**操作系统上机报告**

****

**班级： 191164**

**学号： 20161001514**

**姓名： 杨家印**

**日期： 2018/6/10**

实验一 多级队列调度算法

1.1 实习题目

用多级队列调度算法处理多个就绪队列。

1.2 程序功能

分别利用轮转法和短进程优先调度算法处理对应的进程队列，并输出各进程运行的时间。

1.3 设计思路

轮转法：给CPU设置时间片，CPU按进程排列顺序依次用一个时间片的时间片的时间来处理每个进程，若一个时间片内该进程未被处理完，则将剩余部分保存在队列尾部；若一个时间片内该进程被处理完，则保存该进程的运行时间=前几个未单次处理完毕的进程数\*时间片+该进程须运行时间，将该进程的须运行时间改为0。进程处理循环过程中若遇须运行时间=0的进程，跳过。

短进程优先调度算法：进程按须运行时间来划分优先级，时间短的优先级高，优先级高的进程先被处理。

1.4数据结构

struct LinkNode{ //定义链表结点

string name; //进程名

int need; //需运行时间

int turn; //周转时间

struct LinkNode \*link;

};

1.5算法设计

从文档中导入数据进数组，再由数组建立RQ1、RQ2链表

轮转法

p=RQ1; //利用轮转法处理RQ1

int a=0;

while(p!=NULL){

while(p!=NULL&&p->need>0){ //RQ1未处理完毕

q=p;

while(q->link!=NULL)q=q->link; //处理当前进程

if(p->need>pie){ //进程需求时间大于时间片

clock+=pie; //时钟增加一个时间片的时间

p->need-=pie; //该进程须运行时间减去一个时间片的时间

if(s!=NULL) s->link=p->link; //进程未运行完的部分存入表尾

q->link=p;

p=p->link;

q->link->link=NULL;

}

else{//进程需求时间小于时间片

a++;

clock+=p->need; //时钟增加该进程的需运行时间

p->need=0; //该进程处理完毕

p->turn+=clock; //进程周转时间加上之前的未单次处理完毕的进程数时间片及此次进程须运行时间

if(a==1) RQ1=p; //进程成为RQ1链表的头结点

s=p;

p=p->link; //处理下一进程

}

}

while(p!=NULL&&p->need==0){ //跳过已处理完毕的进程

p=p->link;

}

}

短进程优先调度算法

for(int b=0;b<5;b++){ //用短进程优先调度处理RQ2

q=RQ2;

p=(LinkNode\*)malloc(sizeof(LinkNode));

p->need=1000;

p->link=NULL;

while(q!=NULL){

if(q->need!=0&&q->need < p->need) p=q;

q=q->link;

}

clock+=p->need;

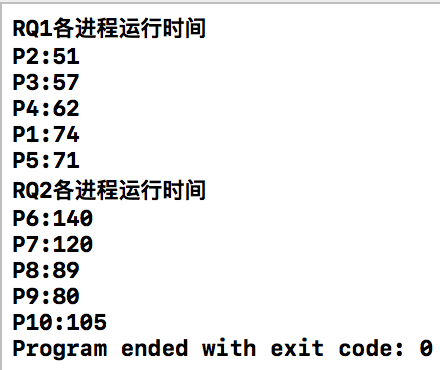
p->turn+=clock;

p->need=0; }

1.6程序运行情况

**测试数据如下：RQ1: P1-P5, RQ2: P6-P10**

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **进程** | **P1** | **P2** | **P3** | **P4** | **P5** | **P6** | **P7** | **P8** | **P9** | **P10** |
| **运行时间** | **16** | **11** | **14** | **13** | **15** | **21** | **18** | **10** | **7** | **14** |
| **已等待时间** | **6** | **5** | **4** | **3** | **2** | **1** | **2** | **3** | **4** | **5** |



1.7实习心得

进程 P1 P2 P3 P4 P5

运行时间 16 11 14 13 15

已等待时间 6 5 4 3 2

困难：将文件中的数据导入链表中，由于数据是横向的，不能直接利用循环将数据导入链表，于是先将数据从文件导入二维数组中。

实验心得：对于实际应用的程序，一次又一次的编译调试是不可避免的，每一次都要能够发现存在的问题并加以改正。

实验二 银行家算法

2.1实习题目

利用银行家算法（预见下一步进程会发生的情况）来避免死锁的发生。

2.2程序功能

根据输入的各进程资源总需求量、已分配资源、系统总资源，来判断某一进程接下来请求资源的操作是否会产生死锁，产生，撤回操作；不产生，给予资源并输出安全操作的进程序列。

2.3设计思路

一进程请求资源时，先判断是否会使系统产生死锁（所有进程能否安全执行），不产生，修改系统的各参数；产生，撤回刚才的资源。

2.4算法设计

数据结构为数组操作。

判断进程请求资源的操作是否产生死锁

int chkerr(){ //当有请求时，判断是否会使系统产生死锁，若否，输出当前的安全序列

int Finish[M] = { 0 }; //Finish为一维状态矩阵，对应的进程若成功执行释放资源后，标记为1

int Result[M] = { 0 }; //Result记录安全序列

int Work[N]; //Work为系统当前资源矩阵的镜像

int sum = 0; //sum为系统中已经安全执行并释放资源的进程数

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

Work[i] = Available[i];

while (true){

int flag1, flag2=0; //flag1判定系统当前资源量能否满足某进程还需要的资源量

//flag2判定当前系统是否还有进程能安全执行并释放资源

for (int i = 0; i < M; i++){

flag1 = 1;

if (Finish[i]== 0){

for (int j = 0; j < N; j++){

if (Need[i][j]>Work[j]) flag1 = 0;

}

if (flag1){ //若某进程还需要的资源能安全得到

flag2 = 1;

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

Work[j] = Work[j] + Allocation[i][j]; //进程执行完毕，释放该进程得到的资源

Finish[i] = 1; //状态矩阵记录当前进程已执行完毕释放

Result[sum] = i+1; //在安全序列中记录进程号

sum++; //完成的进程数增加1

}

}

}

if (sum == M){ //若所有的进程都安全执行完毕，则系统是安全的

cout << "当前系统的安全序列为:";

for (int i = 0; i <M; i++){

cout << Result[i]<<" ";

}

cout<<endl;

cout<<endl;

return 0;

}

if (!flag2){ //若进程执行完 ，且剩余的进程无法获得足够的资源，则说明系统是不安全的

return 1;

}

}

}

2.5程序运行情况

**测试数据：m=3:种类型的资源（A,B,C,） 进程个数n=5**

**Available=(2,3,3);**

|  |  |  |
| --- | --- | --- |
|  | **已分配资源数量** | **资源需求量** |
| **A B C** | **A B C** |
| **P1** | **2 1 2** | **3 4 7** |
| **P2** | **4 0 2** | **1 3 4** |
| **P3** | **3 0 5** | **0 0 3** |
| **P4** | **2 0 4** | **2 2 1** |
| **P5** | **3 1 4** | **1 1 0** |

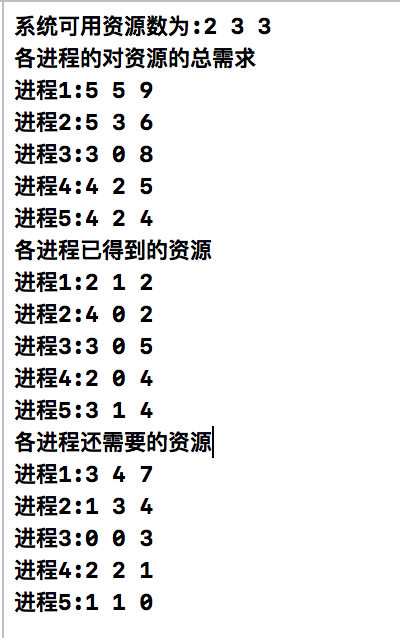
**请求序列如下：**

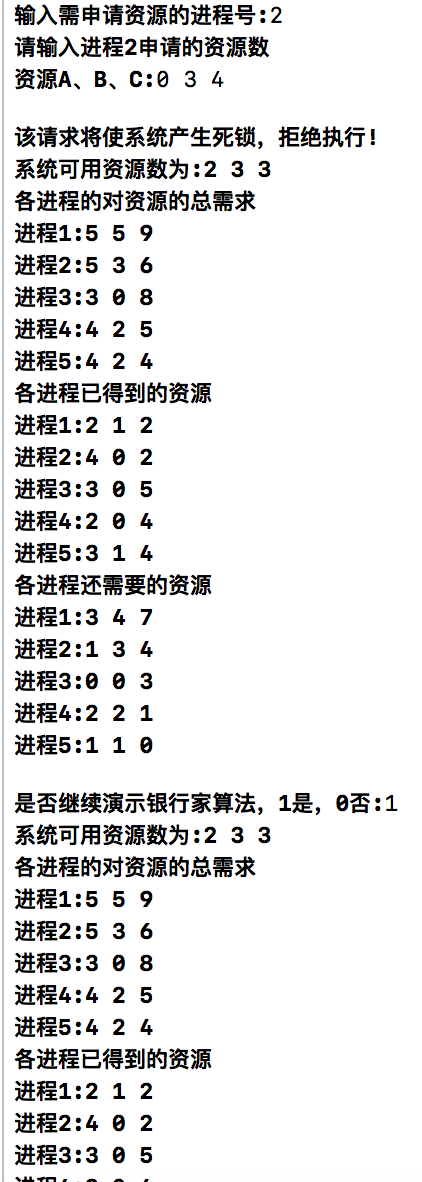
**a: 进程P2请求资源（0，3，4）**

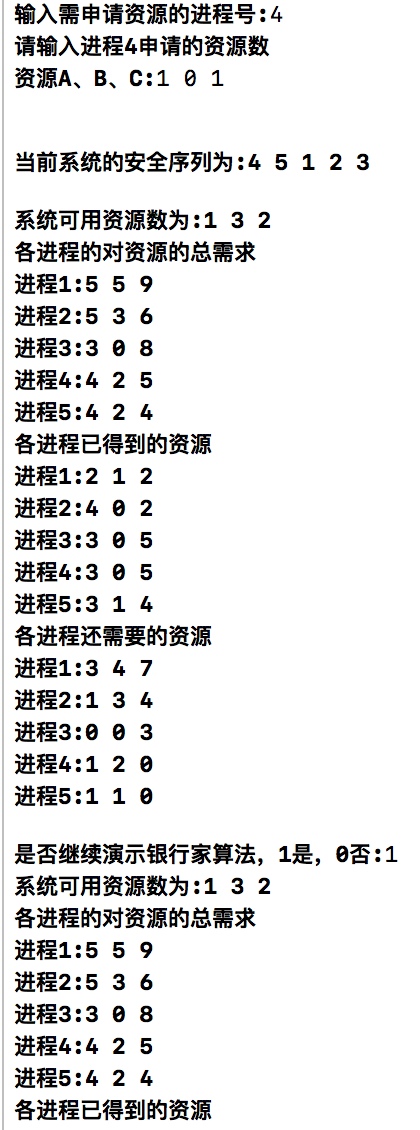
**b 进程P4请求资源（1，0，1）**

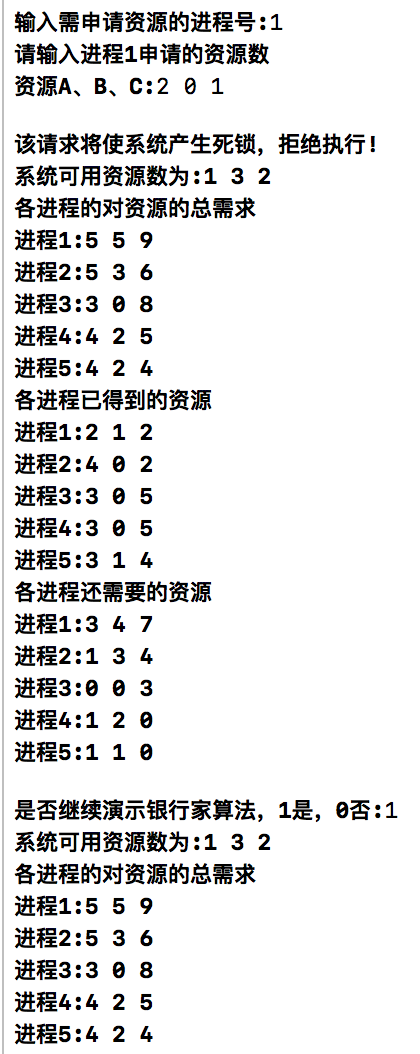
1. **进程P1请求资源（2，0，1）**

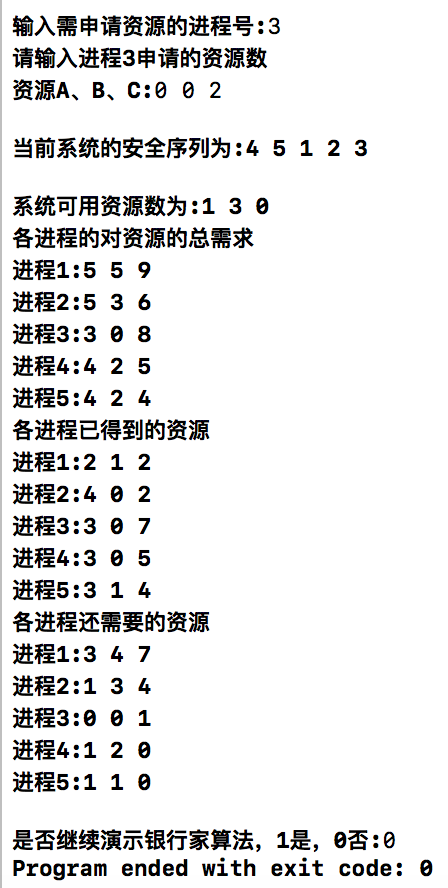
**d.进程P3请求资源（0，0，2）**











2.6实习心得

很实用的算法，代码不一定很长，功能良好的良好不在于代码的长度，而在于算法的精细，所以在面对一个实现功能的程序题时，不应该一上手就开始编码，而是要先想好数据结构及流程，这样才能事半功倍。

实验三 动态分区式存贮区管理

3.1实习题目

设计一个动态分区式存贮区管理程序，支持不同的放置策略。

3.2程序功能

进程可以进行申请内存、释放进程的操作，主界面可以查看当前系统内存分配情况，有循环处理用户请求的功能。

3.3设计思路

进程请求内存时，寻找内存空闲区，若空闲区富裕，则整体分配，否则，在释放内存的区域中需找空闲区，必要时应当切割空闲区。

进程释放内存时，判断进程是否有邻接的空闲区，若有则合并。

3.4数据结构

struct{ //已分配表

float address; //已分配区起始地址

float len; //已分配区长度

int flag; //已分配区表登记栏标志，"0"表示空栏目

}used[N];

struct{ //空闲区表

float address; //空闲区起始地址

float len; //空闲区长度

int flag; //空闲区表登记栏标志，用"0"表示空栏目，用"1"表示未分配

}frees[M];

3.5算法设计

int distribute(int name, float n\_len){ //动态分区

int i, k=-1; //k用于定位在空闲表中选择的未分配栏

float ads,len;

int count=0;

i=0;

while(i<=M-1){ //寻找最佳的空闲分区

if(frees[i].flag==1 && n\_len<=frees[i].len){

count++;

if(count==1||frees[i].len < frees[k].len)

k=i;

}

i=i+1;

}

if(k!=-1){

if((frees[k].len-n\_len)<=minsize) { //整体分配

frees[k].flag=0;

ads=frees[k].address;

len=frees[k].len;

}

else{ //切割空闲区

ads=frees[k].address;

len=n\_len;

frees[k].address+=n\_len;

frees[k].len-=n\_len;

}

i=0;

while(used[i].flag!=0){ //寻找内存分配表中标志为空的项

i=i+1;

}

if(i<=N-1){ //找到，在已分配区表中登记一个表项

used[i].address=ads;

used[i].len=len;

used[i].flag=name;

count1++;

}

else{ //已分配区表长度不足

if(frees[k].flag == 0){ //已做的整个分配撤销

frees[k].flag=1;

frees[k].address=ads;

frees[k].len=len;

}

else{ //将已做的切割分配撤销

frees[k].address=ads;

frees[k].len+=len;

}

cout<<"内存分配区已满，分配失败！\n";

return 0;

}

}

else{

cout <<"无法为该作业找到合适分区！\n";

return 0;

}

return name;

}

int release(int name) //释放函数

{

int y=0;

float ads, len;

int i,j,k; //j栏是下邻空闲区，k栏是上栏空闲区

int x;

while(y<=N-1&&used[y].flag!=name){

y=y+1;

}

if(y<=N-1){

ads=used[y].address;

len=used[y].len;

used[y].flag=0;

count2++;

}

else{

cout<<"该作业不存在！\n";

return 0;

}

j=k=-1;

i=0;

while(!(i>=M||(k!=-1&&j!=-1))){ //修改空闲分区表

if(frees[i].flag==1){

if((frees[i].address+frees[i].len)==ads)

k=i; //判断是否有上邻接

if((ads+len)==frees[i].address)

j=i; //判断是否有下邻接

}

i=i+1;

}

//合并空闲区

if(k!=-1) //回收区有上邻接

{

if(j!=-1){ //回收区也有下邻接，和上下邻接合并

frees[k].len+=frees[j].len+len;

frees[j].flag=0; //将第j栏的标记置为'0'

}

else //不存在下邻接，和上邻接合并

frees[k].len+=len;

}

else if(j!=-1){ //只有下邻接，和下邻接合并

frees[j].len+=len;

frees[j].address=ads;

}

else{ //上下邻接都没有

x=0;

while(frees[x].flag!=0)

x=x+1; //在空闲区表中查找一个状态为'0'的栏目

if(x<=M-1){ //找到后，在空闲分区中登记回收的内存

frees[x].address=ads;

frees[x].len=len;

frees[x].flag=1;

}

else{ //空闲表已满，执行回收失败

used[y].flag=name;

cout<<"空闲区已满，回收失败！\n";

return 0;

}

}

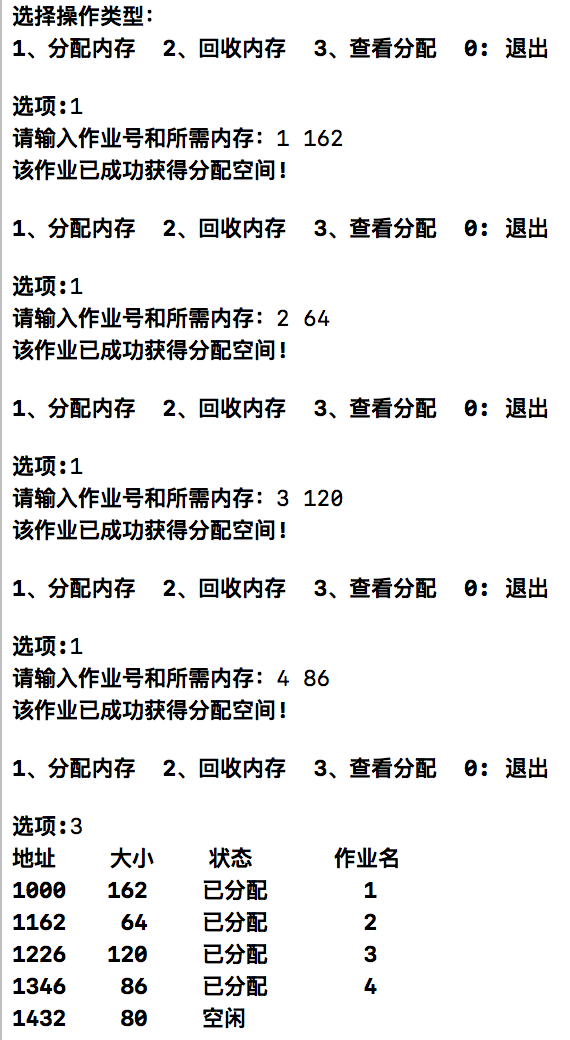
return name;

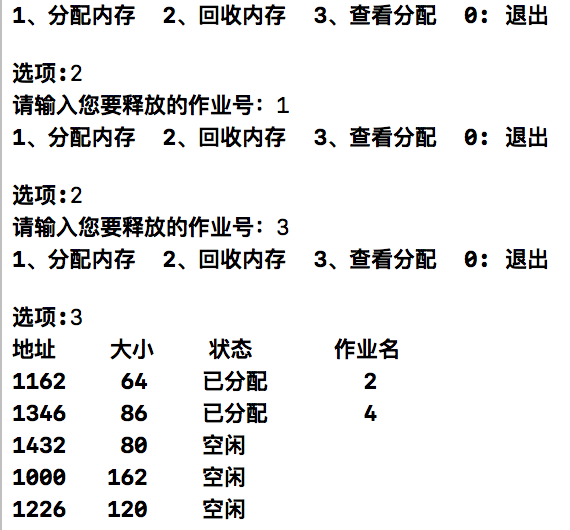
}

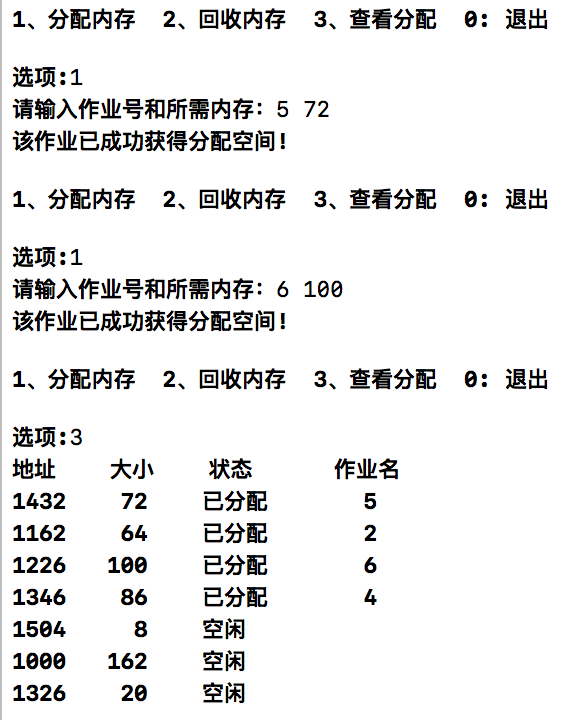
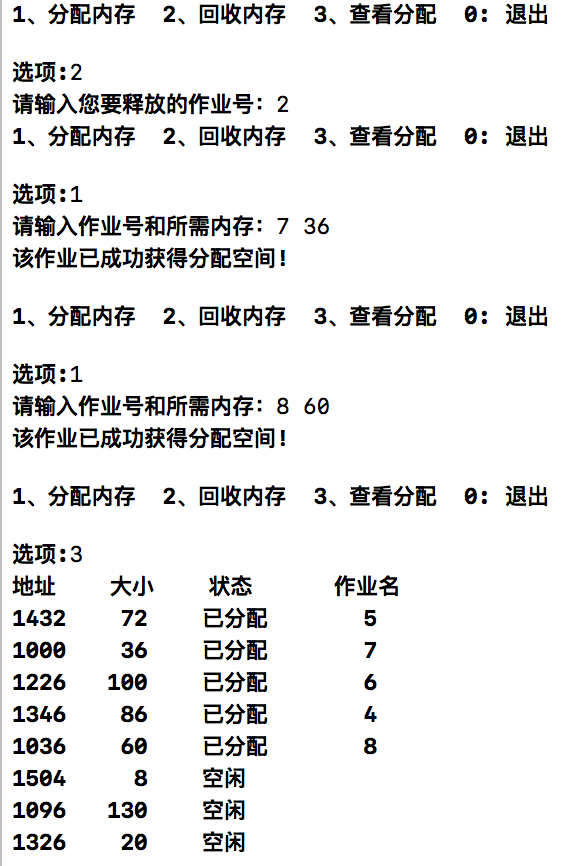
3.6程序运行情况

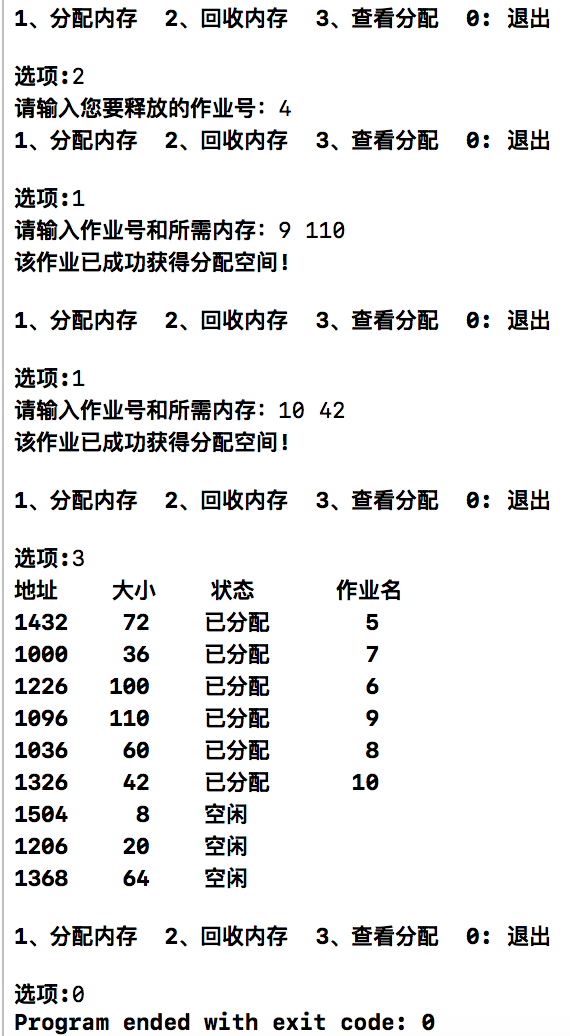
**数据实例：maxsize=512**

**J1申请162，J2申请64，J3申请120，J4申请86，J1完成，J3完成，J5申请72，J6申请100，J2完成，J7申请36，J8申请60，J4完成，J9申请110，J10申请42。**









3.7实习心得

困难：内存大小确定，开始程序中进程释放的内存难以被重复利用，尤其出现新进程申请的内存大于一个释放进程的内存。后来，通过对释放的空闲区进行合并操作以及切割空闲区，实现真正的动态分区。

实验心得：对于复杂的程序设计，应当在考虑完流程及数据结构后根据功能一步步的分析并完善程序，不应因为题目困难而苦恼，逐渐的思考中，解决问题的方法总会出现。