****

**中国地质大学(武汉)**

课程名称: 操作系统

教师姓名: 张求明

姓名: 蹇治鹏

学号: 20151001178

所在班级: 191152

**目录**：

[一、多级队列调度算法 3](#_Toc483139582)

[(一)、程序功能： 3](#_Toc483139583)

[(二)、设计思路 3](#_Toc483139584)

[(三)、数据结构 3](#_Toc483139585)

[1、PCB类(进程控制块) 3](#_Toc483139586)

[2、RQ类(调度队列) 3](#_Toc483139587)

[(四)、算法设计 3](#_Toc483139588)

[1、PCB类的构造 3](#_Toc483139589)

[2、RQ类的构造 4](#_Toc483139590)

[3、添加PCB到调度队列的尾部 4](#_Toc483139591)

[4、输出RQ队列 4](#_Toc483139592)

[5、轮转调度算法 5](#_Toc483139593)

[6、短进程优先调度 6](#_Toc483139594)

[(五)、程序运行情况 7](#_Toc483139595)

[1、测试数据 7](#_Toc483139596)

[2、测试结果 7](#_Toc483139597)

[二、银行家算法 7](#_Toc483139598)

[(一)、程序功能 7](#_Toc483139599)

[(二)、设计思路 7](#_Toc483139600)

[(三)、数据结构 8](#_Toc483139601)

[(四)、算法设计 8](#_Toc483139602)

[1、银行家算法流程图 8](#_Toc483139603)

[2、安全性算法 9](#_Toc483139604)

[(五)、程序运行情况 9](#_Toc483139605)

[1、测试数据 9](#_Toc483139606)

[2、测试结果 10](#_Toc483139607)

[三、动态分区式存贮区管理 10](#_Toc483139608)

[(一)、程序功能 10](#_Toc483139609)

[(二)、设计思路 10](#_Toc483139610)

[(三)、数据结构 10](#_Toc483139611)

[1、PCB类(进程控制器) 10](#_Toc483139612)

[2、block类(分区描述器) 11](#_Toc483139613)

[3、manage类 11](#_Toc483139614)

[(四)、算法设计 11](#_Toc483139615)

[1、首次适应算法 11](#_Toc483139616)

[2、最佳适应算法 12](#_Toc483139617)

[3、最坏适应算法 14](#_Toc483139618)

[4、内存回收 14](#_Toc483139619)

[(五)、程序运行情况 16](#_Toc483139620)

[1、测试数据： 16](#_Toc483139621)

[2、测试结果： 16](#_Toc483139622)

[(1)、首次适应算法 16](#_Toc483139623)

[(2)、最佳适应算法 17](#_Toc483139624)

[(3)、最差适应算法 17](#_Toc483139625)

1. 多级队列调度算法

(一)、程序功能：

1、进程高度算法的原理及物理变化模拟

2、实现简单轮转调度算法和短进程优先调度算法

(二)、设计思路：

1、建立并初始化队列，两种算法均以链表作为调度队列，每种算法只需要一个队列就能满足。

2、第一种算法的思想是采用的轮转调度算法，第二种采用的是短进程优先调度算法，确定好算法步骤，进入输入测试。

3、RQ1队列采用轮转调度算法，时间为7；RQ2采用短进程优先调度算法。并且RQ1>RQ2;

(三)、数据结构

1、PCB类(进程控制块)：

|  |  |  |  |
| --- | --- | --- | --- |
| 私有数据成员 | | | |
| 数据类型 | 名称 | | 注释 |
| string | m\_name | | 进程名 |
| int | m\_need | | 需要运行的时间 |
| int | m\_turn | | 进程周转时间 |
| PCB\* | m\_next | | 下一个进程的地址 |
| 函数声明 | | | |
| 声明 | | 用途 | |
| PCB(string name……); | | 构造函数 | |

注：PCB类是RQ类的友元类。

### 2、RQ类(调度队列)：

|  |  |  |  |
| --- | --- | --- | --- |
| 私有数据成员 | | | |
| 数据类型 | 名称 | | 注释 |
| PCB\* | first | | 队列的头指针 |
| 函数声明 | | | |
| 声明 | | 用途 | |
| RQ(string name……) | | 构造函数 | |
| void set(string name……) | | 队列末尾添加一个PCB | |
| void show() | | 输出整个队列到屏幕 | |
| void RoundRobin(const int&) | | 轮转法，参数为时间片的大小 | |
| void Priority() | | 短进程优先调度算法 | |

## (四)、算法设计

### 1、PCB类的构造

对PCB中的每个私有数据成员使用初始化列表进行初始化，其中PCB\* next初始化为nullptr；

代码：

//PCB构造函数

PCB::PCB(string name = "无", int need = 0, int turn = 0, PCB \*next = nullptr) {

m\_name = name, m\_need = need, m\_turn = turn, m\_next = next;

}

### 2、RQ类的构造

创建一个调度队列时，对RQ中的first指针进行空间分配，如果未对first给出指定值时，使用PCB构造函数中的初始值。

代码：

//RQ构造函数

RQ::RQ(string name = "无", int need = 0, int turn = 0, int clock = 0) {

first = new PCB(name, need, turn, nullptr);

m\_clock = clock;

}

### 3、添加PCB到调度队列的尾部

当调度的进程大于1时，在调度算法运行前，需要将所有进程添加到调度队列中，具体操作是：每次添加时，对队列末尾的PCB中next指针分配空间，并赋值创建一个调度队列时，对RQ中的first指针进行空间分配，如果未对first给出指定值时，使用PCB构造函数中的初始值。

代码：

//添加一个PCB到末尾

void RQ::set(string name, int need, int turn) {

PCB \*temp(first);

while (temp->m\_next != nullptr)

temp = temp->m\_next;

temp->m\_next = new PCB(name, need, turn, nullptr);

}

### 4、输出RQ队列

进程全部运行结束后，输出整个队列到屏幕，在内部设立一个临时指针PCB\* p;初始值为first，如果p指针不为nullptr，输出p中的相关数值，并且p指针后移；否则结束循环。创建一个调度队列时，对RQ中的first指针进行空间分配，如果未对first给出指定值时，使用PCB构造函数中的初始值。

代码：

//显示整个队列数据

void RQ::show() {

cout << "输出各进程的周转时间如下" << endl;

PCB \*p(first);

cout.setf(ios::left);

while (p != nullptr) {

cout << "进程名:" << setw(5) << p->m\_name << "所用时间:" << setw(5) << p->m\_turn << endl;

p = p->m\_next;

}

}

### 5、轮转调度算法

将CPU时间划分为若干时间片，并将这些时间片轮流分配给各个就绪进程，如果时间片结束时，进程未结束，则CPU被剥夺，并分配给下一个就绪进程使用，将该进程移动到就绪队列末尾；如果该进程在时间片内结束，则CPU将该进程从队列中移除，并运行后一个进程。

创建一个调度队列时，对RQ中的first指针进行空间分配，如果未对first给出指定值时，使用PCB构造函数中的初始值。

代码：

//轮转法

void RQ::RoundRobin(const int&piece\_time) {

PCB \*current(first), \*end(first), \*s(nullptr); int tag(0);

//将end指针移动到队列末尾

while (end->m\_next != nullptr)

end = end->m\_next;

while (current != nullptr) {

//当前PCB所需需时间大于时间片

if (current->m\_need>piece\_time) {

m\_clock += piece\_time; //时钟加上时间片

current->m\_need -= piece\_time; //当前指针的need更改

if (current != end) {

end->m\_next = current; //当前进程移动到队列末尾，进行等待

current = current->m\_next; //下一个进程准备运行

end = end->m\_next; //end指针后移

end->m\_next = nullptr; //断开末尾指针的连接

}

}

//当前PCB所需需时间不大于时间片

else {

tag++;

m\_clock += current->m\_need; //时钟加上时间片

current->m\_need = 0; //当前指针的need更改

current->m\_turn += m\_clock; //周转时间更改为当前的时钟时间

if (tag == 1) { //第一个结束的进程作为已结束进程序列的首进程

s = current;

first = s;

current = current->m\_next;

s->m\_next = nullptr;

}

else {

s->m\_next = current; //将结束进程添加到已结束进程的队列中

current = current->m\_next; //下一个进程准备运行

s = s->m\_next;

s->m\_next = nullptr;

}

}

}

}

### 6、短进程优先调度

每个进程按照执行时间长短，被赋予一个优先级，进程越短优先级越高，进程越长，优先级越低。优先级越高的就绪进程先被运行。为防止高优先级的进程无休止的运行下去，调度程序可以在每个时钟中断时适当降低当前进程的优先级，如果这是运行进程优先级低于次高优先级进程，则进程切换。

创建一个调度队列时，对RQ中的first指针进行空间分配，如果未对first给出指定值时，使用PCB构造函数中的初始值。

代码：

//RQ2采用短进程优先调度算法

void RQ::Priority() {

PCB \*min(first), \*current(first);

for (int i = 0; i<5; i++) {

//从队列中选择出最短进程

while (current != nullptr) {

if (current->m\_need != 0 && current->m\_need<min->m\_need)

min = current;

current = current->m\_next;

}

//更改时钟/周转时间/need

m\_clock += min->m\_need;

min->m\_turn += m\_clock;

min->m\_need = 0;

//指针复位，准备挑选下一个进程

current = min = first;

}

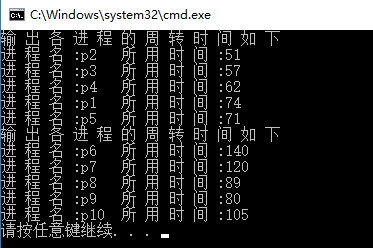
}

## (五)、程序运行情况

### 1、测试数据

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| RQ1采用轮转调度算法，时间片为7；RQ2采用短进程优先调度算法  队列：RQ1: P1—P5 RQ2: P6—P10 | | | | | | | | | | |
| 进程名 | 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 |

### 2、测试结果



1. 银行家算法

(一)、程序功能：实现银行家算法

(二)、设计思路

设request：是Pi进程的请求向量，当Pi发了资源请求后，系统按下述步骤检查：

(1)如果Request[i]<= Need[i]，则转向步骤(2)

(2)若Request[i] <=Available，则转向步骤(3)；

(3)系统试探性地把要求的资源分配给进程Pi，并修改以下数据结构的值：

Available=Available-Request[i]；

Allocation[i]= Allocation[i]+ Request[i]；

Need[i]= Need[i]- Request[i]；

(4)系统执行安全性算法，检查此次资源分配后，系统是否处于安全状态，若安全，才正式将资源分配给Pi进程，完成本次分配；否则，试探性分配作废，恢复原来的资源分配状态，Pi进程进入等待状态。

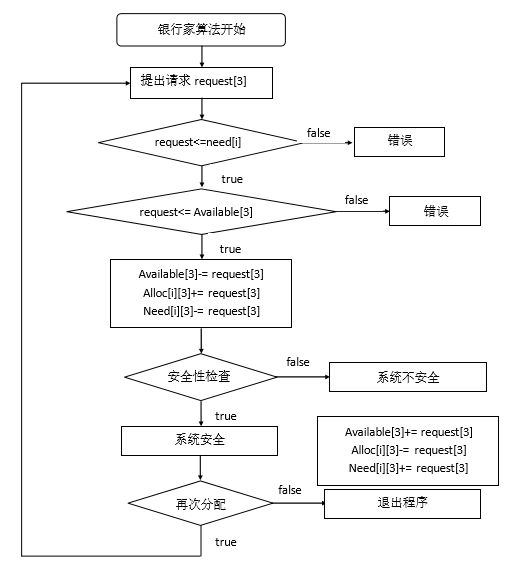
(三)、数据结构

|  |  |  |
| --- | --- | --- |
| 所用的的矩阵 | | |
| 类型 | 名字 | 注释 |
| int | Available[3] | 可利用资源量 |
| int | Alloc[5][3] | 分配矩阵 |
| int | Need[5][3] | 需求矩阵 |
| int | request[3] | 请求向量 |

|  |  |
| --- | --- |
| 相关函数 | |
| 声明 | 注释 |
| bool compare(int \*a, int \*b… …) | 判断a、b向量的大小 |
| bool SecurityCheck(int \*Available… …) | 安全性检查 |
| int main() | 主函数 |

(四)、算法设计

1、银行家算法流程图



2、安全性算法

(1) 设置两个工作向量：Work= Available;finish；

(2) 从进程集合中找到一个满足下述条件的进程：

finish ==false、need[i]<=Work;

如找到，执行(3) ； 否则，执行(4)

(3) 设进程获得资源，可顺利执行，直至完成，从而释放资源。

Work+= Alloc[i]、Finish=true、GOTO (2)

(4) 如所有的进程Finish= true ，则表 示安全；否则系统不安全。

代码：

//安全性检查

bool SecurityCheck(int \*Available, int Need[5][3], int Alloc[5][3])

{

bool finish[5] = { false,false,false,false,false };

int Work[3], i(0);

for (; i<3; i++)

Work[i] = Available[i]; //Work向量初始化

for (i = 0; i < 3; i++)

//寻找need<=Work并且未运行的进程

if (finish[i] == false&&compare(Need[i], Work, 3)) {

for (int j = 0; j < 3; j++)

Work[j] += Alloc[i][j]; //资源回收

finish[i] = true;

i = 0;

}

for (i = 0; i < 5; i++)

if (finish[i] == false) //存在不满足的进程

return false;

return true; //满足所有进程

}

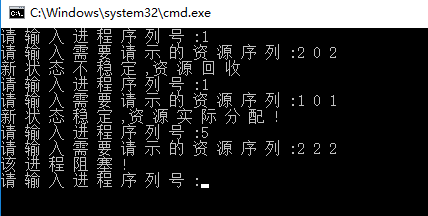
## (五)、程序运行情况

### 1、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);

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 进程名 | 已分配资源 | | | 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 |

### 2、测试结果



# 三、动态分区式存贮区管理

## (一)、程序功能：模拟动态分区式存贮区管理

## (二)、设计思路：

1、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。如首次、最佳、最坏。

2、分区描述器中为 ：flag/size/next;

3、自由主存队列按链表组织，主存大小假设为maxsize（单位为节=rd的大小）。

4、作业申请n节,实际分配的分区大小应为n+1节。 其中一节作为分区描述器，其他n节提供给作业。

5、已分配区放在高地址处。

6、合并时应考虑四种情况：

假设回收区为r,上邻为f1（f1需搜索自由主存队列）,下邻为f2（f2可直接计算）

A) f1空闲，f2已分配；

B) f1已分配，f2空闲；

C) f1空闲，f2空闲；

D) f1已分配，f2已分配；

## (三)、数据结构

### 1、PCB类(进程控制器)：

|  |  |  |  |
| --- | --- | --- | --- |
| 私有数据成员 | | | |
| 类型 | 名称 | | 注释 |
| int | begin | | 进程开始的首地址 |
| int | size | | 进程占用的大小 |
| string | name | | 进程名 |
| 函数 | | | |
| 声明 | | 注释 | |
| PCB(string n="无'",int s=0,int b=0) | | 构造函数 | |
| ostream& operator<<(ostream&os, PCB&P) | | 重定义输出函数 | |

注：PCB类是manage类的友元类。

### 2、block类(分区描述器)：

|  |  |  |  |
| --- | --- | --- | --- |
| 私有数据成员 | | | |
| 类型 | 名称 | | 注释 |
| int | m\_size | | 分区的大小 |
| bool | m\_falg | | 判断该分区是否被占用 |
| block\* | next | | 下一个分区的地址 |
| 函数 | | | |
| 函数声明 | | 注释 | |
| block(bool flag = false… …) | | 构造函数 | |
| ostream& operator<<(ostream&os, block&P) | | 重定义输出函数 | |

注：block类是manage类的友元类。

### 3、manage类

|  |  |  |  |
| --- | --- | --- | --- |
| 私有数据成员 | | | |
| 类型 | 名称 | | 注释 |
| block\* | first; | | 分区链表首地址 |
| vector<PCB> | lib; | | 未结束的所有进程 |
| 函数 | | | |
| 声明 | | 注释 | |
| manage() | | 构造函数 | |
| void first\_time(); | | 首次适应算法 | |
| void best(); | | 最佳适应算法 | |
| void worst(); | | 最坏适应算法 | |
| void free(); | | 内存回收 | |
| void show(); | | 显示内存分配情况 | |

## (四)、算法设计

### 1、首次适应算法

首次适应算法是将程序放入到主存中，按地址查找到第一个能装入它的空闲区。在首次适应算法中，空闲区是按其位置的顺序链接在一起，即每一个后继空闲去的起始地址总比他前者大。当要分配一个分区时，总是从低地址空闲区开始查寻，直到找到第一个足以满足该程序要求的空闲区为止。

代码：

void manage::first\_time() {

cout << "输入进程名和大小:";

string name; int size,addr(0);

cin >> name >> size;

block\* p(first);

while (p != nullptr) { //寻找块

addr +=(p->m\_size+1); //计算地址

if (p->m\_flag == false) { //当前块未被占用

if (p->m\_size > size) { //当前块的大小大于进程申请的大小

p->m\_size -= (size + 1); //从当前块中减去进程申请的大小

block\* temp=new block(true,size,p->m\_next); //新建一个进程块

p->m\_next = temp; //添加到当前块的后面

addr -= (size + 1); //重新计算当前块的地址

break;

}

else if (p->m\_size == size) { //当前块的大小刚好够进程申请的大小

p->m\_flag = true; //直接更改当前块的状态为已占用，不用分裂块

addr -= (size + 1); //重新计算地址

break;

}

else p = p->m\_next; //当前块的大小不满足进程申请的大小，先后继续寻找。

}

else p = p->m\_next; //当前块已被占用，先后继续寻找

}

if (p == nullptr)cout << "分配失败" << endl; //未找到符合条件的块，分配失败

else { //找到符合条件的块，该进程存入lib进程库中

PCB temp(name, size, addr);

lib.push\_back(temp);

}

}

### 2、最佳适应算法

最佳适应算法是将程序放入主存中与他所需大小最接近的空闲区中，在最佳适应算法中，空闲区队列是按空闲区大小递增顺序链接在一起的。在进行分配时总是从最小的空闲区开始查寻，因而找到的第一个能满足要求的空闲区便是最佳的一个。即从所要求的大小来看，该区和气候的所有空闲区相比它是最接近的。  
代码：

void manage::best() {

cout << "输入进程名和大小:";

string name; int size, addr1(0),addr2(0);

cin >> name >> size;

block \*p(first),\*q(nullptr);

int Error(512);

while (p != nullptr) {

addr1 += (p->m\_size + 1);

if (p->m\_flag == false) { //当前块未被占用

if (p->m\_size > size) { //当前块的大小满足进程请求的大小

if (p->m\_size - size < Error) { //当前块更合适该进程

Error = p->m\_size - size; //计算偏差

q = p; //保存当前块的地址

addr2 = addr1 - size - 1;

p = p->m\_next; //指针后移

}

else p = p->m\_next; //和之前的块相比，之前的块更合适该进程

}

else if (p->m\_size == size) { //当前块的大小刚好等于进程申请的大小

q = p; //已找到最合适的块，保存当前块的信息并退出

addr2 = addr1 - size - 1;

Error = 0;

break;

}

else p = p->m\_next; //当前块的大小小于进程请求的大小，继续先后搜索

}

else p = p->m\_next; //当前块已经被占用，继续先后搜索

}

if (Error == 512)cout << "分配失败" << endl;

else {

if (Error == 0)q->m\_flag = true;

else {

q->m\_size -= (size + 1); //从当前块中减去进程申请的大小

block\* temp = new block(true, size, q->m\_next); //新建一个进程块

q->m\_next = temp; //添加到当前块的后面

}

PCB temp(name, size, addr2);

lib.push\_back(temp);

}

}

### 3、最坏适应算法

最坏适应算法就是将程序放入主存中最不适合它的空闲区，即最大的空闲区内。在最坏适应算法中，空闲区是按大小递减顺序连接到一起的，因此，其队列指针总是指向最大空闲区，在进行分配时，总是从最大空闲区开始查寻。

代码：

void manage::worst() {

cout << "输入进程名和大小:";

string name; int size, addr1(0), addr2(0);

cin >> name >> size;

block \*p(first), \*q(nullptr);

int Error(0);

while (p != nullptr) {

addr1 += (p->m\_size + 1);

if (p->m\_flag == false) { //当前块未被占用

if (p->m\_size >= size) { //当前块的大小满足进程请求的大小

if (p->m\_size - size >= Error) {//当前块比之前块更大

Error = p->m\_size - size; //计算偏差

q = p; //保存当前块的地址

addr2 = addr1 - size - 1;

p = p->m\_next; //指针后移

}

else p = p->m\_next; //和之前的块相比，之前的块更合适该进程

}

else p = p->m\_next; //当前块的大小小于进程请求的大小，继续先后搜索

}

else p = p->m\_next; //当前块已经被占用，继续先后搜索

}

if (Error == 0)cout << "分配失败" << endl;

else {

q->m\_size -= (size + 1); //从当前块中减去进程申请的大小

block\* temp = new block(true, size, q->m\_next); //新建一个进程块

q->m\_next = temp; //添加到当前块的后面

PCB temp1(name, size, addr2);

lib.push\_back(temp1);

}

}

### 4、内存回收

在内存回收时应考虑四种情况：

假设回收区为r,上邻为f1（f1需搜索自由主存队列）,下邻为f2（f2可直接计算）

A) f1空闲，f2已分配：r和f1合并

B) f1已分配，f2空闲：r和f2合并

C) f1空闲，f2空闲：r和f1、f2合并

D) f1已分配，f2已分配：r不和任何分区合并

代码：

void manage::free() {

cout << "输入回收进程的名称:";

string name;

cin >> name;

//在lib库中查找是否存在该进程

vector<PCB>::iterator it = lib.begin();

while (it != lib.end()){

if (it->name == name)

break;

else it++;

}

//该进程不存在，直接提示并退出函数

if (it == lib.end()) {

cout << "该进程不存在" << endl;

return;

}

//进程存在

else {

int addr(0); block \*p(first);

while (p != nullptr) { //查找该进程所占用的内存块

if (it->begin == addr)break; //查找到该进程

addr += (p->m\_size + 1);

p = p->m\_next;

}

if (p == nullptr)cout << "未在内存中查找到该进程" << endl;

else {

p->m\_flag = false; //该内存块的状态更改为未占用

//所回收的内存块后面存在空闲的内存块

if (p->m\_next != nullptr&&p->m\_next->m\_flag==false) {

p->m\_size += p->m\_next->m\_size;//该内存块的大小增加为为俩内存大小之和

block \*temp(p->m\_next);

p->m\_next = p->m\_next->m\_next; //更改指针指向

delete temp; //释放后一指针的空间

}

lib.erase(it); //进程库中删除当前进程

//所回收的内存块前面存在空闲的内存块

block \*q(first);

//寻找到所回收的内存块前面的内存块

while (q != nullptr) {

//所回收的内存块前面存在空闲的内存块

if (q->m\_next==p&&q->m\_flag == false) {

q->m\_size += (p->m\_size + 1);

q->m\_next = p->m\_next;

delete p;

break;

}

q = q->m\_next;

}

}

}

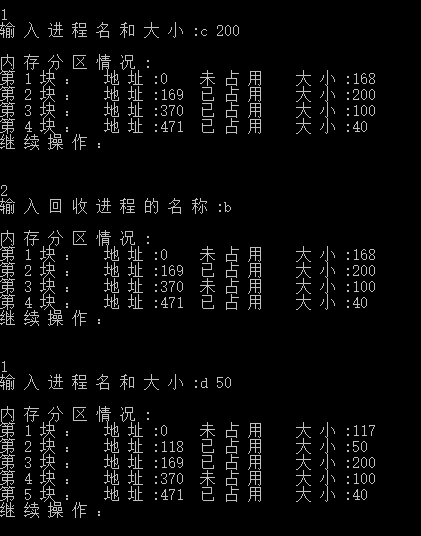
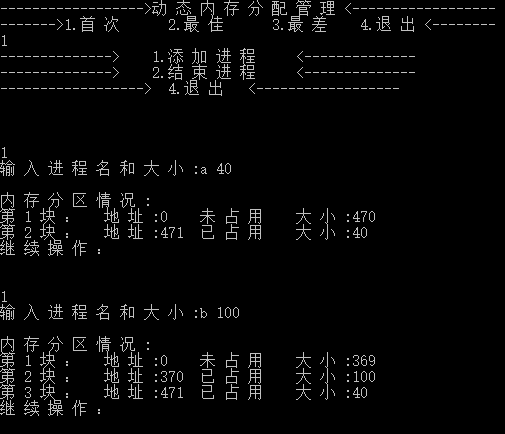
}

## (五)、程序运行情况

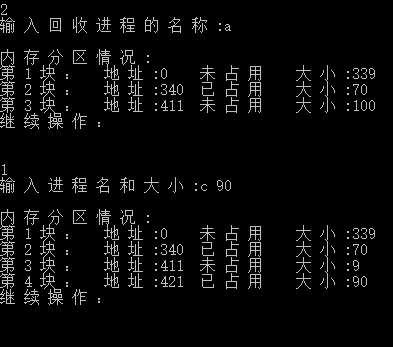
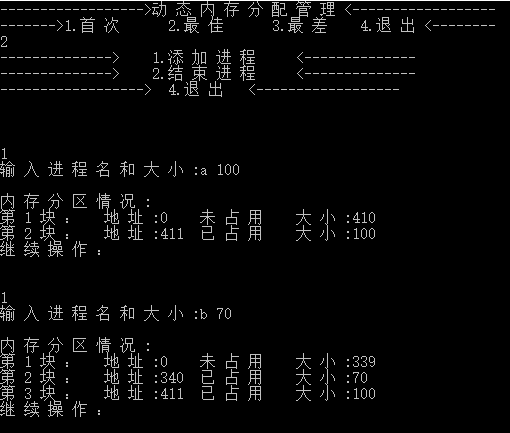
### 1、测试数据：初始值大小为512(分区描述器占一位，实际大小为511)；

### 2、测试结果：

#### (1)、首次适应算法



#### (2)、最佳适应算法



#### (3)、最差适应算法

