0].LeftTime- Timepice;

process[0].leval =process[0].leval-1;

cout<<" "<<process[0].ProcessName <<" "<<process[0].Time <<" ";

cout<<process[0].LeftTime <<" "<<process[0].leval<<" 运行";

cout<<endl;

for(int s=1;s<num;s++)

{

cout<<" "<<process[s].ProcessName <<" "<<process[s].Time <<" ";

cout<<process[s].LeftTime <<" "<<process[s].leval;

if(s==1)

cout<<" 就绪"<<endl;

else

cout<<" 等待"<<endl;

}

Process temp;

temp = process[0];

for( int j=0;j<num;j++)

process[j] = process[j+1];

process[num-1] = temp;

} // else

cout<<endl;

system(" pause");

cout<<endl;

} // while

}

/\* 优先级调度算法的实现\*/

void Priority( Process process[], int num, int Timepice)

{

while( true)

{

if(num==0)

{

cout<< "所有进程都已经执行完毕!"<<endl;

exit(1);

}

if(process[0].LeftTime==0)

{

cout<<" 进程" << process[0].ProcessName <<" 已经执行完毕! "<<endl;

for( int m=0;m<num;m++)

process[m] = process[m+1]; //一个进程执行完毕后从数组中删除

num--; // 此时进程数目减少一个

}

if( num!=1 && process[num-1].LeftTime ==0 )

{

cout<<" 进程" << process[num-1].ProcessName <<" 已经执行完毕! "<<endl;

num--;

}

if(process[0].LeftTime > 0)

{

cout<<endl; //输出正在运行的进程

process[0].LeftTime=process[0].LeftTime- Timepice;

process[0].leval =process[0].leval-1;

cout<<" "<<process[0].ProcessName <<" "<<process[0].Time <<" ";

cout<<process[0].LeftTime <<" "<<process[0].leval<<" 运行";

cout<<endl; // 输出其他进程

for(int s=1;s<num;s++)

{

cout<<" "<<process[s].ProcessName <<" "<<process[s].Time <<" ";

cout<<process[s].LeftTime <<" "<<process[s].leval ;

if(s==1)

cout<<" 就绪"<<endl;

else

cout<<" 等待 "<<endl;

}

} // else

Sort(process, num);

cout<<endl;

system(" pause");

cout<<endl;

} // while

}