# 课题二：

**多级反馈队列调度算法**

1. 背景/需求分析

在科学技术迅速发展的当代以及经济发展的需求,服务器和任务的数量都在高速增长,同时资源调度的方式以及数量也在成倍增长，目前存在的多种调度算法中，有的调度算法有利于长作业，有的有利于短作业，都有其优点和缺点。然而，有一种调度算法既适用于长作业调度，又能适用于短作业调度，还能很好的处理长作业和短作业的先后调度关系，比如多级反馈队列调度算法。多级反馈队列调度算法是目前公认的较好的一种进程调度算法，它能较好的满足各类进程的需要。

多级反馈队列调度算法是一种CPU处理机调度算法,UNIX操作系统采取的便是这种调度算法。 多级反馈队列调度算法即能使高优先级的作业得到响应又能使短作业(进程)迅速完成。 多级反馈队列调度算法是一种性能较好的作业低级调度策略,能够满足各类用户的需要。对于分时交互型短作业,系统通常可在第一队列(高优先级队列)规定的时间片内让其完成工作,使终端型用户都感到满意;对短的批处理作业,通常,只需在第一或第一、第二队列(中优先级队列)中各执行一个时间片就能完成工作,周转时间仍然很短;对长的批处理作业,它将依次在第一、第二、......,各个队列中获得时间片并运行,决不会出现得不到处理的情况。此系统模拟了多级反馈队列调度算法及其实现。

1. 系统设计

程序采用键盘输入或随机生成的方式来模拟网络信号、文件操作等各类等待事件。程序的主体部分为三个函数体，init，MFQ，print。使用结构体来存储进程的信息。

主函数main中给用户提供随机测试样例和键盘输入测试样例两种选择，并打印菜单。在init函数中完成变量输入与初始化，包括就绪队列总数、第一队列时间片大小（该程序中设定后一队列时间片大小为前一队列时间片大小两倍）、进程总数、各个进程的到达时间和运行时间。在MFQ函数中主要模拟多级反馈队列调度算法。设置多级就绪队列，优先级最低的采用RR算法，其他采用FCFS算法。在这里先把所有的队列都用FCFS排列，先调度高优先度的进程，若在时间片内不能运行完就放入下一队列，这样能够保证优先级最低的队列是RR。如果有高优先级的进程到达，则放弃正在调度的进程，转向调度最高优先级的进程。在print函数中，将算法计算出的进程完成时间、周转时间、带权周转时间、平均周转时间、平均带权周转时间输出，作为测试结果。

1. 详细设计

设置多级就绪队列，各级队列优先级从高到低，时间片从小到大。

新进程到达时先进入第1级队列，按FCFS原则排队等待被分配时间片。若用完时间片进程还未结束，则进程进入下一级队列队尾。如果此时已经在最下级的队列，则重新放回最下级队列队尾。只有第k级队列为空时，才会为k+1级队头的进程分配时间片被抢占处理机的进程重新放回原队列队尾。

对于优先级最低的队列来说，里面是遵循时间片轮转法（RR）。也就是说，位于队列QN中有M个作业，它们的运行时间是通过QN这个队列所设定的时间片来确定的；对于其他队列，遵循的是先来先服务算法（FCFS），每一进程分配一定的时间片，若时间片运行完时进程未结束，则进入下一优先级队列的末尾。

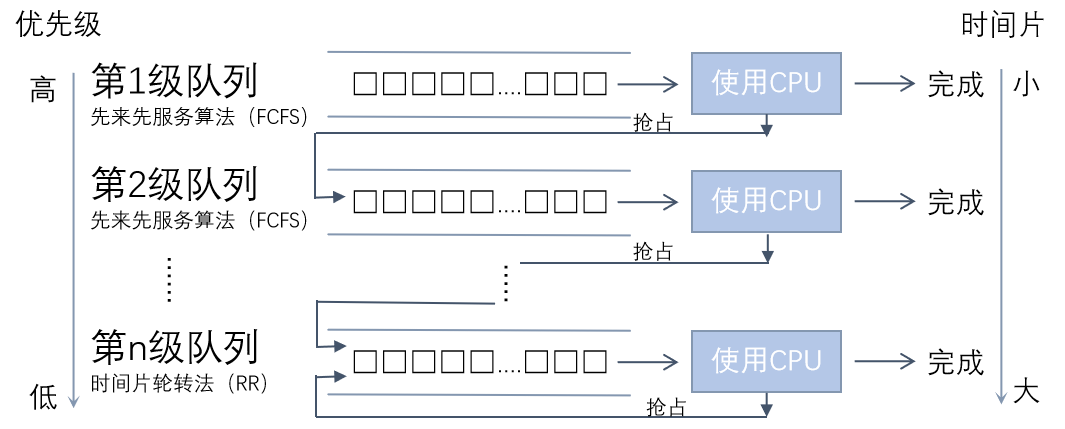


图3-1算法示意图

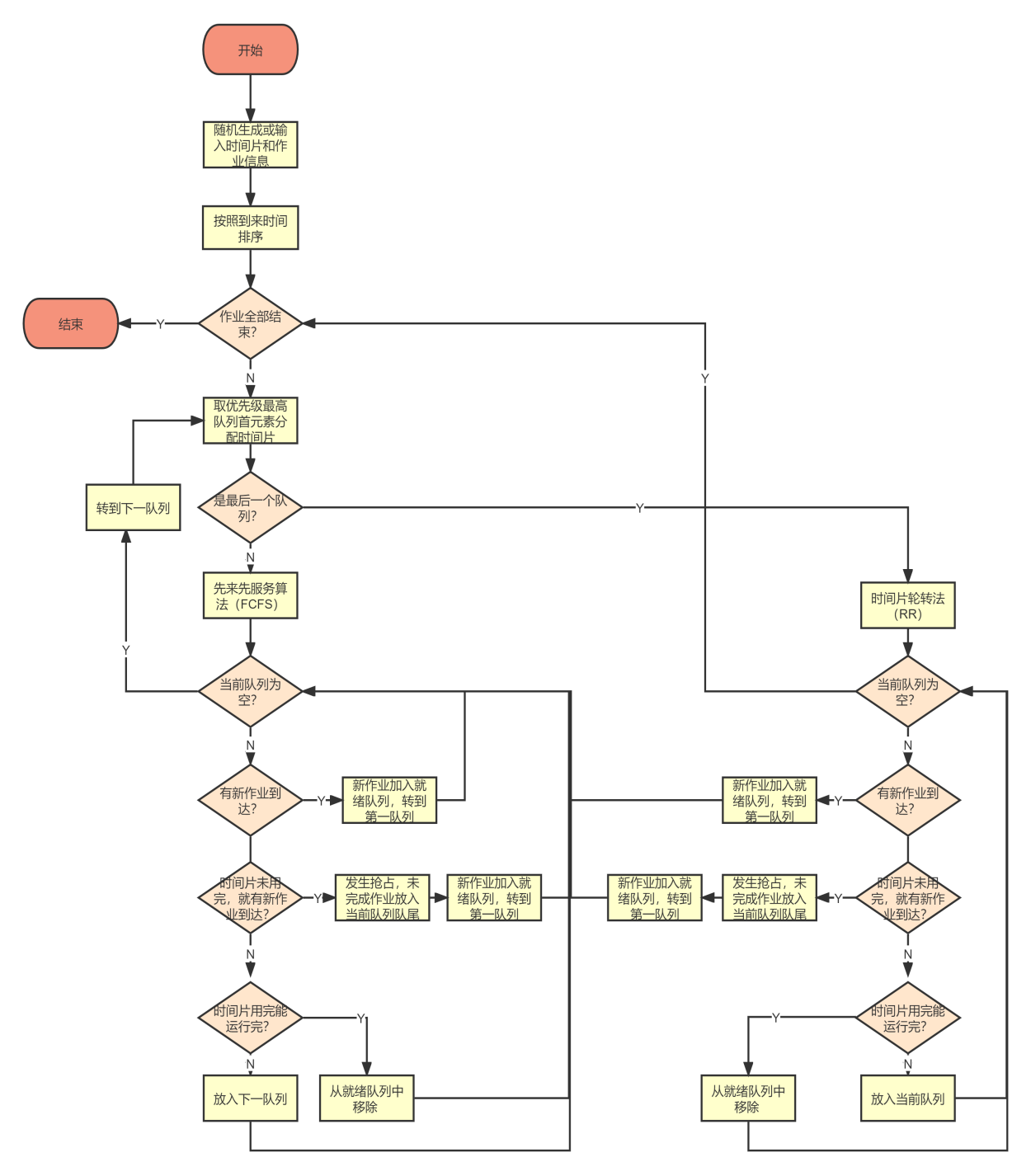


图3-2算法流程图

1. 系统编码

main.cpp

#include<bits/stdc++.h>

using namespace std;

const int MAXSIZE=100;

int qsize=7; //就绪队列总数默认为7

struct node

{

int id; //进程编号

int num; //进程个数

int arrive; //到达时间

int finish; //完成时间

int run; //运行时间

int turn; //周转时间

double wturn; //带权周转时间

};

vector<node> v;

//进程按照到达时间先后排序

bool cmp(node x,node y)

{

return x.arrive<y.arrive;

}

//变量输入与初始化

void init(int test,int &tslice)

{

srand(time(0));

int number; //第一队列时间片大小，进程个数

node temp;

if(test==1)

{

tslice=rand()%5+1;

number=rand()%30+2;

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

{

temp.id=i;

temp.arrive=rand()%30;

temp.run=rand()%50+1;

temp.finish=0;

v.push\_back(temp);

}

}

else

{

cout<<"请输入就绪队列总数：";cin>>qsize;

cout<<"请输入第一队列时间片大小（默认后一队列时间片大小为前一队列时间片大小两倍）：";

cin>>tslice;

cout<<"请输入进程个数：";cin>>number;

cout<<"请输入进程的到达时间和运行时间：";

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

{

temp.id=i;

cin>>temp.arrive>>temp.run;

temp.finish=0;

v.push\_back(temp);

}

}

}

//多级反馈队列调度算法(抢占式)

void MFQ(int tslice)

{

queue<node> ready[MAXSIZE]; //定义就绪队列

vector<node> ans; //答案

sort(v.begin(),v.end(),cmp); //按照FCFS原则排序

int time=(\*v.begin()).arrive; //第一个进程到达时间

ready[0].push(\*v.begin());v.erase(v.begin());

cout<<endl<<"--测试进程运行过程--"<<endl;

while(!v.empty())

{

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

{

int nowslice=tslice\*pow(2,i);

if(i!=qsize-1)//当前不是最后一个

{

while(!ready[i].empty())

{

if(!v.empty()&&time>=(\*v.begin()).arrive)//有新进程到达，加入就绪队列，转到第一队列

{

ready[0].push(\*v.begin());

v.erase(v.begin());

i=0;nowslice=tslice\*pow(2,i);

continue;

}

if(!v.empty()&&((time+nowslice>(\*v.begin()).arrive&&ready[i].front().finish+nowslice<ready[i].front().run)

||(time+ready[i].front().run-ready[i].front().finish>(\*v.begin()).arrive&&ready[i].front().finish+nowslice>=ready[i].front().run)))//时间片未用完，就有新进程到达

{

//cout<<"\*\*\*\*time:"<<time<<" (\*v.begin()).arrive:"<<(\*v.begin()).arrive<<" nowslice:"<<nowslice<<" "<<ready[i].front().run-ready[i].front().finish<<endl;

ready[i].front().finish+=(\*v.begin()).arrive-time;

ready[i].push(ready[i].front());

ready[i].pop();

//cout<<"ready[i].front().finish:"<<ready[i].front().finish<<" time:"<<time<<" i:"<<i<<endl;

cout<<"->p"<<ready[i].front().id<<"("<<(\*v.begin()).arrive-time<<")"<<endl;

time=(\*v.begin()).arrive;

ready[0].push(\*v.begin());

v.erase(v.begin());

i=0;nowslice=tslice\*pow(2,i);

continue;

}

else //整个时间片都没有新进程

{

if(ready[i].front().finish+nowslice<ready[i].front().run)//时间片用完没运行完,加入下一队列队尾

{

ready[i].front().finish+=nowslice;

ready[i+1].push(ready[i].front());

cout<<"->p"<<ready[i].front().id<<"("<<nowslice<<")"<<endl;

ready[i].pop();

time+=nowslice;

}

else//该进程结束

{

time+=ready[i].front().run-ready[i].front().finish;

cout<<"->p"<<ready[i].front().id<<"("<<ready[i].front().run-ready[i].front().finish<<")"<<endl;

ready[i].front().finish=time-1;

ready[i].front().turn=ready[i].front().finish-ready[i].front().arrive+1;

ready[i].front().wturn=(double)ready[i].front().turn/ready[i].front().run;

//从就绪队列中移除

ans.push\_back(ready[i].front());

ready[i].pop();

}

}

}

}

else //按照RR算法调度

{

while(!ready[i].empty())

{

if(!v.empty()&&time>=(\*v.begin()).arrive)//有新进程到达，加入就绪队列，转到第一队列

{

ready[0].push(\*v.begin());

v.erase(v.begin());

i=-1;

break;

}

if(!v.empty()&&((time+nowslice>(\*v.begin()).arrive&&ready[i].front().finish+nowslice<ready[i].front().run)

||(time+ready[i].front().run-ready[i].front().finish>(\*v.begin()).arrive&&ready[i].front().finish+nowslice>=ready[i].front().run)))//时间片未用完，就有新进程到达

{

//cout<<"\*\*\*\*time:"<<time<<" (\*v.begin()).arrive:"<<(\*v.begin()).arrive<<" nowslice:"<<nowslice<<" "<<ready[i].front().run-ready[i].front().finish<<endl;

ready[i].front().finish+=(\*v.begin()).arrive-time;

ready[i].push(ready[i].front());

ready[i].pop();

//cout<<"ready[i].front().finish:"<<ready[i].front().finish<<" time:"<<time<<" i:"<<i<<endl;

cout<<"->p"<<ready[i].front().id<<"("<<(\*v.begin()).arrive-time<<")"<<endl;

time=(\*v.begin()).arrive;

ready[0].push(\*v.begin());

v.erase(v.begin());

i=-1;

break;

}

else //整个时间片都没有新进程

{

if(ready[i].front().finish+nowslice<ready[i].front().run)//时间片用完没运行完,加入下一队列队尾

{

ready[i].front().finish+=nowslice;

ready[i].push(ready[i].front());

cout<<"->p"<<ready[i].front().id<<"("<<nowslice<<")"<<endl;

ready[i].pop();

time+=nowslice;

}

else//该进程结束

{

time+=ready[i].front().run-ready[i].front().finish;

cout<<"->p"<<ready[i].front().id<<"("<<ready[i].front().run-ready[i].front().finish<<")"<<endl;

ready[i].front().finish=time-1;

ready[i].front().turn=ready[i].front().finish-ready[i].front().arrive+1;

ready[i].front().wturn=(double)ready[i].front().turn/ready[i].front().run;

//从就绪队列中移除

ans.push\_back(ready[i].front());

ready[i].pop();

}

}

}

}

}

}

v=ans;

sort(v.begin(),v.end(),cmp);

}

//输出

void print()

{

int sumturn=0;

double sumwturn=0;

cout<<endl<<"--测试结果--"<<endl;

cout<<" 进程编号 到达时间 运行时间 完成时间 周转时间 带权周转时间"<<endl;

for(auto i=v.begin();i!=v.end();i++)

{

sumturn+=(\*i).turn;

sumwturn+=(\*i).wturn;

printf("%6d%10d%10d%10d%10d%13.2lf\n",(\*i).id,(\*i).arrive,(\*i).run,(\*i).finish,(\*i).turn,(\*i).wturn);

}

printf("平均周转时间:%.2lf\n",(double)sumturn/v.size());

printf("平均带权周转时间:%.2lf\n",sumwturn/v.size());

}

signed main()

{

while(1)

{

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

cout<<"1.随机样例测试"<<endl;

cout<<"2.输入样例测试"<<endl;

cout<<"0.结束测试"<<endl;

cout<<"请输入编号:";

int test;cin>>test;

if(!test)break;

else if(test!=1&&test!=2){cout<<"输入错误"<<endl;continue;}

int tslice=1;

init(test,tslice);

MFQ(tslice);

print();

v.clear();

}

}

1. 系统测试

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 序号 | 条件 | 预期结果 | 测试结果 | 测试过程 |
| ① | qsize=7;num=20 | finish=469 | OK | 图5-1 |
| ② | qsize=10;num=6 | finish=24 | OK | 图5-2 |
| ③ | qsize=7;num=30 | finish=770 | OK | 图5-3 |
| ④ | qsize=10;num=13 | finish=270 | OK | 图5-4 |
| ⑤ | qsize=7;num=29 | finish=719 | OK | 图5-5 |
| ⑥ | qsize=7;num=14 | finish=157 | OK | 图5-6 |
| ⑦ | qsize=7;num=3 | finish=111 | OK | 图5-7 |

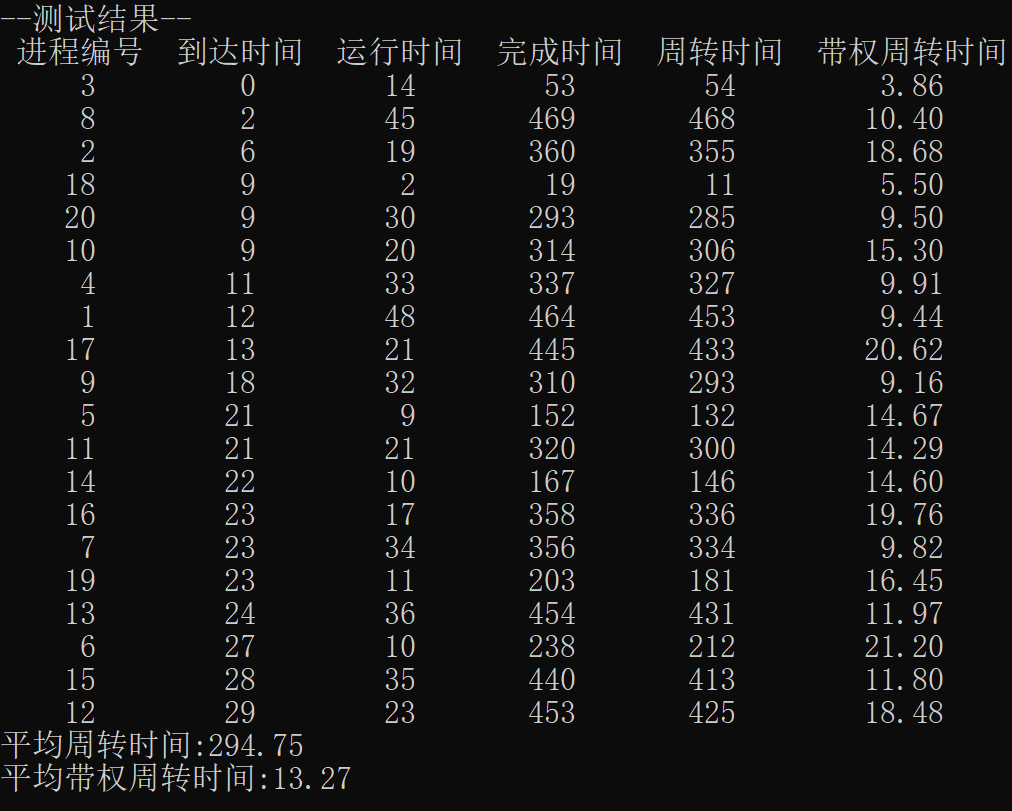


图5-1



图5-2

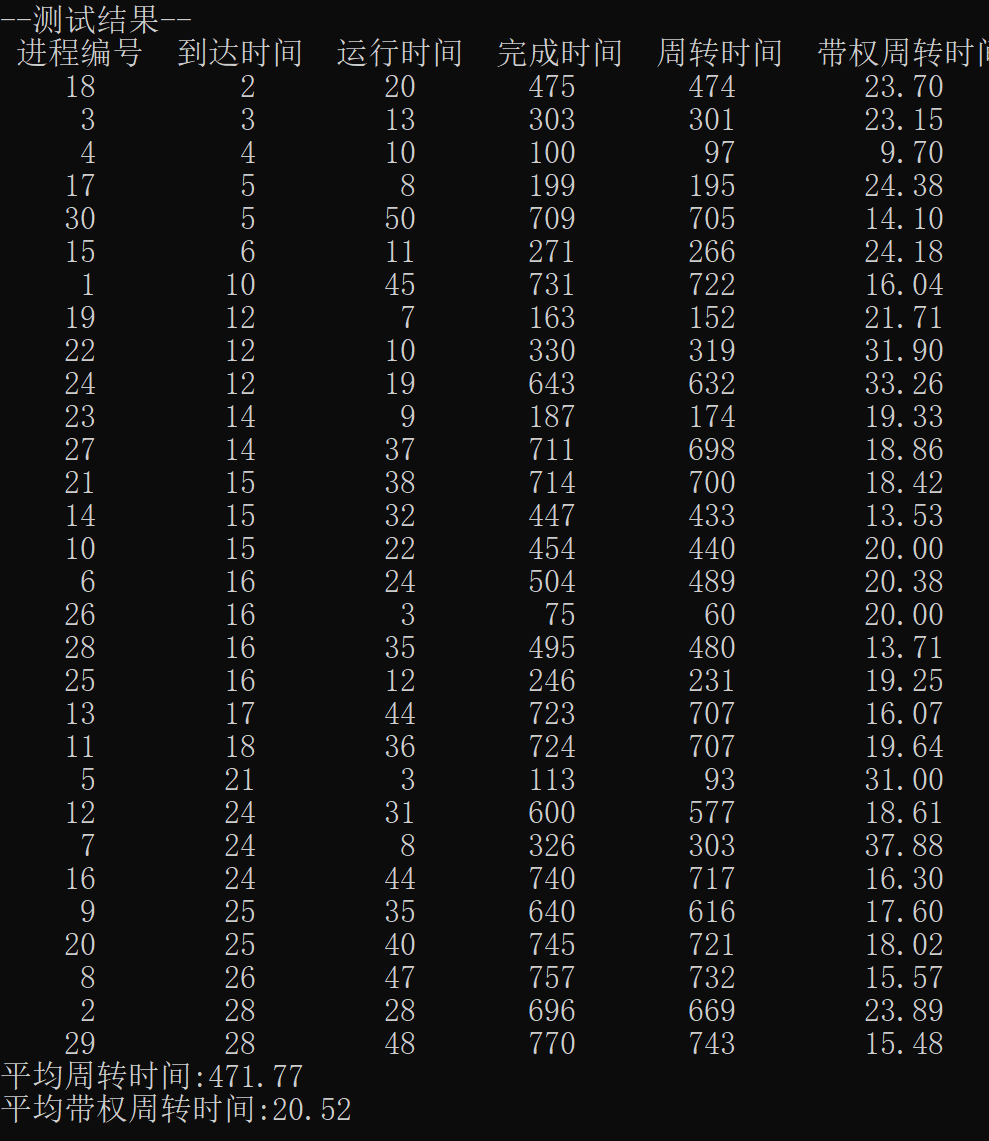


图5-3



图5-4

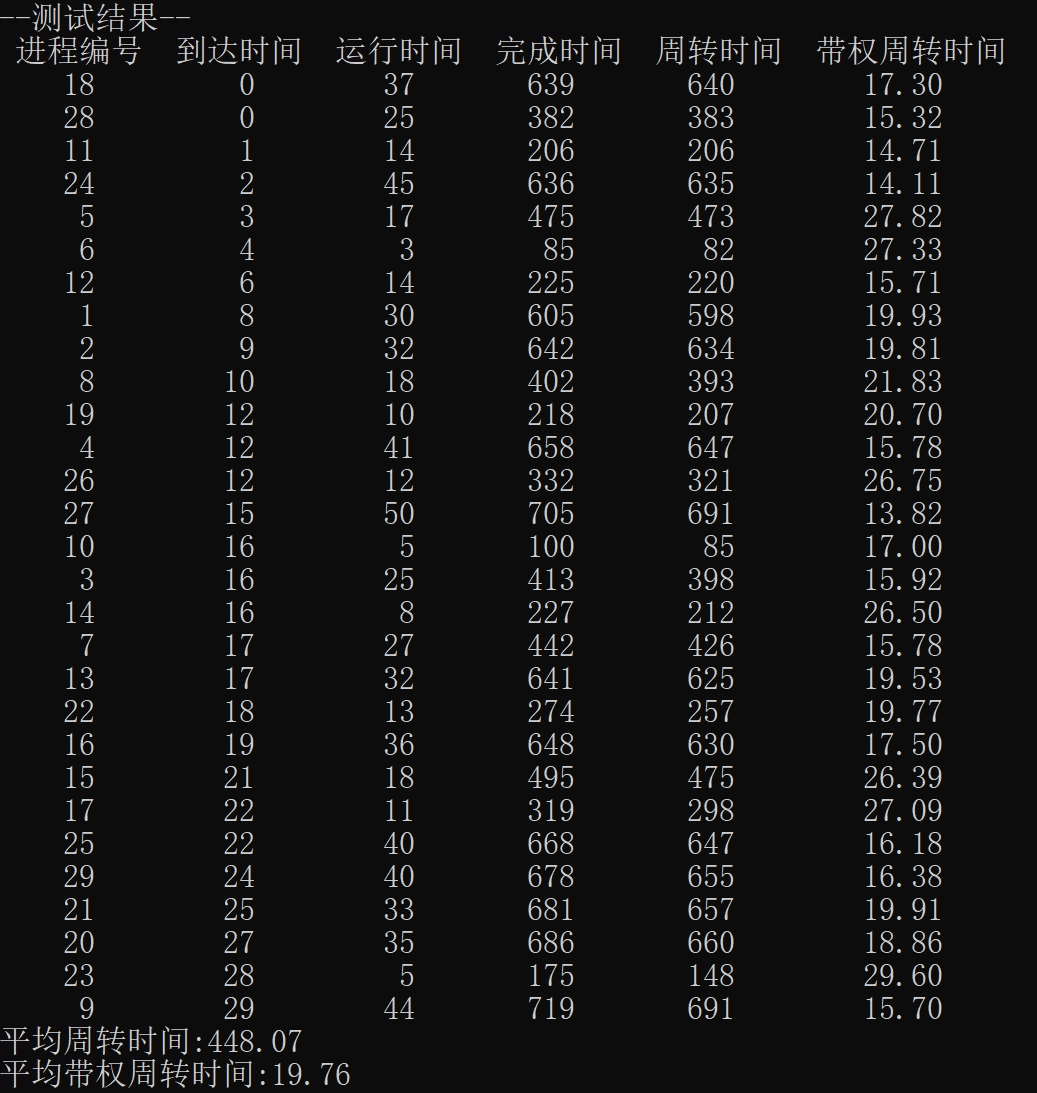


图5-5

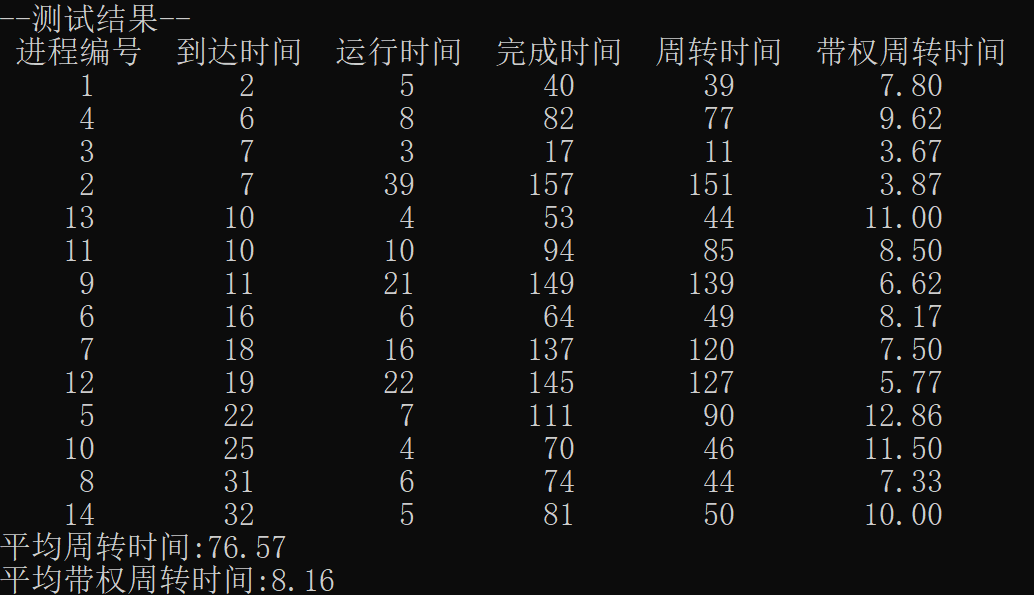


图5-6

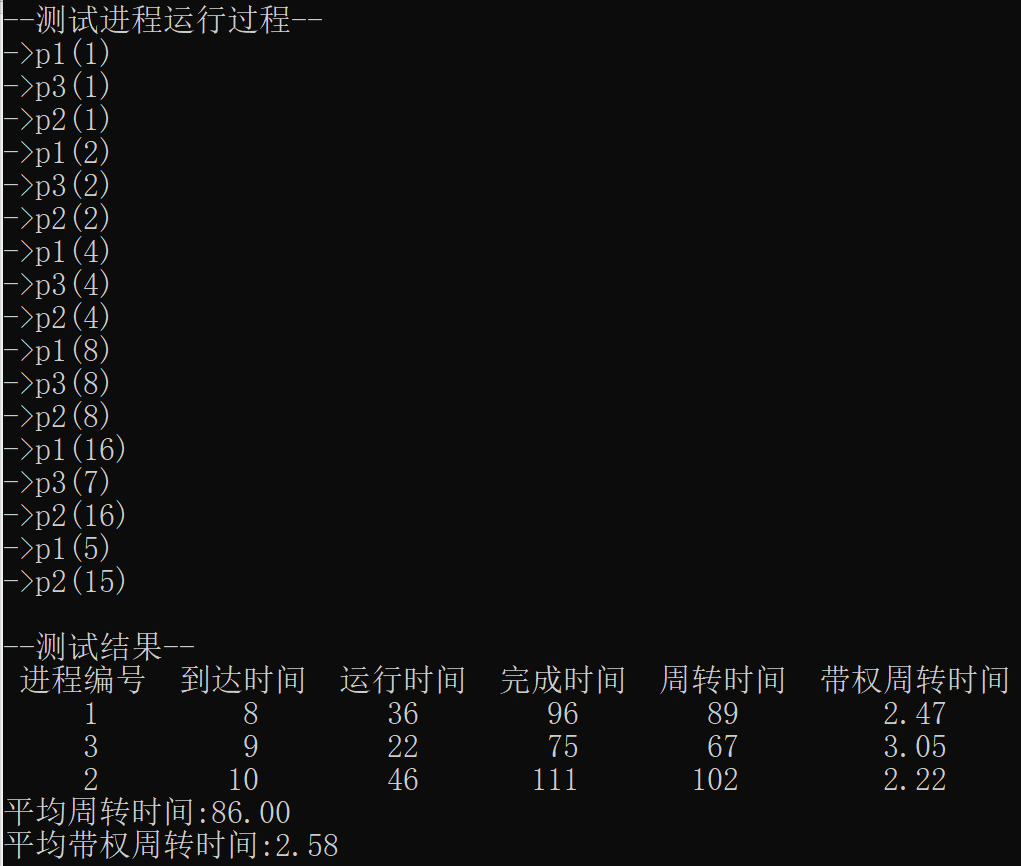


图5-7

1. 总结

在本次的课题实践中，多级反馈队列调度算法是一种综合性特别强的算法，它不必事先知道各进程所需的执行时间，而且还可以满足各种类型进程的需要。它是目前被公认的种较好的进程调度算法。

在编写程序之前我先模拟了一下多级反馈队列调度算法的运行过程，并选择使用结构体来存储进程信息，使用STL保存进程串和各级就绪队列。由于此算法还运用到了先来先服务算法和时间片轮转算法，我一时间无法把握程序的切入口，我参考了网络上的流程，绘制了自己的流程图。但是我发现网上的图和内容都有些错误或者缺漏或是和我最初的想法不是很吻合，特别是在有新进程到来发生抢占的时候。为了不让网络上的内容影响我最初的想法，我将程序段中的框架编辑好后，主要操作的部分按照自己绘制的图进行模拟。好在大学前两年学习算法过程中提高的代码能力，让我在实现起这个算法时并没有很困难。虽然中间运行过程中发生了运行无法结束的情况，后来在我一步步调试后发现是因为没有对queue判空就对其的front进行取值操作，if的分类讨论不够仔细。在测试过程中我还发现计算出的进程结束时间竟是负数，后来发现是因为我设置的保存进程信息的vector是全局变量，每次计算完我忘记clear了。最开始我的输出只设置了测试结果输出进程信息，后来觉得有必要把进程调度过程的运行过程也一起输出，每次进程几运行了几个时间单位，这样有利于用户测试和理解。

。。。。。。