**武汉大学计算机学院**

**本科生课程设计报告**

**操作系统实验总体设计与实现**

专 业 名 称 ：计算机科学与技术

课 程 名 称 ：操作系统实验

指 导 教 师 ：宋伟 职称 副教授

学 生 学 号 ：2016301500017

学 生 姓 名 ：伍丹梅

二○一八年十一月

**郑 重 声 明**

本人呈交的设计报告，是在指导老师的指导下，独立进行实验工作所取得的成果，所有数据、图片资料真实可靠。尽我所知，除文中已经注明引用的内容外，本设计报告不包含他人享有著作权的内容。对本设计报告做出贡献的其他个人和集体，均已在文中以明确的方式标明。本设计报告的知识产权归属于培养单位。

本人签名： 伍丹梅 日期： 2018.11.30

摘 要

操作系统实验包括处理器调度，主存空间的分配和回收，磁盘存储空间的分配和回收，银行家算法，模拟页面地址重定位。

实验目的：

1. 处理器调度：模拟在单处理器环境下的处理器调度，加深了解处理器调度的工作。
2. 主存空间的分配和回收：通过本实习帮助理解在不同的存储管理方式下应怎样进行存储空间的分配和回收。
3. 磁盘存储空间的分配和回收：通过本实习使学生掌握磁盘存储空间的分配和回收算法。
4. 银行家算法：熟悉银行家算法，对预防死锁有更深刻的认识。
5. 模拟页面地址重定位：编写和调试模拟实现页式地址重定位；加深理解页式地址重定位技术在多道程序设计中的作用和意义。

实验设计主要遵循实验设计原理

实验内容主要包括：

1. 处理器调度：选择一个调度算法，实现处理器调度。
2. 主存空间的分配和回收：主存储器空间的分配和回收。
3. 磁盘存储空间的分配和回收：模拟磁盘空闲空间的表示方法，以及模拟实现磁盘空间的分配和回收。
4. 银行家算法：编写实现银行家算法，实现资源的安全分配。
5. 模拟页面地址重定位：设计页表结构；设计地址重定位算法

实验结论：

通过以上实验，我对操作系统工作原理有了更深的理解。

**关键词：**处理器调度，主存空间的分配和回收，磁盘存储空间的分配和回收，银行家算法，模拟页面地址重定位。

**目 录**

**1** **实验目的和意义**

1.1 实验目的 ………………………………………………………………………1

1.2 实验意义………………………………………………………………………..1

**2** **实验设计**

2.1 概述………………………………………………………………………………2

2.2 实验原理…………………………………………………………………………4

2.3 实验方案…………………………………………………………………………2

**结论** …………………………………………………………………………………33

**附录** …………………………………………………………………………………34

**1 实验目的和意义**

* 1. **实验目的**

1. 处理器调度：模拟在单处理器环境下的处理器调度，加深了解处理器调度的工作。
2. 主存空间的分配和回收：通过本实习帮助理解在不同的存储管理方式下应怎样进行存储空间的分配和回收。
3. 磁盘存储空间的分配和回收：通过本实习使学生掌握磁盘存储空间的分配和回收算法。
4. 银行家算法：熟悉银行家算法，对预防死锁有更深刻的认识。
5. 模拟页面地址重定位：编写和调试模拟实现页式地址重定位；加深理解页式地址重定位技术在多道程序设计中的作用和意义。

* 1. **实验意义**

1. 处理器调度：模拟在单处理器环境下的处理器调度，加深了解处理器调度的工作。
2. 主存空间的分配和回收：通过可变分区管理方式下采用首次适应算法实现主存分配和回收，更加了解主存空间分配。
3. 磁盘存储空间的分配和回收：怎样有效地管理磁盘存储空间是操作系统应解决的一个重要问题，通过本实验加深理解。
4. 银行家算法：了解死锁产生的条件和原因，并采用银行家算法有效得防止死锁的发生，以加深对课堂上所授知识的理解。
5. 模拟页面地址重定位：加深对操作系统页面地址重定位的理解。

**2实验设计**

* 1. **概述**

1. 处理器调度：设计一个按优先数调度算法实现处理器调度的程序。

（1）假定系统有5个进程，每个进程用一个PCB来代表。PCB的结构为：

* 进程名——如P1~P5。
* 指针——按优先数的大小把5个进程连成队列，用指针指出下一个进程PCB的首地址。
* 要求运行时间——假设进程需要运行的单位时间数。
* 优先数——赋予进程的优先数，调度时总是选取优先数大的进程先执行。
* 状态——假设两种状态：就绪和结束，用R表示就绪，用E表示结束。初始状态都为就绪状态。

（2）每次运行之前，为每个进程任意确定它的“优先数”和“要求运行时间”。

（3）处理器总是选择队首进程运行。采用动态改变优先数的办法，进程每运行1次，优先数减1，要求运行时间减1。

（4）进程运行一次后，若要求运行时间不等于0，则将它加入就绪队列，否则，将状态改为“结束”，退出队列。

1. 若就绪队列为空，结束，否则转到(3)重复。

要求能接受键盘输入的进程优先数及要求运行时间，能显示每次进程调度的情况，如哪个进程在运行，哪些进程就绪，就绪进程的排列情况。

1. 主存空间的分配和回收：可变分区管理方式下采用首次适应算法实现主存分配和回收

（1）可变分区方式是按作业需要的主存空间大小来分割分区的。当要装入一个作业时，根据作业需要的主存容量查看是否有足够的空闲空间，若有，则按需分配，否则，作业无法装入。假定内存大小为128K(可输入)，空闲区说明表格式为：

* 起始地址——指出空闲区的起始地址；
* 长度——一个连续空闲区的长度；
* 状态——有两种状态，一种是“未分配”状态；另一种是“空表目”状态，表示该表项目前没有使用。

（2）采用首次适应算法分配回收内存空间。运行时，输入一系列分配请求和回收请求。

要求能接受来自键盘的空间申请及释放请求，能显示分区分配及回收后的内存布局情况。

1. 磁盘存储空间的分配和回收：用位示图管理磁盘存储空间

（1）为了提高磁盘存储空间的利用率，可在磁盘上组织成链接文件、索引文件，这类文件可以把逻辑记录存放在不连续的存储空间。为了表示哪些磁盘空间已被占用，哪些磁盘空间是空闲的，可用位示图来指出。位示图由若干字节构成，每一位与磁盘上的一块对应，“1”状态表示相应块已占用，“0”状态表示该块为空闲。位示图的形式与实习二中的位示图一样，但要注意，对于主存储空间和磁盘存储空间应该用不同的位示图来管理，绝不可混用

（2）申请一块磁盘空间时，由分配程序查位示图，找出一个为“0”的位，计算出这一位对应块的磁盘物理地址，且把该位置成占用状态“1”。假设现在有一个盘组共8个柱面，每个柱面有2个磁道（盘面），每个磁道分成4个物理记录。那么，当在位示图中找到某一字节的某一位为“0”时，这个空闲块对应的磁盘物理地址为：

柱面号=字节号

磁道号= 位数 / 4

物理记录号= 位数 % 4

（3）归还一块磁盘空间时，由回收程序根据归还的磁盘物理地址计算出归还块在位示图中的对应位，把该位置成“0”。按照（2）中假设的盘组，归还块在位示图中的位置计算如下：

字节号=柱面号

位数=磁道号×4+物理记录号

1. 设计申请磁盘空间和归还磁盘空间的程序。

要求能接受来自键盘的空间申请及释放请求，要求能显示或打印程序运行前和运行后的位示图；分配时把分配到的磁盘空间的物理地址显示或打印出来，归还时把归还块对应于位示图的字节号和位数显示或打印出来。

1. 银行家算法

初始状态下，设置数据结构存储可利用资源向量（Available），最大需求矩阵（MAX），分配矩阵（Allocation），需求矩阵（Need），输入待分配进程队列和所需资源。

设计安全性算法，设置工作向量表示系统可提供进程继续运行的可利用资源数目。

如果进程队列可以顺利执行打印输出资源分配情况，如果进程队列不能顺利执行打印输出分配过程，提示出现死锁位置。

1. 模拟页面地址重定位：设计页表结构；设计地址重定位算法；有良好的人机对话界面。
   1. **实验原理**

2.2.1**处理器调度原理**

处理器总是选择队首进程运行。采用动态改变优先数的办法，进程每运行1次，优先数减1，要求运行时间减1。进程运行一次后，若要求运行时间不等于0，则将它加入就绪队列，否则，将状态改为“结束”，退出队列。如果队列为空，则继续。

2.2.2**主存空间的分配和回收原理**

可变分区方式是按作业需要的主存空间大小来分割分区的。当要装入一个作业时，根据作业需要的主存量查看是否有足够的空闲空间，若有，则按需要量分割一个分区分配给该作业；若无，则作业不能装入。随着作业的装入、撤离，主存空间被分成许多个分区，有的分区被作业占用，而有的分区是空闲的。

当进程运行完毕释放内存，系统根据回收区的首址，从空闲区链表中找到相应的插入点，此时可能出现以下4种情况之一：

1.回收区与插入点的前一个空闲分区F1相邻接，此时将两个分区合并

2.回收区与插入点的后一个空闲分区F2相邻接，此时将两个分区合并

3.回收区与插入点的前，后两个空闲分区相邻接，此时将三个分区合并

4.回收区既不与F1相邻接，又不与F2相邻接，此时应为回收区单独建立一个新表项

2.2.3**磁盘存储空间的分配和回收原理**

申请一块磁盘空间时，由分配程序查位示图，找出一个为“0”的位，计算出这一位对应块的磁盘物理地址，且把该位置成占用状态“1”。假设现在有一个盘组共8个柱面，每个柱面有2个磁道（盘面），每个磁道分成4个物理记录。那么，当在位示图中找到某一字节的某一位为“0”时，这个空闲块对应的磁盘物理地址为：

柱面号=字节号

磁道号= 位数 / 4

物理记录号= 位数 % 4

归还一块磁盘空间时，由回收程序根据归还的磁盘物理地址计算出归还块在位示图中的对应位，把该位置成“0”。按照（2）中假设的盘组，归还块在位示图中的位置计算如下：

字节号=柱面号

位数=磁道号×4+物理记录号

2.2.4**银行家算法原理**

　 银行家算法最初为银行系统设计，以确保银行在发放现金贷款时，不会发生不能满足所有客户需要的情况。在OS设计中，银行家算法可以用它来预防死锁。   
　 为实现银行家算法，每个新进程在进入系统时它必须申明在运行过程中，可能需要的每种资源类型的最大单元数目，其数目不应超过系统所拥有的资源总量。当某一进程请求时，系统会自动判断请求量是否小于进程最大所需，同时判断请求量是否小于当前系统资源剩余量。若两项均满足，则系统预先分配资源并执行安全性检查算法。如果通过安全性检验，那么分配成功，如果不通过，则认为该分配序列不安全，不给予分配。

安全性检查算法：

　 安全性检查算法用于检查系统进行资源分配后是否安全，若安全系统才可以执行此次分配；若不安全，则系统不执行此次分配。   
　　安全性检查算法原理为：在系统预先分配资源后，算法从现有进程列表寻找出一个可执行的进程进行执行，执行完成后回收进程占用资源；进而寻找下一个可执行进程。当进程需求量大于系统可分配量时，进程无法执行。当所有进程均可执行，则产生一个安全执行序列，系统资源分配成功。若进程无法全部执行，即无法找到一条安全序列，则说明系统在分配资源后会不安全，所以此次分配失败。

**2.2.5模拟页面地址重定位原理**

当进程在CPU上运行时，如指令中涉及逻辑地址时，操作系统自动根据页长得到页号和页内偏移，把页内偏移拷贝到物理地址寄存器，再根据页号，查页表，得到该页在内存中的块号，把块号左移页长的位数，写到物理地址寄存器。

* 1. **实验方案**

**2.3.1 处理器调度**

**数据结构：**

typedef struct PCB {//进程块

struct PCB \*next;//定义指向下一个节点的指针

char name[10];//定义进程名

int Required\_time;//定义需求运行时间

int Prio;//定义优先级

char state;//定义进程状态Ready/End

}\*Proc

**优先级调度算法：**

处理器总是选择队首进程运行。采用动态改变优先数的办法，进程每运行1次，优先数减1，要求运行时间减1。进程运行一次后，若要求运行时间不等于0，则将它加入就绪队列，否则，将状态改为“结束”，退出队列。若就绪队列为空，结束，否则重复。

**优先级调度算法流程图：**

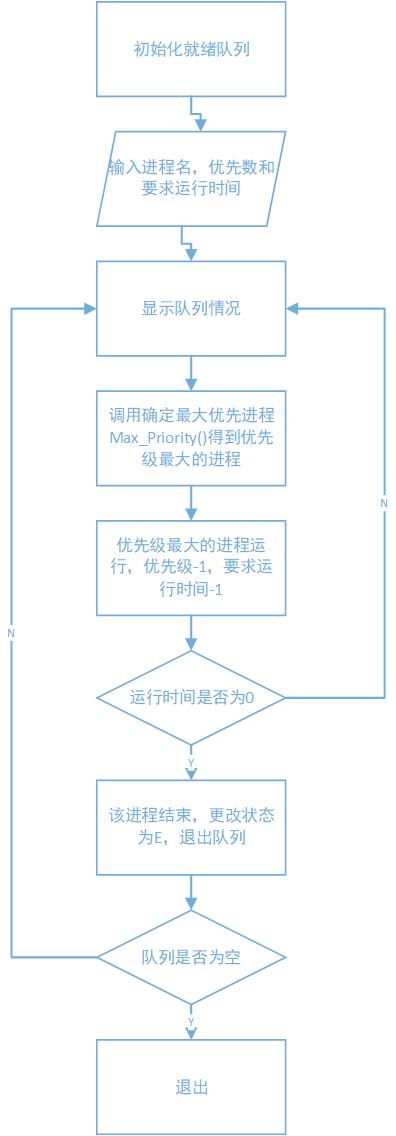
****

图2-1

**确定最大优先级算法流程图：**

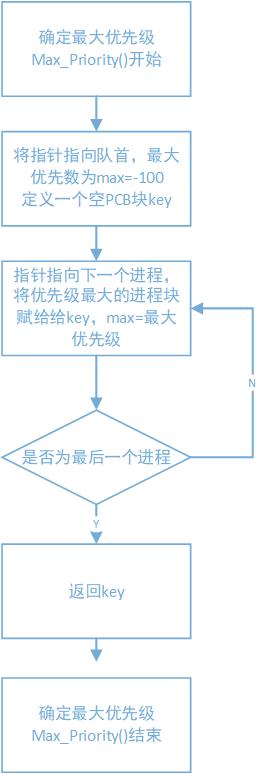
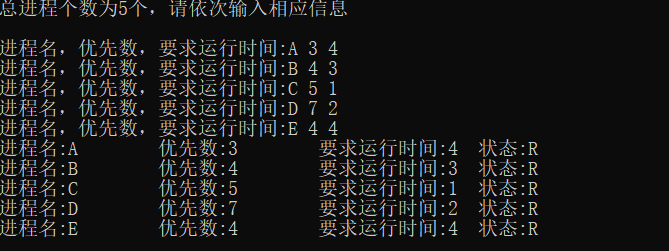
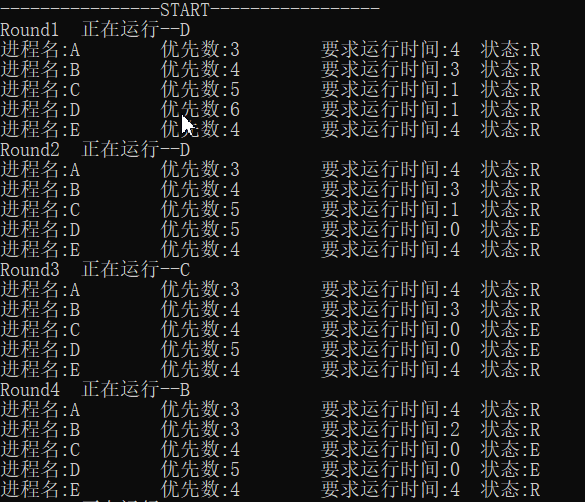
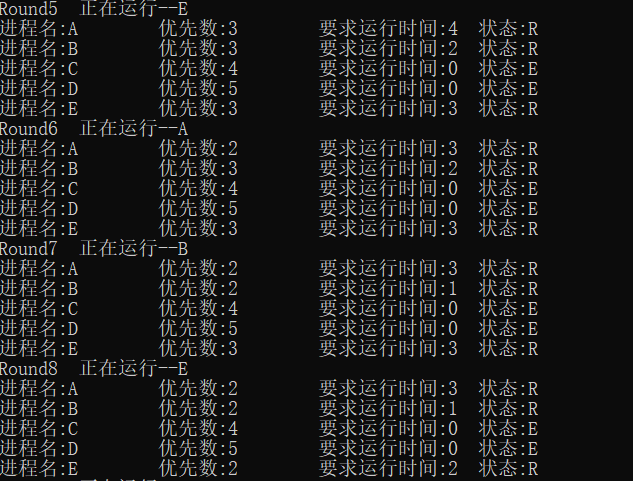
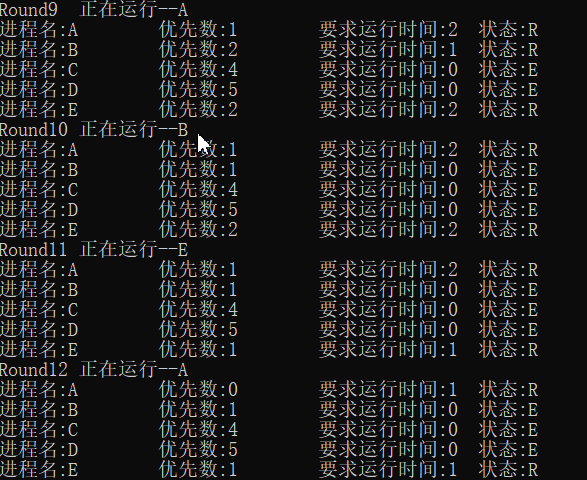
****

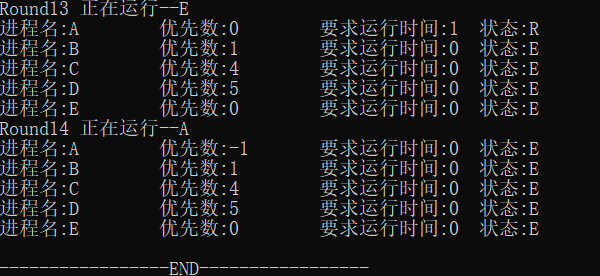
图2-2

**测试：**

输入进程名，优先数，要求运行时间：

可看到运行情况：





**2.3.2 主存空间的分配和回收**

**数据结构：**

typedef struct spare{

int start;//始址

int length;//长度

int state;//状态

struct spare \*next;

}SPARESPACE;

//未分配表unallocated，已分配表assigned

SPARESPACE \*unallocated = NULL, \*assigned = NULL;

**可变分区管理方式下采用首次适应算法：**

当要装入一个作业时，根据作业需要的主存量查看是否有足够的空闲空间，若有，则按需要量分割一个分区分配给该作业；若无，则作业不能装入。随着作业的装入、撤离，主存空间被分成许多个分区，有的分区被作业占用，而有的分区是空闲的。

当进程运行完毕释放内存，系统根据回收区的首址，从空闲区链表中找到相应的插入点，此时可能出现以下4种情况之一：

1.回收区与插入点的前一个空闲分区F1相邻接，此时将两个分区合并

2.回收区与插入点的后一个空闲分区F2相邻接，此时将两个分区合并

3.回收区与插入点的前，后两个空闲分区相邻接，此时将三个分区合并

4.回收区既不与F1相邻接，又不与F2相邻接，此时应为回收区单独建立一个新表项

**算法流程图：**

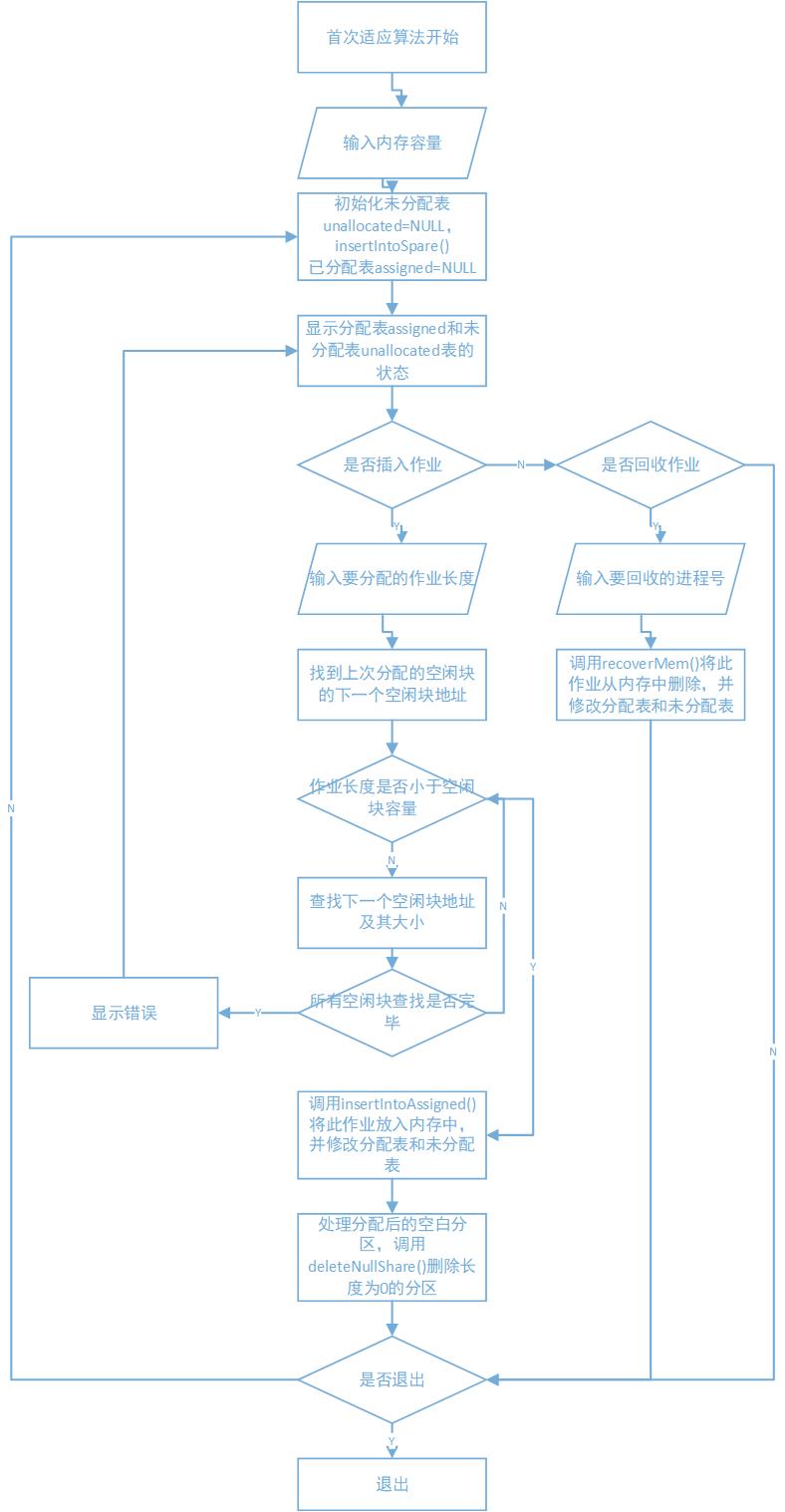
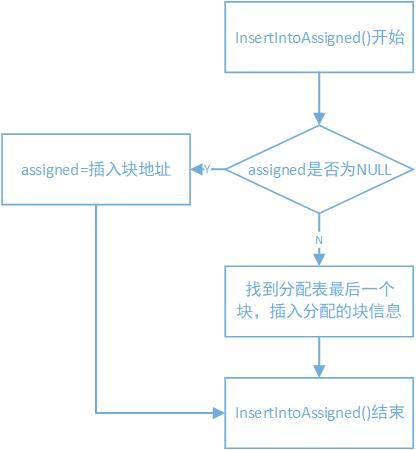


图2-3

****

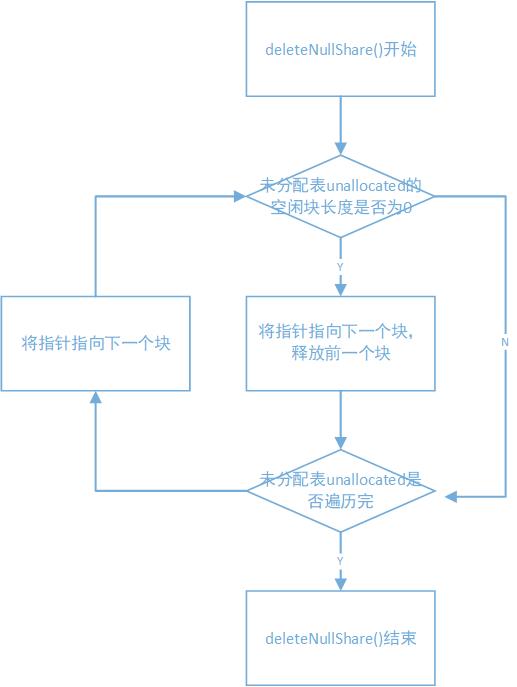
****

图2-4

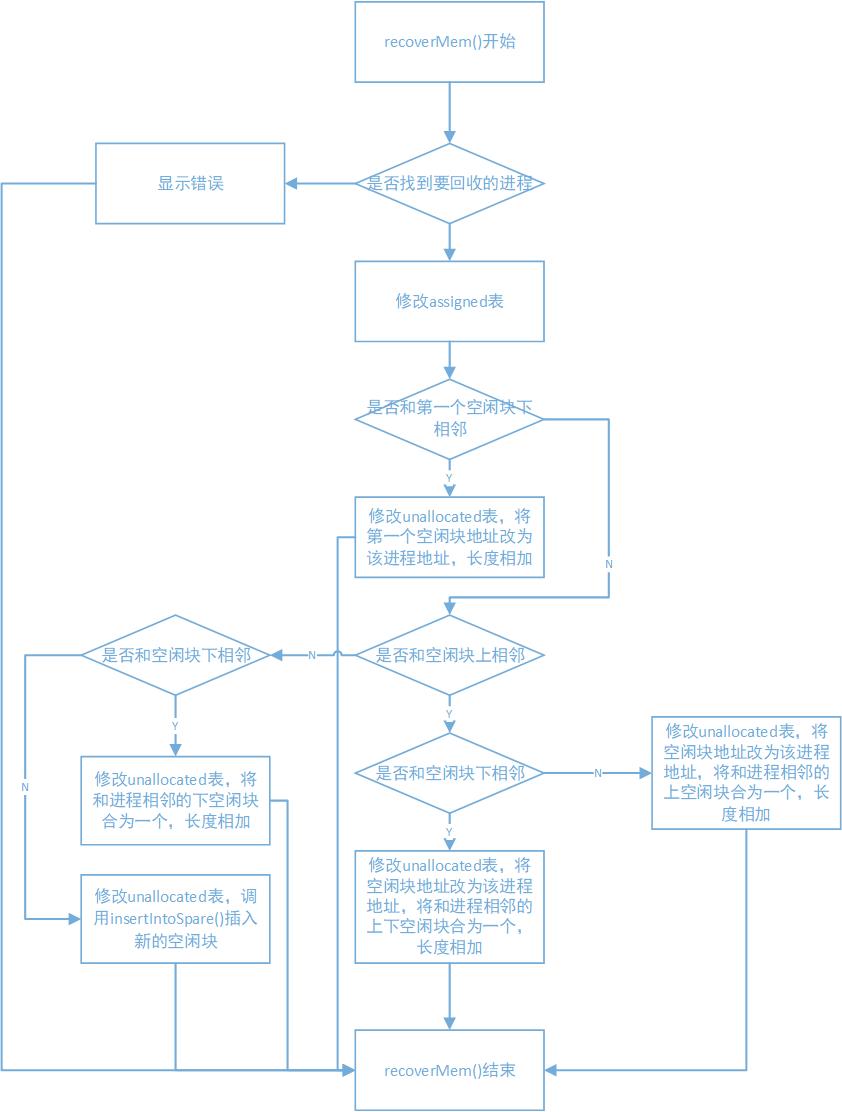
****

图2-5

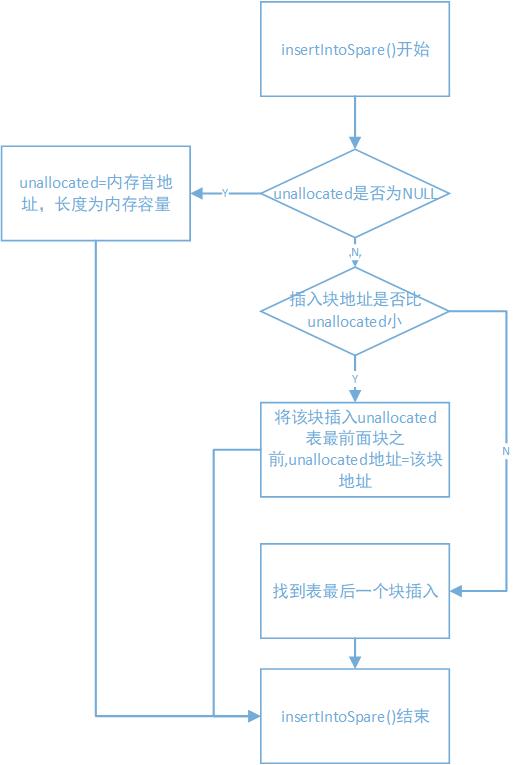
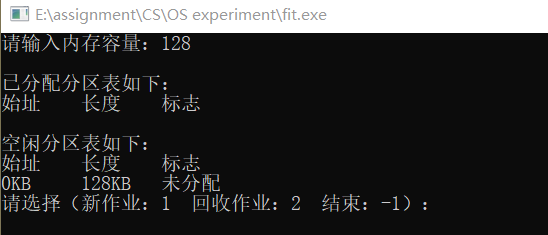
****

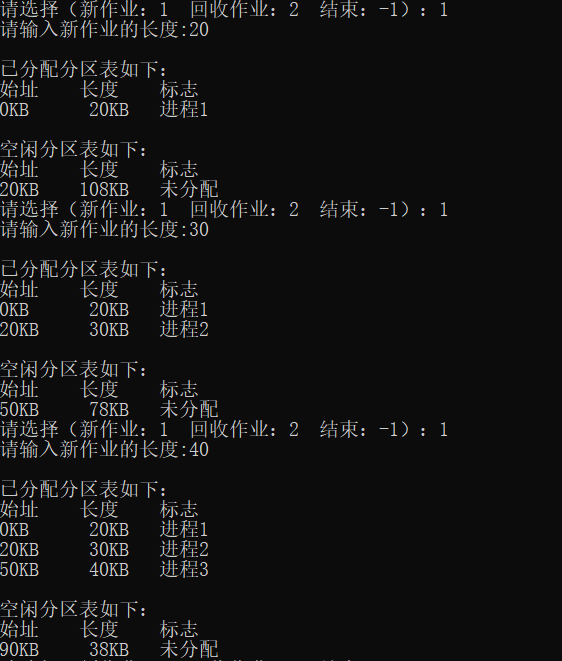
图2-6

**测验：**

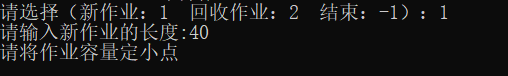
输入内存容量



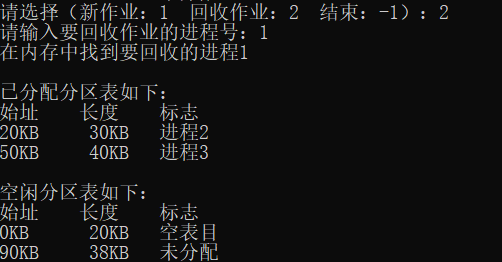
输入作业长度

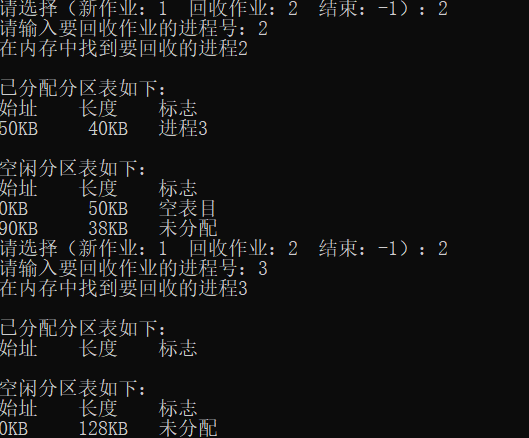


如果此时输入的作业长度大于空闲块长度，会报错



回收进程：





**2.3.3 磁盘存储空间的分配和回收**

**数据结构：**

typedef struct DiskStruct{ //磁盘结构

int cylindernum;//柱面号

int tracknum;//磁道号

int recordnum;//物理记录号

}Disk;

//位示图，由盘组共8个柱面，每个柱面有2个磁道，每个磁道分成4个物理记录，需要64位来表示位示图

struct Bitmap{

int bitmap[COUNT];//0空，1占有

int fileid[COUNT];//请求的物理块属于第几个

};

**用位示图管理磁盘存储空间算法：**

盘组共8个柱面，每个柱面有2个磁道，每个磁道分成4个物理记录，需要64位来表示位示图

申请磁盘空间：

空闲块对应的磁盘物理地址为：

柱面号=字节号

磁道号= 位数 / 4

物理记录号= 位数 % 4

归还磁盘空间：

归还块在位示图中的位置计算：

字节号=柱面号

位数=磁道号×4+物理记录号

算法流程图：

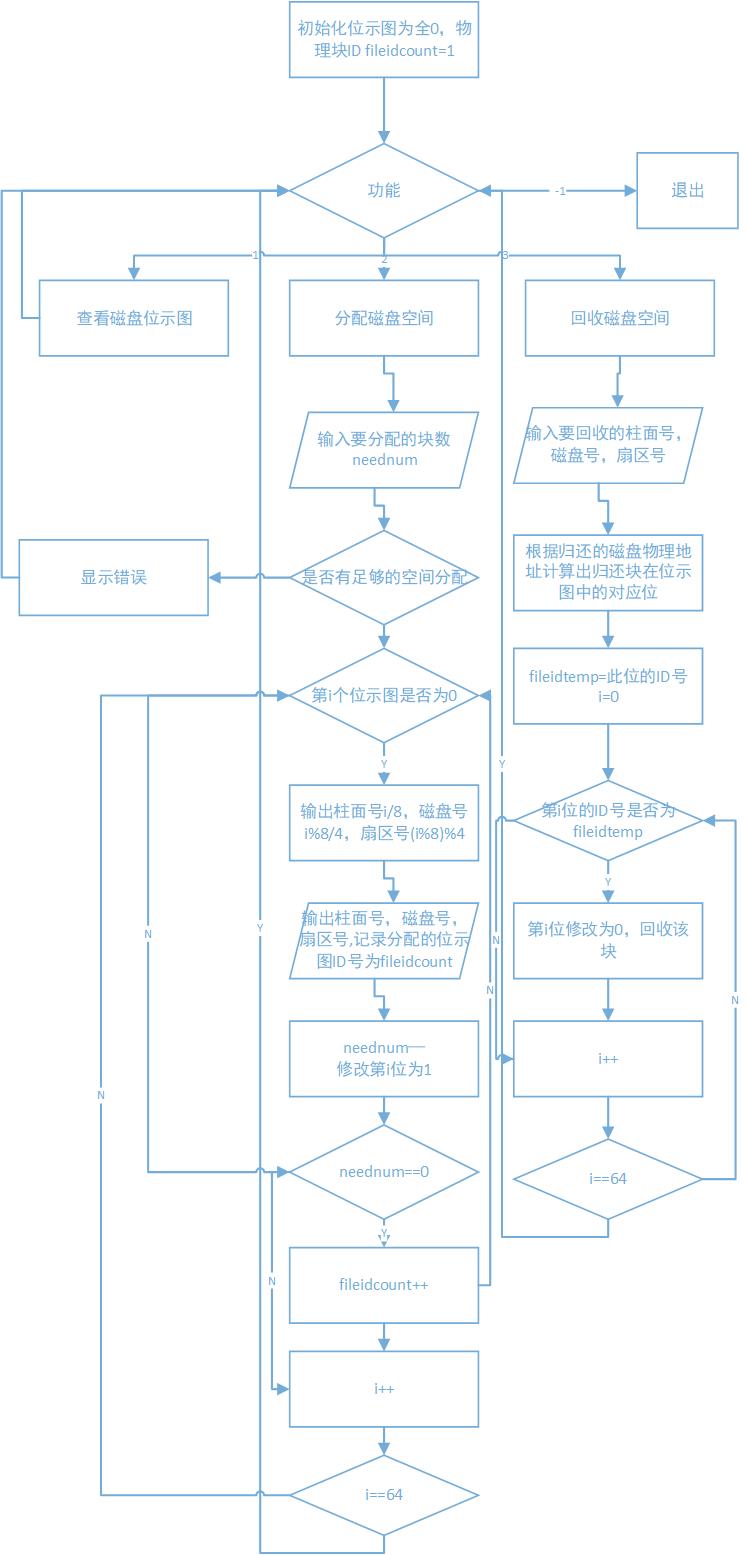
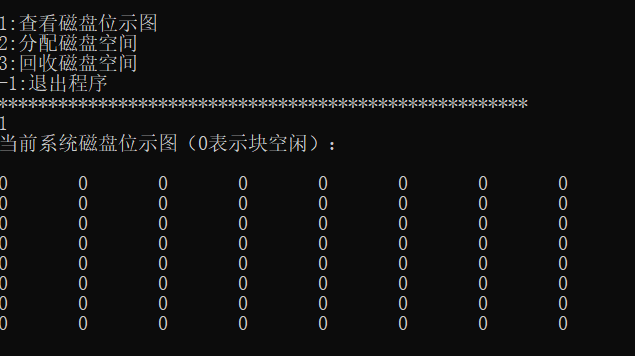


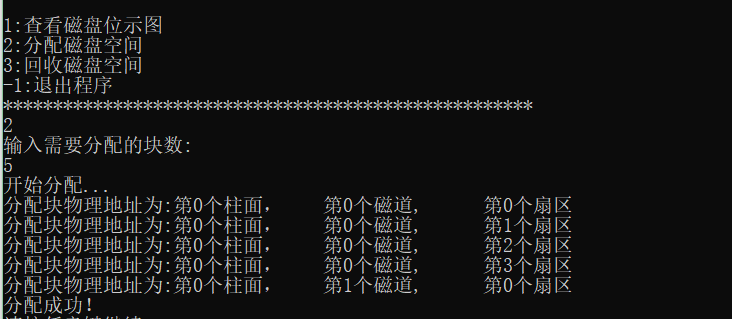
图2-7

**测试：**

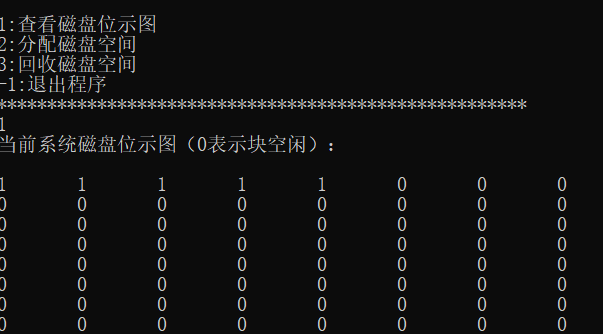
初始化：



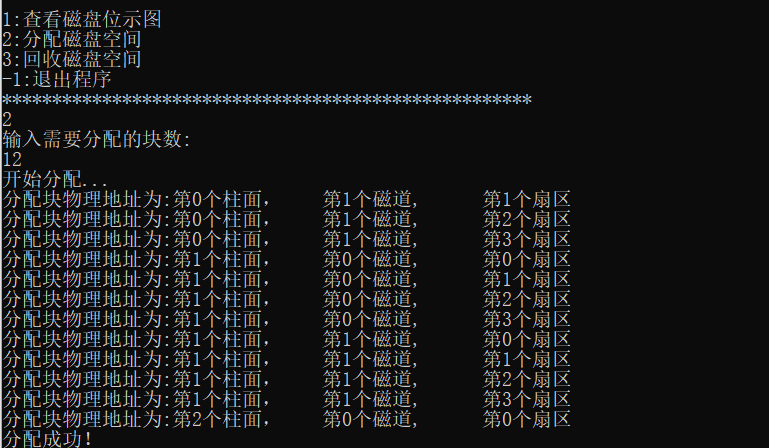
分配磁盘：



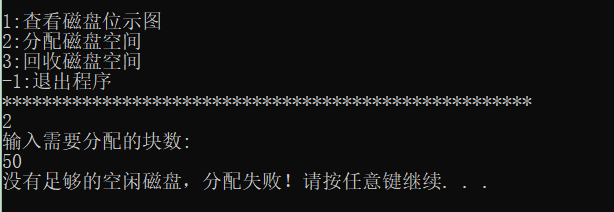
查看分配成功后的位示图：



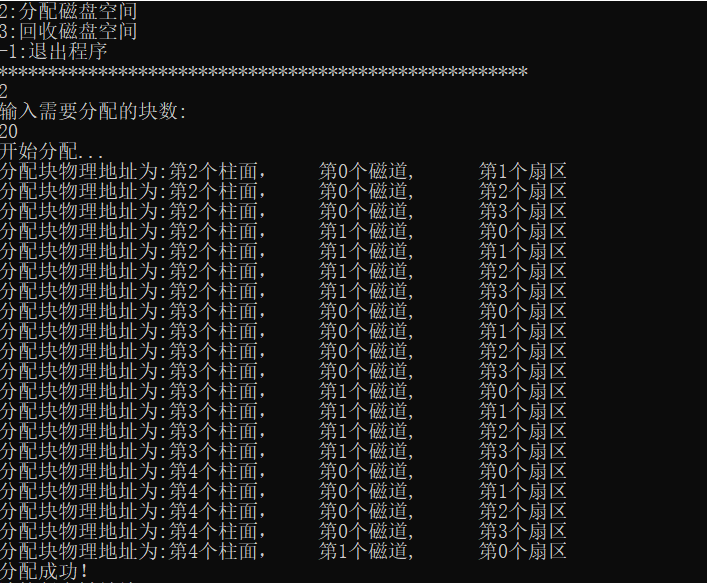
继续分配：



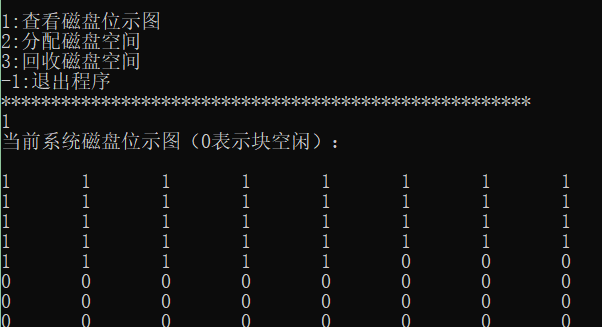
如果需要分配空间大于已有空闲块大小，会报



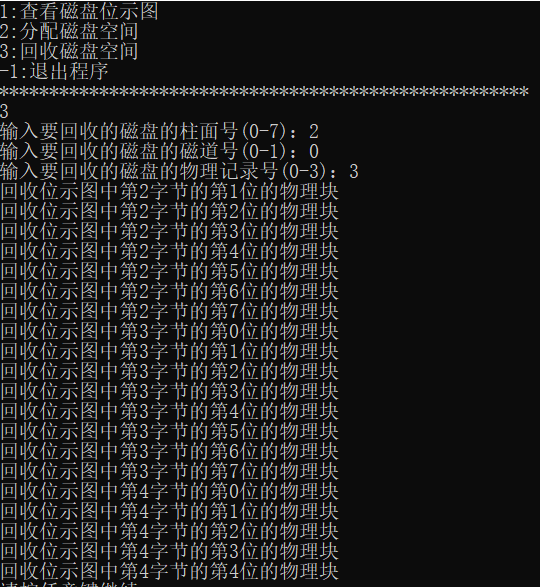
继续分配：



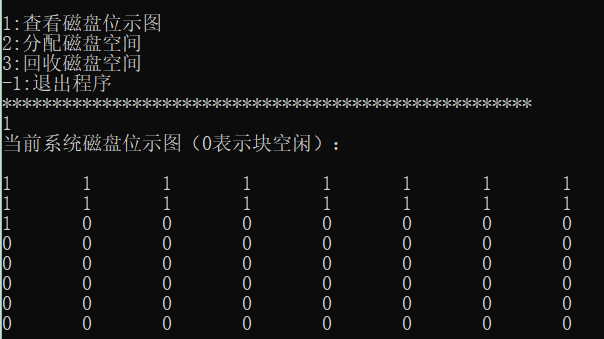
查看位示图：



回收磁盘空间：



查看剩余的磁盘空间：



**2.3.4 银行家算法**

**数据结构**

可利用资源向量Availble：含有m个元素的数组，其中的每一个元素代表一类可用的资源数目。Available[j]=k，则表示系统中现有第j类资源K个。

最大需求矩阵（MAX）：为n×m矩阵，定义了系统中n个进程中的每一个进程对m类资源的最大需求。Max[i,j]=K，则表示进程i需要第j类资源最大数目为K个。

分配矩阵（Allocation）：为n×m矩阵，定义了系统中每一类资源当前已分配给每一进程的资源数。Allocation[i,j]=K，则表示进程i当前已分配的第j类资源的数目为K。

需求矩阵（Need）：为n×m矩阵，表示每个进程尚需的各类资源数，Need[i,j]=K，则表示进程i还需要第j类资源数目为K个。

**银行家算法**

设Request i是进程Pi的请求矢量，如果Request i[j]=K，表示进程Pi需要第j类资源K个。当Pi发出资源请求后，系统按下述步骤进行检测：

1. 如果Request i[j] ≤Need[i,j]，便转向步骤2，否则认为出错，因为它所需的资源数超过了它所宣布的最大值。
2. 如果Request i[j] ≤Available[i,j]，便转向步骤3，否则，表示尚无足够资源，Pi须等待
3. 系统试探着把资源分配给进程Pi，并修改下面数据结构中的数值：

Available[j]=Availabe[j]-Request i[j]

Allocation[i,j]=Allocation[i,j]+Request i[j]

Need[i,j]=Need[i,j]-Request i[j]

1. 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进程Pi，以完成本次分配

**安全性检查算法**

1. 设置两个矩阵。

工作矩阵Work；它表示系统可提供给进程继续运行所需的各类资源数目，它含有m个元素，在执行安全算法开始时，Work=Available；

Finish：它表示系统是否有足够的资源分配给进程，使之运行完成。开始时Finish[i]=false；当有足够资源分配给进程Pi时，再令Finish[i]=true;

1. 从进程集合中找到一个能满足下述条件的进程：

Finish[i]=false;

Need[i,j]=Work[j];

若找到，执行下一步骤，否则，执行步骤4。

1. 当进程Pi获得资源后，可顺利执行，直至完成，并释放分配给它的资源

Work[j]=Work[j]+Allocation[i,j];

Finish[i]=true;

执行步骤2

1. 如果所有进程的Finish[i]=true满足，则表示系统处于安全状态；否则系统处于不安全状态

**算法流程图**

银行家算法流程图：

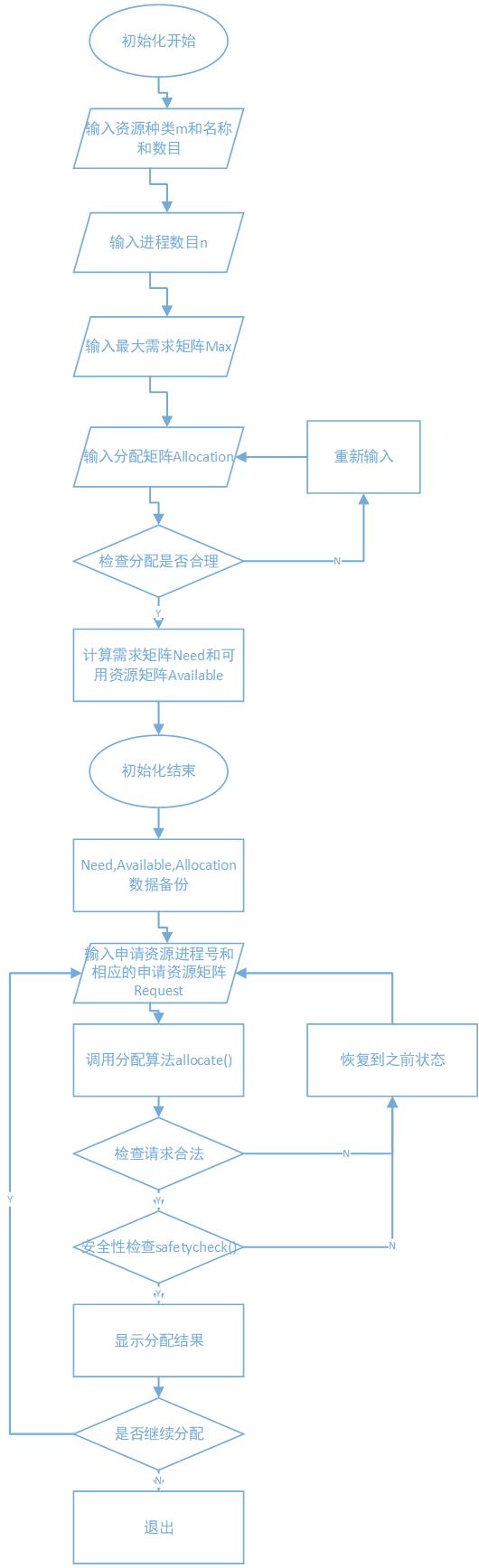


图2-8

分配算法allocate()流程图：

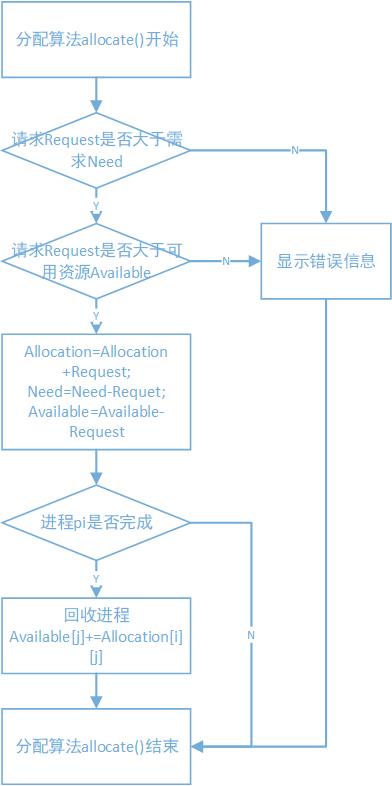


图2-9

安全检查算法safetycheck()：

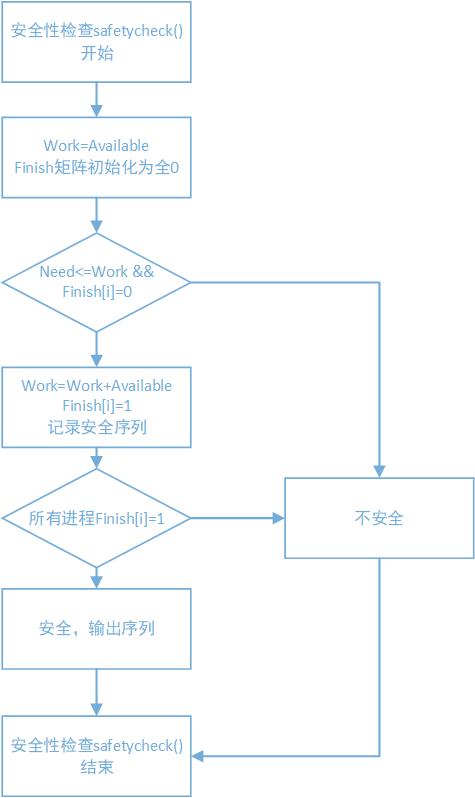
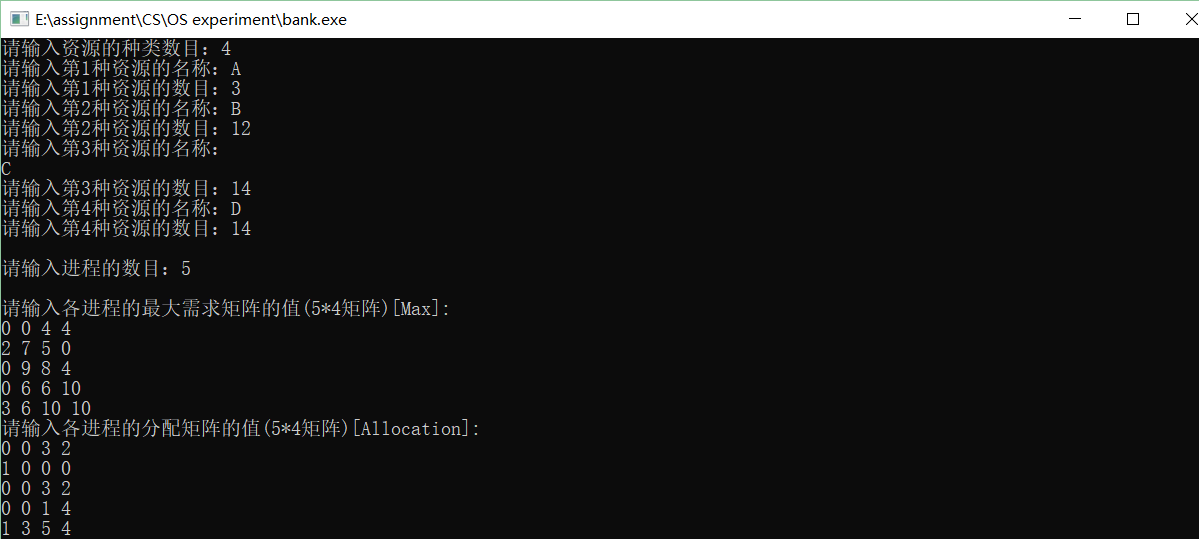


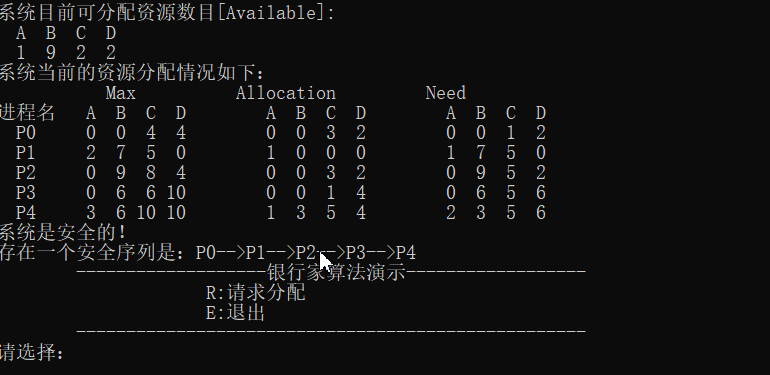
图2-10

**测试**

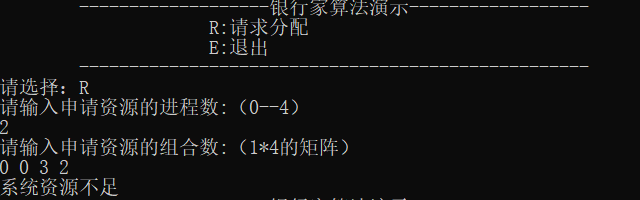
输入资源种类及其名称数量，进程数量，最大需求矩阵和分配矩阵



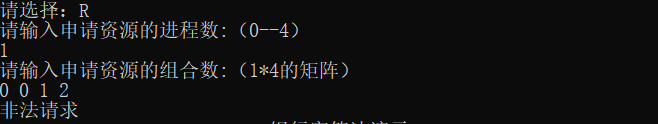
得到可分配矩阵和安全序列检查结果：



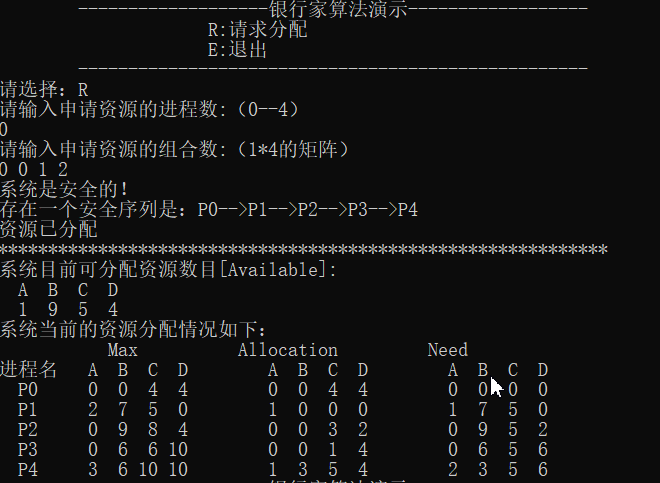
输入R请求分配：可看出请求资源大于系统已有资源会报错



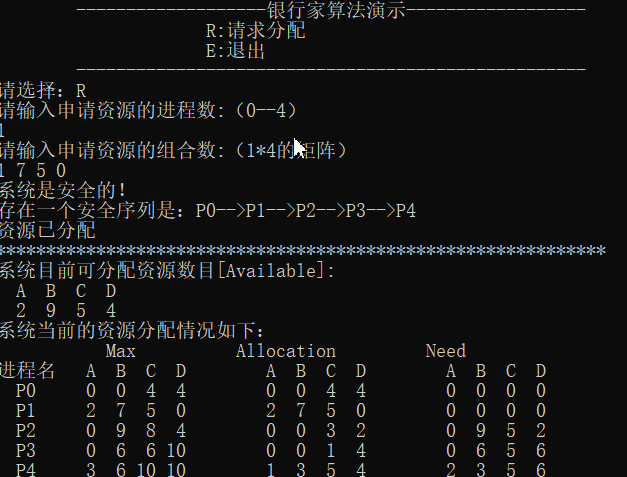
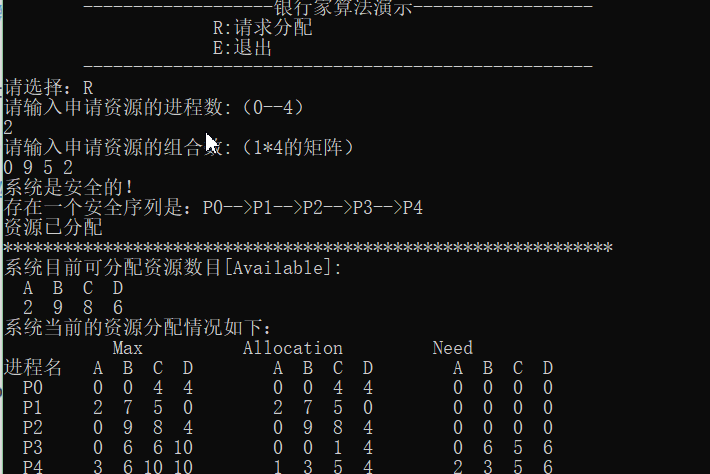
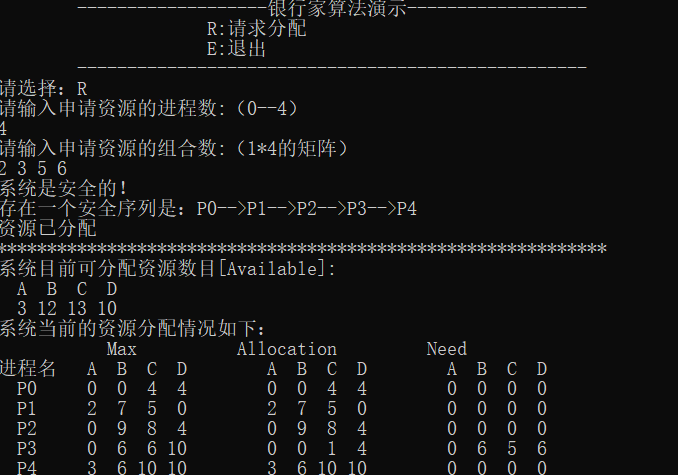
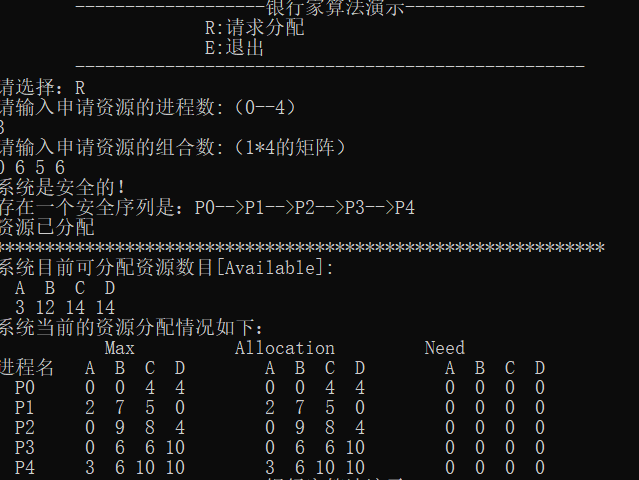
输入R请求分配：可看出请求资源大于需求会报错：



输入R请求分配：请求成功



再继续请求分配：

最后分配完毕和，可看出和之前的资源总数保持一致

**2.3.5 模拟页面地址重定位**

**模拟页面地址重定位算法**：

根据逻辑地址（页长=页号+页面偏移）中的页号查找页表，得到该页在内存中的块号，块号\*页面大小再加上页面偏移等于物理地址。

**算法流程图**：

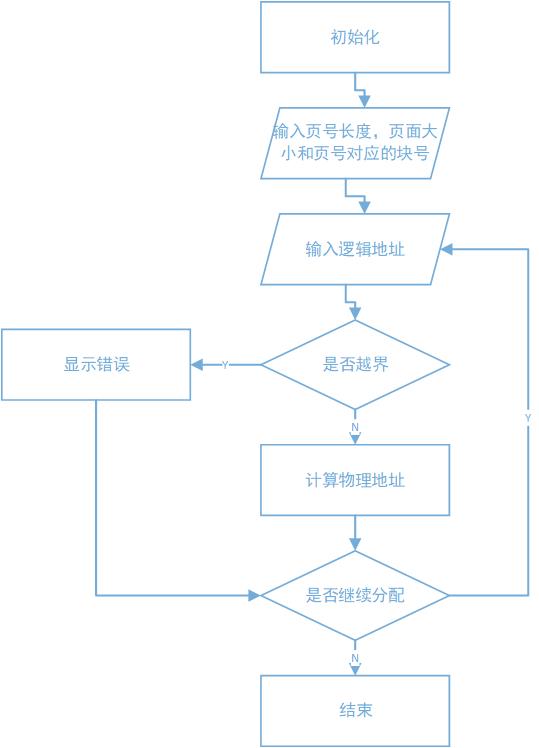
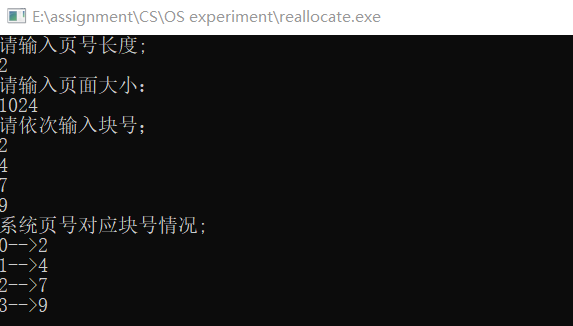


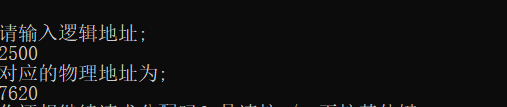
图2-11

**测试：**

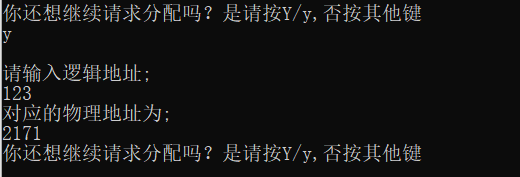
输入信息：



输入逻辑地址：



继续：



退出：



**结论**

通过操作系统的实验，我更加了解了处理器调度的优先级调度算法，主存空间分配和回收的首次适应算法，磁盘存储空间分配和回收中位示图的用法，银行家算法是如何解决死锁问题的，模拟页面地址重地位中逻辑地址是如何通过页表转换成物理地址的。

**附录**

1. **处理器调度**

//按优先数调度算法实现处理器调度的程序

#include "pch.h"

#include<iostream>

#include<cstdlib>

using namespace std;

typedef struct PCB {

struct PCB \*next;//定义指向下一个节点的指针

char name[10];//定义进程名

int Required\_time;//定义需求运行时间

int Prio;//定义优先级

char state;//定义进程状态Ready/End

}\*Proc;

int ProcNum = 5;//总进程数

//初始化就绪队列

void lnitPCB(Proc &H) {

int Num = ProcNum;

H = (Proc)malloc(sizeof(PCB)); //建立头结点

H->next = NULL;

Proc p = H; //定义一个指针

cout << "总进程个数为" << ProcNum << "个，请依次输入相应信息" << endl;

cout << endl;

while (Num--) {

p = p->next = (Proc)malloc(sizeof(PCB));

cout << "进程名，优先数，要求运行时间:";

cin >> p->name >> p->Prio >> p->Required\_time;

p->state = 'R';

p->next = NULL;

}

p->next = H->next;

}

//输入运行中的进程信息

void Displnfo(Proc H) {

Proc p = H->next;

do {

cout << "进程名:" << p->name << "\t优先数:" << p->Prio << "\t要求运行时间:" << p->Required\_time << "\t状态:" << p->state << endl;

p = p->next;

} while (p != H->next); //整个进程链条始终完整，只是状态位有差异

}

Proc Max\_Priority(Proc H) {//确定最大优先级进程子程序

Proc p = H->next;

int max = -100;//max为最大优先数，初始化为-100

Proc key=NULL;

do {

if (p->state == 'W')//W为辅助状态标志，表示正在运行

return NULL;//返回null

else

if (max < p->Prio&&p->state == 'R') {//从就绪进程中选取优先数最大的进程

max = p->Prio;//max存放每次循环中的最大优先数

key = p;//将优先级最高进程赋给key

}

p = p->next;

} while (p != H->next);

if (key->state == 'E')//具有最大优先数的进程若已运行完毕

return NULL;//则返回null

else

return key;//将key作为返回值返回

}

void Priority(Proc &H) {

cout << endl << "----------------START-----------------\n";

int flag = ProcNum;//记录剩余进程数

int round = 0;//记录轮转数

int All\_time = 0;

Proc key;

Proc p = H->next;

do {

All\_time += p->Required\_time;

p = p->next;

} while (p != H->next);

p = H->next;

for (round = 0; round < All\_time; round++) {

cout << "Round" << round + 1;

key = Max\_Priority(H);

if (key != NULL) {

key->state = 'W';

}

if (key->state == 'W') {

cout << "\t正在运行--" << key->name << endl;

key->Required\_time--;

key->Prio--;

if (key->Required\_time == 0) {

key->state = 'E';

flag--;

}

else {

key->state = 'R';

}

}

else

key->state = 'R';

key = key->next;

while (flag&&key->Required\_time == 0)

key = key->next;

Displnfo(H);

}

cout << endl << "-----------------END-----------------\n";

}

int main() {

Proc H;

lnitPCB(H); //数据初始化

Displnfo(H); //输出此刻的进程状态

Priority(H);//优先调度法

system("pause");

}

1. **主存空间的分配和回收**

#include<stdio.h>

#include<stdlib.h>

#define MAXSIZE 50

typedef struct spare{

int start;//始址

int length;//长度

int state;//状态

struct spare \*next;

}SPARESPACE;

int memory;//内存大小

int count;//记录任务序列

SPARESPACE \*unallocated = NULL, \*assigned = NULL;//未分配表unallocated，已分配表assigned

void insertIntoSpare(SPARESPACE \*spare){

SPARESPACE \*p = NULL,\*pre = NULL;

p = pre = unallocated;

if (unallocated == NULL){

unallocated = spare;

}

else if (p->start > spare->start){

spare->next = p;

unallocated = spare;

}

else{

p = p->next;

while (p){

if (p->start > spare->start){

break;

}

pre = p;

p = p->next;

}

spare->next = p;

pre->next = spare;

}

}

void insertIntoAssigned(SPARESPACE \*spare){

SPARESPACE \*p;

p = assigned;

if (assigned == NULL){

assigned = spare;

}

else{

while (p->next){

p = p->next;

}

p->next = spare;

}

}

int init(){

SPARESPACE \*spare;

spare = (SPARESPACE\*)malloc(sizeof(SPARESPACE));

spare->start = 0;

spare->length =memory;

spare->state = 0;

spare->next = NULL;

insertIntoSpare(spare);

count=0;

return 0;

}

void deleteNullShare()

{

SPARESPACE \*pre, \*p;

p = pre = unallocated;

if (p->length == 0){

unallocated = p->next;

free(p);

}

else{

while (p){

if (p->length == 0)

break;

pre = p;

p = p->next;

}

if (p){

pre->next = p->next;

free(p);

}

}

}

void showMemState(){

SPARESPACE \*p = NULL;

SPARESPACE \*temp = NULL;

p = assigned;

printf("\n已分配分区表如下：\n");

printf("始址\t长度\t标志\n");

while (p){

temp = p;

printf("%dKB\t%3dKB\t", p->start, p->length);

if (p->state){

printf("进程%d\n",p->state);

}

p = p->next;

}

p = unallocated;

printf("\n空闲分区表如下：\n");

printf("始址\t长度\t标志\n");

while (p){

printf("%dKB\t%3dKB\t", p->start, p->length);

if(temp==NULL&&p->state==0)

printf("未分配\n");

else if (p->state == 0 )

if(temp->start>p->start)

printf("空表目\n");

else

printf("未分配\n");

p = p->next;

}

}

void recoverMem(int ID){

SPARESPACE \*p = NULL,\*pre=NULL;

p = pre = assigned;

while (p){

if (p->state == ID){

break;

}

pre = p;

p = p->next;

}

if (p == NULL){

printf("内存中没有该进程！\n");

}

else{

printf("在内存中找到要回收的进程%d\n",p->state);

if (p == assigned)

assigned = p->next;

else

pre->next = p->next;

SPARESPACE \*temp = unallocated;

int flag = 0;

if (p->start + p->length == temp->start){//和第一个下相邻

flag = 1;

temp->start = p->start;

temp->length += p->length;

}

else{

while (temp->next){

if (temp->start + temp->length == p->start){//上相邻

flag = 1;

if (p->start + p->length == temp->next->start){//下相邻

temp->length = temp->length + p->length + temp->next->length;

SPARESPACE \*tp = temp->next;

temp->next = temp->next->next;

free(tp);

}

else{//仅上相邻

temp->length = temp->length + p->length;

}

break;

}

if (p->start + p->length == temp->next->start){//仅下相邻

flag = 1;

temp->next->start = p->start;

temp->next->length += p->length;

break;

}

temp = temp->next;

}

if (flag == 0){

if (temp->start + temp->length == p->start){

temp->length += p->length;

}

else{

SPARESPACE \*spare = (SPARESPACE \*)malloc(sizeof(SPARESPACE));

spare->start = p->start;

spare->length = p->length;

spare->state = 0;

spare->next = NULL;

insertIntoSpare(spare);

}

}

}

}

free(p);

}

int firstFit(){

int request;

int select;

int recover;

SPARESPACE \*p;

SPARESPACE \*spare = NULL;

while(1){

showMemState();

printf("请选择（新作业：1 回收作业：2 结束：-1）：");

scanf("%d",&select);

if (select == 1){

printf("请输入新作业的长度:");

do{

scanf("%d", &request);

if(request<=0){

printf("请重新输入一个>0的值：\n");

}

}while(request<=0);

count++;

p = unallocated;

while (p){

if (p->length >= request){

break;

}

p = p->next;

}

if (p == NULL){

printf("请将作业容量定小点\n");

count--;

continue;

}

spare = (SPARESPACE \*)malloc(sizeof(SPARESPACE));

spare->start = p->start;

spare->length = request;

spare->state = count;

spare->next = NULL;

insertIntoAssigned(spare);

//处理分配后的空白分区

p->start = p->start + request;

p->length = p->length - request;

deleteNullShare();

}

else if (select == 2){

printf("请输入要回收作业的进程号：");

scanf("%d",&recover);

recoverMem(recover);

}

else if (select == -1){

break;

}

}

return 0;

}

int main(){

printf("请输入内存容量：");

scanf("%d",&memory);

init();

firstFit();

return 0;

}

1. **磁盘存储空间的分配和回收**

//用位示图管理磁盘存储空间

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#define COUNT 64

typedef struct DiskStruct{ //磁盘结构

int cylindernum;//柱面号

int tracknum;//磁道号

int recordnum;//物理记录号

}Disk;

//位示图，由盘组共8个柱面，每个柱面有2个磁道，每个磁道分成4个物理记录，需要64位来表示位示图

struct Bitmap{

int bitmap[COUNT];//0空，1占有

int fileid[COUNT];//请求的物理块属于第几个

};

struct Bitmap \*bitinfo = NULL;

//打印位示图信息

void showbitmap(){

int i = 0;

printf("当前系统磁盘位示图（0表示块空闲）：\n");

for(i=0;i<COUNT;i++){

if(i%8==0){

printf("\n");

}

printf("%d\t",bitinfo->bitmap[i]);

}

system("pause");

}

int fileidcount=1; //物理块ID

//分配物理块

void diskallocate(){

//由用户输入需要的物理块数：

int neednum;

int i;

int occupy = 0;

printf("输入需要分配的块数:\n");

scanf("%d",&neednum);

for(i=0;i<COUNT;i++){

occupy=occupy+bitinfo->bitmap[i];

}

if(COUNT-occupy<neednum){

printf("没有足够的空闲磁盘，分配失败！");

}

else{

printf("开始分配...\n");

for(i=0;i<COUNT;i++){

if( bitinfo->bitmap[i]==0){

printf("分配块物理地址为:第%d个柱面，\t第%d个磁道,\t第%d个扇区\n",i/8,i%8/4,(i%8)%4);

bitinfo->bitmap[i] = 1;

bitinfo->fileid[i]=fileidcount;

neednum--;

}

if(neednum==0){

fileidcount++;

break;

}

}

printf("分配成功！\n");

}

system("pause");

}

//回收

void diskrecycle(){

int fileidtemp,bitmapid;

Disk disk;

int i;

do{

printf("输入要回收的磁盘的柱面号(0-7)：");

scanf("%d",&disk.cylindernum);

}while(disk.cylindernum<0||disk.cylindernum>7);

do{

printf("输入要回收的磁盘的磁道号(0-1)：");

scanf("%d",&disk.tracknum);

}while(disk.tracknum!=0&&disk.tracknum!=1);

do{

printf("输入要回收的磁盘的物理记录号(0-3)：");

scanf("%d",&disk.recordnum);

}while(disk.recordnum<0||disk.recordnum>3);

//根据归还的磁盘物理地址计算出归还块在位示图中的对应位

bitmapid=disk.cylindernum\*8+disk.tracknum\*4+disk.recordnum;

fileidtemp=bitinfo->fileid[bitmapid];

for( i=0;i<COUNT;i++){

if(fileidtemp!=bitinfo->fileid[i]){

continue;

}

else{

bitinfo->bitmap[i] = 0;

printf("回收位示图中第%d字节的第%d位的物理块\n",i/8,i%8);

}

}

system("pause");

}

int main (){

int j;

//初始化

bitinfo = (struct Bitmap\*)malloc(sizeof(struct Bitmap));

for(j=0;j<COUNT;j++){

bitinfo->bitmap[j]=0;

bitinfo->fileid[j]=0;

}

//功能选择

int func = 0;

while(1){

printf("\n1:查看磁盘位示图\n");

printf("2:分配磁盘空间\n");

printf("3:回收磁盘空间\n");

printf("-1:退出程序\n");

printf("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

scanf("%d",&func);

switch(func){

case 1 :showbitmap();break;

case 2 :diskallocate();break;

case 3 :diskrecycle();break;

case -1 :return 0;

}

}

}

1. **银行家算法：**

#include<iostream>

#include<iomanip>

using namespace std;

//定义变量并初始化

//进程的最大数

int N=10;

//资源的最大数

int M=50;

//可利用资源向量

int Available[50] = {0};

int Available\_A[50] = {0};

//最大需求量

int Max[10][50] = {0};

//已分配资源量

int Allocation[10][50] = {0};

int Allocation\_A[10][50] = {0};

//尚需资源数目

int Need[10][50] = {0};

int Need\_N[10][50] = {0};

//请求资源向量

int Request[100]={0};

//工作向量

int Work[50] = {0};

//分配向量

int Finish[50] = {0};

//安全序列

int Security[10] = {0};

//判断值

int Bool = 0;

//暂存矩阵

int Temp[50] = {0};

//资源名称

char Name[20] = {0};

//初始化资源数目M、输入每一种资源的名称和数目以及进程数目N

void input\_1(){

int m,n;

cout<<"请输入资源的种类数目：";

cin>>m;

if(m<=M){

M=m;

//输入每一种资源的名称和数目Available

for(int i=1;i<=M;i++){

cout<<"请输入第"<<i<<"种资源的名称：";

cin>>Name[i-1];

cout<<"请输入第"<<i<<"种资源的数目：";

cin>>Available[i-1];

}

cout<<endl;

}

else

cout<<"资源数目太多，请重新输入···"<<endl;

cout<<"请输入进程的数目：";

cin>>n;

if(n<=N){

N=n;

cout<<endl;

}

else

cout<<"进程太多，请重新输入···"<<endl;

}

//输入最大需求矩阵和分配矩阵

void input\_2(){

int i,j;

//最大需求矩阵

cout<<"请输入各进程的最大需求矩阵的值("<<N<<"\*"<<M<<"矩阵)[Max]:"<<endl;

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

for(j=0;j<M;j++){

cin>>Max[i][j];

}

}

//分配矩阵

cout<<"请输入各进程的分配矩阵的值("<<N<<"\*"<<M<<"矩阵)[Allocation]:"<<endl;

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

for(j=0;j<M;j++){

cin>>Allocation[i][j];

}

}

}

//计算系统目前可用资源和需求矩阵

void input\_4();

void input\_3(){

int i,j;

Bool=1;

//判断分配是否合理（看是否有分配大于最大需求）

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

for(j=0;j<M;j++){

if(Allocation[i][j]>Max[i][j]){

cout<<"不合理分配"<<endl;

Bool=0;

break;

}

}

}

//如果判断合理

if(Bool==1){

//计算系统目前可用资源

for(j=0;j<M;j++){

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

Temp[j] += Allocation[i][j];

}

Available[j] -= Temp[j];

}

//得到需求矩阵

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

for(j=0;j<M;j++){

Need[i][j] = Max[i][j]-Allocation[i][j];

}

}

}

//如果判断分配不合理

else

input\_4();

}

//重新输入

void input\_4(){

int i,j;

cout<<"请重新输入各进程的分配矩阵的值("<<N<<"\*"<<M<<"矩阵)[Allocation]:"<<endl;

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

for(j=0;j<M;j++){

cin>>Allocation[i][j];

}

}

//继续判断

input\_3();

}

//显示信息

void show(){

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

cout<<"系统目前可分配资源数目[Available]:"<<endl;

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

cout<<setw(3)<<Name[i];

}

cout<<endl;

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

cout<<setw(3)<<Available[i];

}

cout<<endl;

cout<<"系统当前的资源分配情况如下："<<endl;

cout<<" Max Allocation Need"<<endl;

cout<<"进程名 ";

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

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

cout<<setw(3)<<Name[i];

cout<<" ";

}

cout<<endl;

//打印矩阵

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

cout<<" P"<<i<<" ";

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

cout<<setw(3)<<Max[i][j];

cout<<" ";

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

cout<<setw(3)<<Allocation[i][j];

cout<<" ";

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

cout<<setw(3)<<Need[i][j];

cout<<endl;

}

}

//判断此时系统是否是安全的

int safetycheck(){

int i,j;

for(j=0;j<M;j++){

Work[j]=Available[j];

}

int s=0;

//本层循环保证进行N次排查，使得前几次排查过程中不能满足的进程能继续判断

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

//i=0;

//本层循环控制从0~N-1个进程的判断

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

//判断进程资源请求状态是否为未完成

if(Finish[i]==0){

//判断每一种资源的需求是否小于可用

j=0;

while(j<M){

if(Need[i][j]<=Work[j]){

//该条件满足时，说明符合分配规则，进行试分配

if(j==M-1){

//分配资源

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

Work[b] += Allocation[i][b];

}

//修改状态

Finish[i]=1;

//记录安全顺序

Security[s]=i;

s++;

}

j++;

}

//只要有一种需求不能被满足，则跳过

else

break;

}

}

}

}

Bool=1;

//判断是否有没有得到分配的进程

//如果有，使Bool的值改为 0

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

if(Finish[j]==0){

Bool=0;

break;

}

}

//Bool=1表示所有进程的Finish都等于 1 ，既系统是安全的

if(Bool==1){

cout<<"系统是安全的！"<<endl;

cout<<"存在一个安全序列是：";

//输出安全序列

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

cout<<"P"<<Security[j];

if(j<N-1)

cout<<"-->";

}

cout<<endl;

return Bool;

}

//Bool！=1；说明有没有得到资源的进程，系统不安全

else{

cout<<"系统不是安全的！"<<endl;

return Bool;

}

}

//分配资源

int allocate(int p){

int j=0;

Bool=1;

int flag=1;

//判断申请的合法性

for(j=0;j<M;j++){

//判断请求是否大于需求

if(Request[j]>Need[p][j]){

Bool=0;

//大于时将 0 值返回给主调函数

return Bool;

break;

}

//判断请求是否大于可用资源

else if(Available[j]<Request[j]){

Bool=-1;

//大于时将 -1 值返回给主调函数

return Bool;

break;

}

}

//当Bool值还是 1 ，说明请求合理，给予分配

if(Bool==1){

for(j=0;j<M;j++){

Allocation[p][j] += Request[j];

Need[p][j] -= Request[j];

Available[j] -= Request[j];

}

//判断进程是否完成，如果完成，回收进程，更新可用资源

for(j=0;j<M;j++){

if(Need[p][j]!=0)

flag=0;

break;

}

if(flag==1){

for(j=0;j<M;j++){

Available[j]+=Allocation[p][j];

}

}

return Bool;

}

}

//发出资源申请请求并检验请求的合理性

void bank(){

int i,j,p;

//数据备份

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

Available\_A[i]=Available[i];

for(j=0;j<M;j++){

Allocation\_A[i][j]=Allocation[i][j];

Need\_N[i][j]=Need[i][j];

}

}

//接收申请

cout<<"请输入申请资源的进程数:（0--"<<N-1<<"）"<<endl;

cin>>p;

cout<<"请输入申请资源的组合数:（1\*"<<M<<"的矩阵）"<<endl;

for(j=0;j<M;j++){

cin>>Request[j];

}

//调用分配算法

int value\_1=allocate(p);

//如果请求不合法

if(value\_1==0){

cout<<"非法请求"<<endl;

//退出该函数

return;

}

if(value\_1==-1){

cout<<"系统资源不足"<<endl;

//退出该函数

return;

}

//如果请求符合规范

if(value\_1==1){

//执行安全性算法

int value\_2=safetycheck();

//符合安全性算法

if(value\_2==1){

cout<<"资源已分配"<<endl;

show();

}

//不符合安全性算法

else{

cout<<"该请求导致系统不安全，不进行分配。"<<endl;

//数据还原

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

Available[i]=Available\_A[i];

for(j=0;j<M;j++){

Allocation[i][j]=Allocation\_A[i][j];

Need[i][j]=Need\_N[i][j];

}

}

show();

}

}

}

//主函数

int main(){

char choice;

int a=1;

//输入

input\_1();

input\_2();

input\_3();

//显示

show();

//判断输入的状态是否安全

safetycheck();

//功能选项

while(a==1){

cout<<"\t-------------------银行家算法演示------------------"<<endl;

cout<<" R:请求分配 "<<endl;

cout<<" E:退出 "<<endl;

cout<<"\t---------------------------------------------------"<<endl;

cout<<"请选择：";

cin>>choice;

switch(choice){

case 'R':

bank();

break;

case 'E':

a=0;

break;

default: cout<<"请正确选择!"<<endl;

break;

}

}

return 0;

}

/\*示例：

4

A

3

B

12

C

14

D

14

5

max：

0 0 4 4

2 7 5 0

0 9 8 4

0 6 6 10

3 6 10 10

allocate:

0 0 3 2

1 0 0 0

0 0 3 2

0 0 1 4

1 3 5 4

\*/

1. **模拟页面地址重定位：**

#include <iostream>

#include <math.h>

#define PagetableLength 64

using namespace std;

int PageTable[PagetableLength] = {0};

int main(){

int LogicalAddress = 0;

int PageNum = 0;

int w = 0;

int i = 0;

int num = 0;

int f;

int PageSize;

char flag;

cout << "请输入页号长度;" << endl;

cin >> num;

cout << "请输入页面大小：" << endl;

cin >> PageSize;

cout << "请依次输入块号；" << endl;

for(i = 0; i < pow(2,num); i++ ){

cin >> PageTable[i];

}

cout << "系统页号对应块号情况;\n";

for (i = 0; i <pow(2,num); i++){

cout<<i<<"-->"<< PageTable[i]<<endl;

}

while(f){

cout << endl << "请输入逻辑地址;\n";

cin >> LogicalAddress;

PageNum = LogicalAddress / PageSize;

w = LogicalAddress % PageSize;//页内偏移地址

if(PageNum >= PagetableLength){//判断是否越界

cout << "本次访问的地址已超出进程的地址空间，错误！；\n";

return -1;

}

cout << "对应的物理地址为;" << endl << PageTable[PageNum] \* PageSize + w << endl;

cout << "你还想继续请求分配吗？是请按Y/y,否按其他键" << endl;

cin >> flag;

if(flag=='Y'||flag=='y')

f=1;

else

break;

}

return 0;

}