廣東工業大學

实 验 报 告

课程名称 操作系统实验

学生学院 先进制造学院

专业班级 22 计科 8 班

学　　号 3122008883

学生姓名 陈煜祺

指导教师 苏畅

2024 年 6 月 20 日

实验一　利用银行家算法实现进程调度

一、实验目的

通过编写一个模拟动态资源分配的银行家算法程序，进一步深入理解死锁、产生死锁的必要条件、安全状态等重要概念，并掌握避免死锁的具体实施方法。

二、实验内容

（1）模拟银行家算法：设置数据结构设计安全性算法

（2）初始化时让系统拥有一定的资源

（3）用键盘输入的方式申请资源

（4）如果预分配后，系统处于安全状态，则修改系统的资源分配情况

（5）如果预分配后，系统处于不安全状态，则提示不能满足请求

三、实现思路

3.1 程序主体结构的实现思路

(1)初始化 init()：输入进程数量、资源种类、资源可利用量、进程资源已分配量、进程最大需求量

(2)当前安全性检查 safe()：用于判断当前状态安全

(3)银行家算法 bank()：进行银行家算法模拟实现的模块

(4)显示当前状态 show()：显示当前资源分配详细情况

(5)主程序 main()：逐个调用初始化、显示状态、安全性检查、银行家算法函数，使程序有序地进行

3.2 银行家算法 bank()函数的实现思路

先对用户提出的请求进行合法性检查，即检查请求的是否不大于需要的，是否不大于可利用的。 若请求合法，则进行试分配。最后对试分配后的状态调用安全性检查算法进行安全性检查。 若安全，则分配，否则，不分配，恢复原来状态，拒绝申请。

Requesti：进程 Pi 的请求向量。 0<=j<=m-1

(1) 若 Requesti[j] ≤ Need[i,j]，转向(2)，否则出错。

(2) 若 Requesti[j] ≤ Available[j]，转向(3)，否则等待。

(3) 系统试探着把资源分配给进程 Pi，修改下面内容：

Available[j] = Available[j] - Requesti[j];

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

Need[i,j] = Need[i,j] -Requesti[j];

(4) 试分配后，执行安全性算法，检查此次分配后系统是否处于安全状态。若安全，才正式分配；否则，此次试探性分配作废，进程 Pi 等待。

3.3 安全性算法 safe()函数的实现思路

(1) 初始化：设置两个向量 Work(1×m)和 Finish(1×n)

Work - 系统可提供给进程继续运行所需各类资源数，初态赋值 Available

Finish - 系统是否有足够资源分配给进程，初值 false.

(2) 从进程集合中满足下面条件进程：

Finish[i] = false; Need[i,j] ≤ Work[j];

若找到，执行(3)，否则，执行(4)。

(3) 进程 Pi 获得资源，可顺利执行，完成释放所分配的资源。

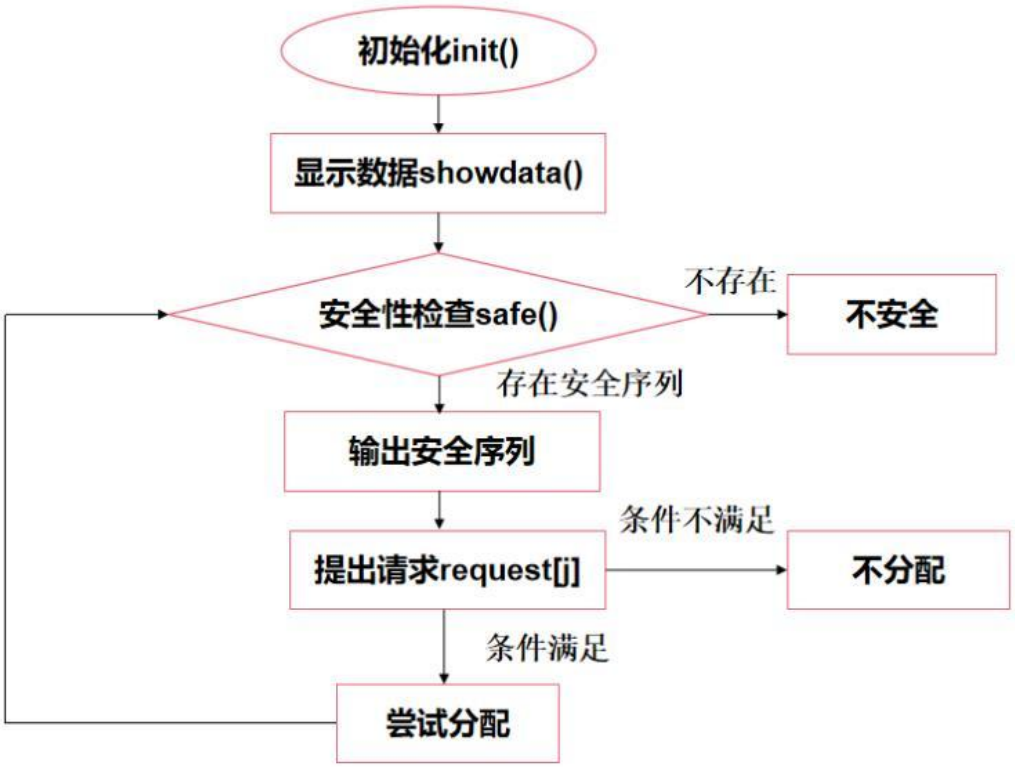
Work[j] = Work[j]+Allocation[i,j]; Finish[i] = true; go to (2).

(4) 若所有进程 Finish[i] = true，表示系统处于安全状态，否则处于不安全状态。

四、主要的数据结构

```
char NAME[100]={0};                 //资源的名称
int Max[100][100]={0};              //最大需求矩阵
int Allocation[100][100]={0};       //系统已分配矩阵
int Need[100][100]={0};             //还需要资源矩阵
int Available[100]={0};             //可用资源矩阵
int Request[100]={0};               //请求资源向量
int Work[100]={0};                  //工作向量，存放系统可提供资源量
int Finish[100]={0};                //标记系统是否有足够的资源分配给各个进程
int Security[100]={0};              //存放安全序列
int M=100;                          //进程的最大数
int N=100;                          //资源的最大数
```

五、程序流程图



六、运行与测试

6.1 输入初始条件

```
系统可用资源种类为:3
资源0的名称:A
资源A的初始个数为:10
资源1的名称:B
资源B的初始个数为:5
资源2的名称:C
资源C的初始个数为:7

请输入进程的数量:5
请输入各进程的最大需求矩阵的值[Max]:
7 5 3
3 2 2
9 0 2
2 2 2
4 3 3
请输入各进程已经分配的资源量[Allocation]:
0 1 0
2 0 0
3 0 2
2 1 1
0 0 2
**********************************************
```

6.2 显示各种资源并对初始数据进行安全性检查

```
************************************************************
系统目前可用的资源[Available]:
A B C
3 3 2
系统当前的资源分配情况如下:
 Max Allocation Need
进程名 A B C  A B C  A B C
 P0 7 5 3  0 1 0  7 4 3
 P1 3 2 2  2 0 0  1 2 2
 P2 9 0 2  3 0 2  6 0 0
 P3 2 2 2  2 1 1  0 1 1
 P4 4 3 3  0 0 2  4 3 1
系统是安全的!
存在一个安全序列: P1->P3->P0->P2->P4
************************************************************
```

回答实验指导手册中的问题：T0 时刻系统处于安全状态。

6.3 T0 之后的 T1 时刻 P1 请求资源 Request1(1,0,2)

```
        ------------------银行家算法演示------------------
 R(r):请求分配
 E(e):退出
        --------------------------------------------------
请选择: R
请输入请求分配资源的进程号(0-4):1
请输入进程P1要申请的资源个数:
A:1
B:0
C:2
************************************************************
系统目前可用的资源[Available]:
A B C
2 3 0
系统当前的资源分配情况如下:
 Max Allocation Need
进程名 A B C  A B C  A B C
 P0 7 5 3  0 1 0  7 4 3
 P1 3 2 2  3 0 2  0 2 0
 P2 9 0 2  3 0 2  6 0 0
 P3 2 2 2  2 1 1  0 1 1
 P4 4 3 3  0 0 2  4 3 1
系统是安全的!
存在一个安全序列: P1->P3->P0->P2->P4
************************************************************
```

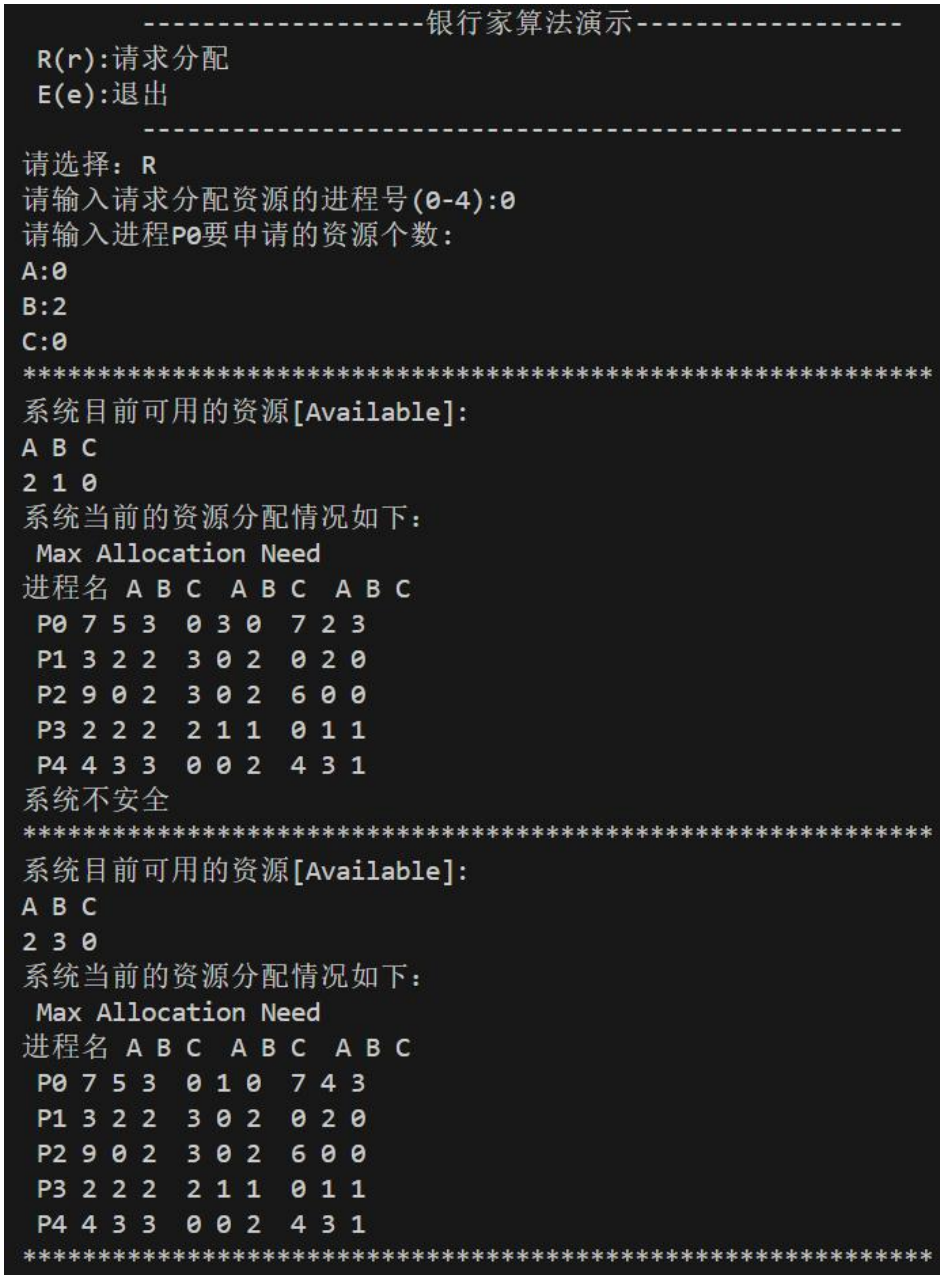
回答实验指导手册中的问题：试分配后进行安全算法检查后得知系统处于安全状态，故 T0 之后的 T1 时刻 P1 请求资源 Request1(1,0,2)将被允许。

6.4 T1 之后的 T2 时刻 P4 请求资源 Request4(3,3,0)

```
        ----------------------银行家算法演示-----------------------
 R(r):请求分配
 E(e):退出
        ------------------------------------------------------------
请选择：R
请输入请求分配资源的进程号(0-4):4
请输入进程P4要申请的资源个数：
A:3
B:3
C:0
进程4申请的资源大于系统现在可利用的资源
系统尚无足够资源，不予分配！
************************************************************************
```

回答实验指导手册中的问题：进程 4 申请的资源大于系统现在可以利用的资源，即可用资源暂不能满足 P4 请求资源需要，P4 阻塞等待，T1 之后的 T2 时刻 P4 请求资源 Request4(3,3,0)将不被允许。

6.5 T2 之后的 T3 时刻 P0 请求资源 Request0(0,2,0)



回答实验指导手册中的问题：试分配后进行安全算法检查后得知系统处于不安全状态，故 T2 之后的 T3 时刻 P0 请求资源 Request0(0,2,0)将不被允许，撤销试分配，回到原来的状态。

七、实验总结与分析

死锁是一种现象，当两个或两个以上的进程在执行过程中，因竞争资源或彼此通信而相互阻塞，如果没有外力介入，它们都无法继续执行，此时系统就处于死锁状态，这些进程被称为死锁进程。银行家算法是一种有效的算法，用于避免操作系统中死锁的发生。

如果系统中所有进程都能构成一个安全序列 P1，…，Pn，那么系统就处于安全状态。安全状态意味着系统不会发生死锁。相反，如果不存在这样一个安全序列，系统则处于不安全状态，虽然这并不意味着一定会发生死锁，但我们仍应避免系统进入这个状态。

Dijkstra 在 1968 年提出的银行家算法是避免死锁的代表性算法之一。该算法的核心是确保系统不进入不安全状态，从而防止死锁的发生。为了实现银行家算法，系统需要设置一些特定的数据结构。它的基本思想和流程可以概括如下：先对用户提出的请求进行合法性检查，即检查请求的是否不大于需要的，是否不大于可利用的。 若请求合法，则进行试分配。最后对试分配后的状态调用安全性检查算法进行安全性检查。 若安全，则分配，否则，不分配，恢复原来状态，拒绝申请。

经过本次银行家算法的模拟实验，我深刻体会到了操作系统中资源分配的复杂性以及避免死锁的重要性。通过编写和测试银行家算法程序，我对死锁的四个必要条件、安全状态的概念有了更加深入的理解，并且掌握了一种有效的死锁避免方法。

在实验过程中，我首先设计了必要的数据结构来模拟资源的分配和需求，包括资源的最大需求矩阵、已分配矩阵、还需资源矩阵以及可用资源矩阵等。通过对这些数据结构的操作，我实现了初始化、安全性检查、资源请求处理等关键功能。

我认识到，银行家算法的核心在于通过试分配后的安全性算法，确保每次资源分配后系统仍然处于安全状态。在编写安全性算法时，我学习了如何通过工作向量和完成向量来寻找系统的安全序列，从而判断系统是否安全。

总的来说，这次实验不仅加深了我对银行家算法的理解，也锻炼了我的编程能力和问题解决能力。

实验二　动态分区分配方式的模拟

一、实验目的

了解动态分区分配方式中使用的数据结构和分配算法，并进一步加深对动态分区存储管理方式及其实现过程的理解。

二、实验内容

（1） 用 C 语言分别实现采用**首次适应算法、最佳适应算法、最坏适应算法**的动态分区分配过程。其中，空闲分区通过空闲分区链来管理；在进行内存分配时，系统优先使用空闲区低端的空间。

（2） 假设初始状态下可用的内存空间为 640KB，并有下列内存请求序列：

- 作业 1 申请 130KB。
- 作业 2 申请 60KB。
- 作业 3 申请 100KB。
- 作业 2 释放内存。
- 作业 4 申请 200KB。
- 作业 3 释放内存。
- 作业 1 释放内存。
- 作业 5 申请 140KB。
- 作业 6 申请 60KB。
- 作业 7 申请 50KB。
- 作业 6 释放内存。

请编写程序分别使用上述几种算法模拟上述内存分区的分配和回收过程，要求在每次分配和回收后显示空闲内存分区链的情况，输出本次分配和回收的查找开销。

（3） 拓展实验

设计不同的内存请求、释放序列，对比验证几种分配算法的特点。

- 请结合理论和本次模拟数据分析对比采用上述几种算法对内存的分配和回收速度有什么不同的影响？

● 请结合理论和本次模拟数据分析对比采用上述几种算法对分区内存利用率有什么不同的影响？如何解决因碎片而造成内存分配速度降低的问题？

三、实现思路

3.1 整体思路

使用双向链表来管理内存的空闲区域，实现对主存分配和回收过程的模拟。程序支持三种不同的内存分配算法：首次适应算法（FF）、最佳适应算法（BF）和最坏适应算法（WF）。用户可以通过选择不同的算法来分配内存，并且可以查看内存的当前分配状态，或者回收已分配的内存块。

3.2 功能介绍

内存分配：用户可以输入所需的内存大小，程序将根据用户选择的算法在内存中查找合适的空闲区域，并将其标记为已分配状态。

内存回收：用户可以指定要回收的内存分区号，程序将该分区标记为空闲状态，并且如果可能的话，算法会将相邻的空闲区域合并，以减少内存碎片。

内存状态查看：用户可以查看当前内存的分配情况，包括每个分区的起始地址、大小和状态（空闲或已分配）。

3.3 算法说明

FF（首次适应算法）：该算法从空闲分区表的起始位置开始搜索，旨在将最先发现的、满足需求的空闲区分配给作业。其核心优势在于能够显著减少搜索时间。为了优化首次适应算法的执行效率，空闲分区表（或称为空闲区链）中的空闲分区必须按照内存地址的递增顺序进行排序。该算法倾向于优先分配低地址区域的空闲区，这可能导致低地址区域出现许多小的空闲区，而在高地址区域则保留较大的空闲区。

BF（最佳适应算法）：最佳适应算法的“最佳”体现在每次为作业分配内存时，总是选择能够满足需求且容量最小的空闲分区。这种做法旨在避免资源的浪费，即避免将大的空闲区分配给小的作业。为了提高搜索效率，算法可以将所有空闲分区按照容量从小到大的顺序组织成链表。这样，首次搜索到的满足条件的空闲区将是最理想的选择。尽管在单个作业分配过程中最

佳适应算法看似最优，但从整体角度来看，它可能导致存储器中留下许多难以利用的小碎片，因为每次分配后剩余的空闲区总是最小的。

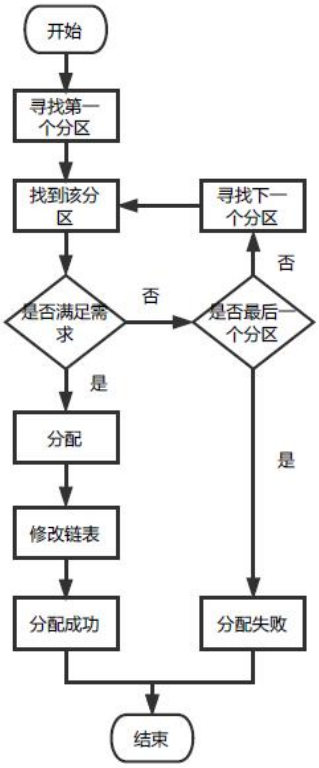
WF（最坏适应算法）：与最佳适应算法相对，最坏适应算法在为作业分配内存时，总是选择容量最大的空闲分区。这种方法的目的是减少内存碎片的产生，因为它能够使得剩下的空闲区不至于太小，以供将来可能需要更大整块内存空间的作业使用。最坏适应算法在处理需要大量连续内存空间的大型作业时特别有效，但可能会导致内存利用率降低，因为可能会存在一些空间的浪费。

四、主要的数据结构

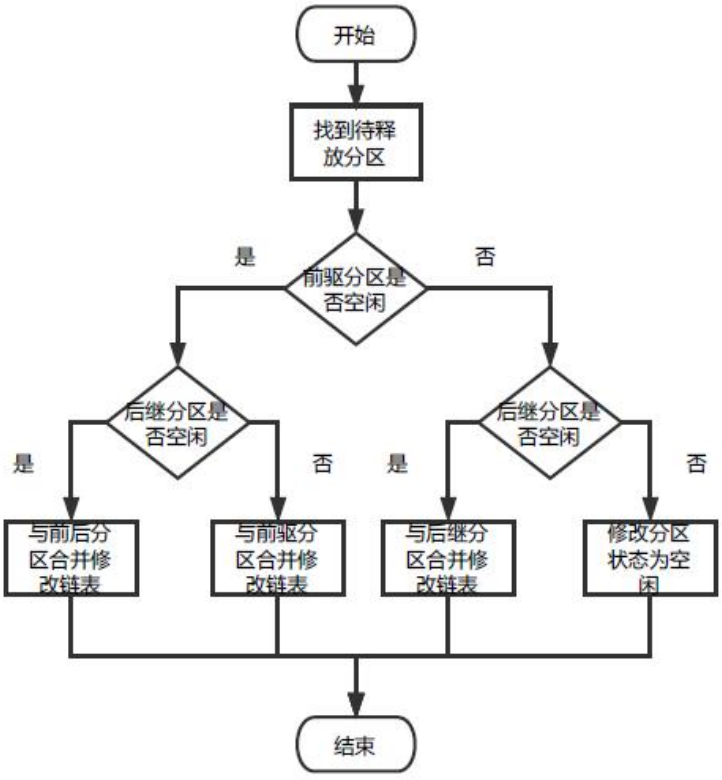
```
//定义空闲区说明表结构
typedef struct freearea
{
    long size;   //分区大小
    long address; //分区地址
    int state;    //状态
} ElemType;

// 定义线性表的双向链表存储结构
typedef struct DuLNode
{
    ElemType data;
    struct DuLNode *prior; //前趋指针
    struct DuLNode *next;  //后继指针
} DuLNode, *DuLinkList;

// 定义全局变量
DuLinkList block_first; //头结点
DuLinkList block_last;  //尾结点
```

五、算法流程图

5.1 首次适应算法（FF）流程图

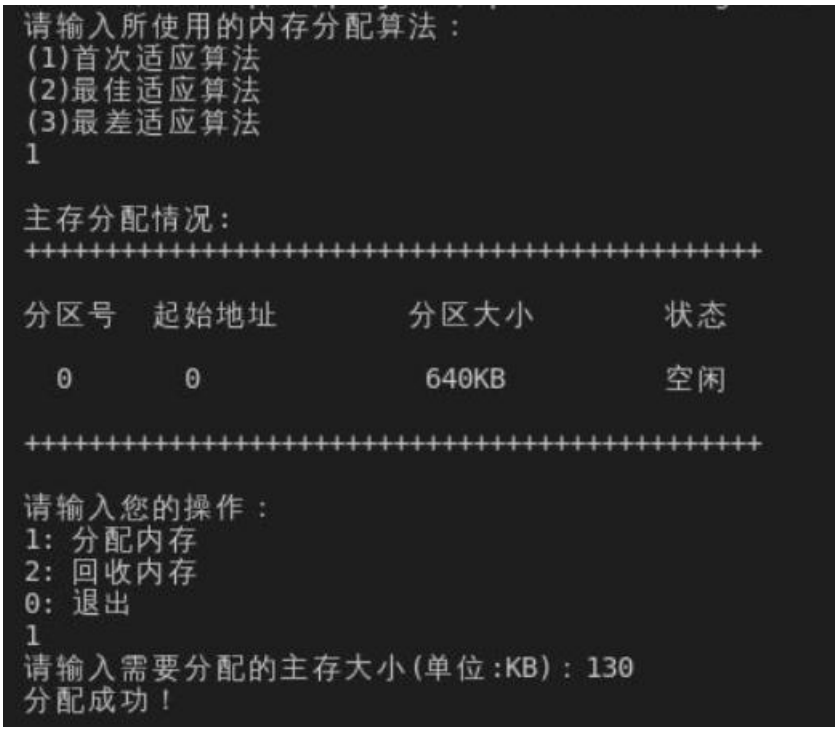


5.2 主存回收算法流程图



六、运行与测试

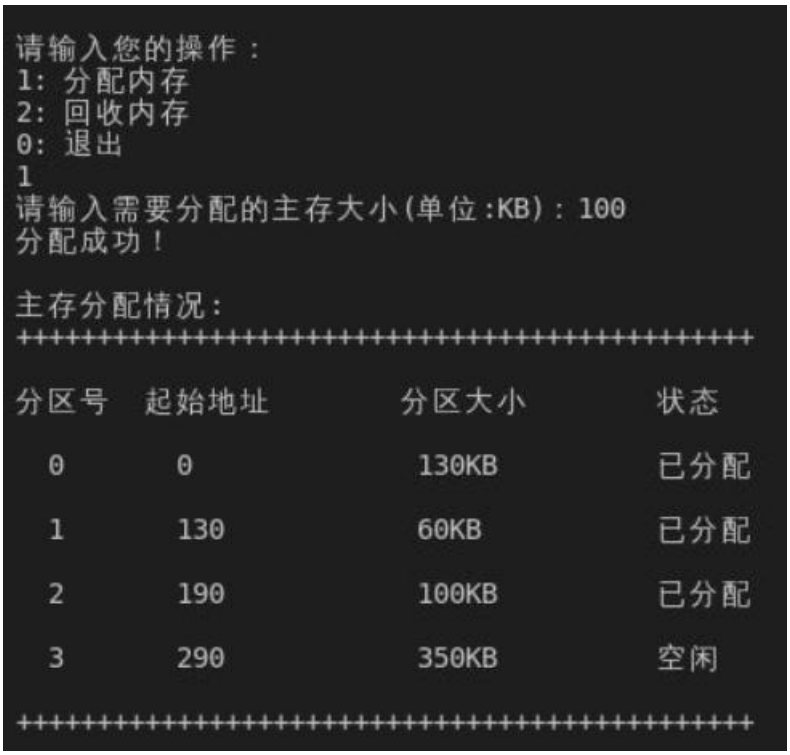
6.1 首次适应算法



中间省略不贴图

6.2 最佳适应算法

分配三次后情况



中间省略不贴图

6.3 最坏适应算法

```
请输入所使用的内存分配算法：
(1)首次适应算法
(2)最佳适应算法
(3)最差适应算法
3

主存分配情况：
++++++++++++++++++++++++++++++++++++++++++++++++

分区号   起始地址          分区大小          状态

  0        0                640KB           空闲

++++++++++++++++++++++++++++++++++++++++++++++++

请输入您的操作：
1: 分配内存
2: 回收内存
0: 退出
1
请输入需要分配的主存大小(单位:KB)：130
分配成功！

主存分配情况：
++++++++++++++++++++++++++++++++++++++++++++++++

分区号   起始地址          分区大小          状态

  0        0                130KB           已分配

  1        130              510KB           空闲

++++++++++++++++++++++++++++++++++++++++++++++++
```

中间省略不贴图

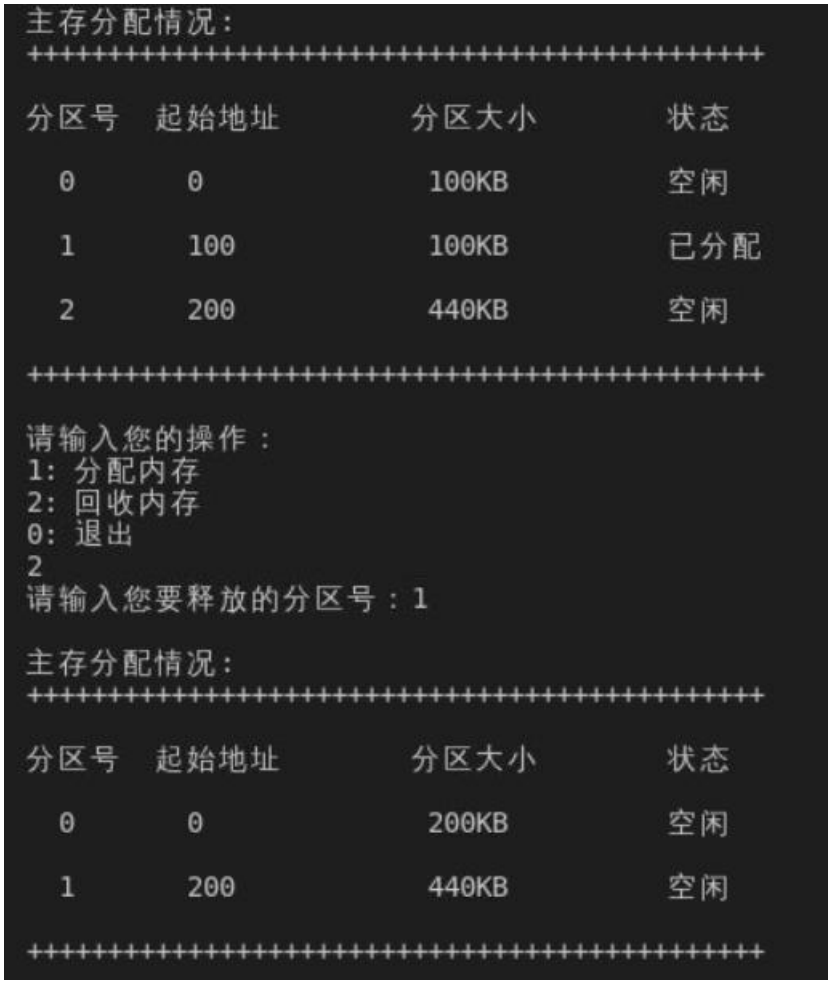
```
主存分配情况：
++++++++++++++++++++++++++++++++++++++++++++++++

分区号   起始地址          分区大小          状态

  0        0                140KB           已分配

  1        140              150KB           空闲

  2        290              200KB           已分配

  3        490              50KB            已分配

  4        540              100KB           空闲

++++++++++++++++++++++++++++++++++++++++++++++++
```

七、改进的方向

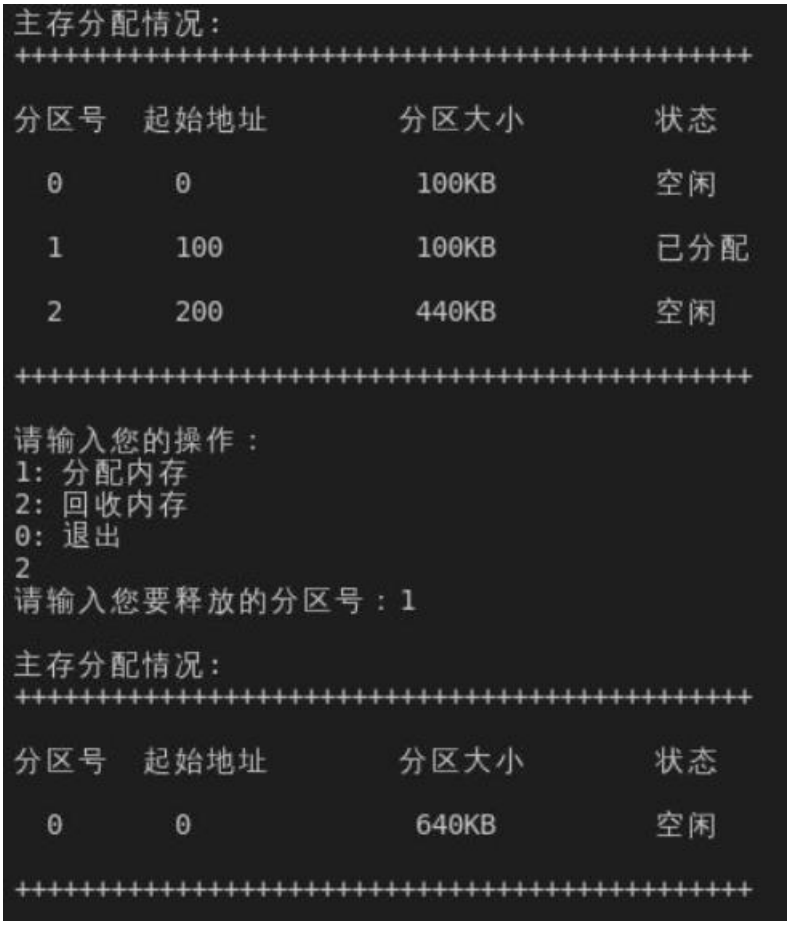
实验中使用的希冀平台提供的主存回收算法函数模版存在一定的问题，当待回收区块前后都存在空闲区块时，只能对前一个区块进行合并，而不能对后一个区块也进行合并，这导致了内存碎片的产生。

为了改进这一问题，可以对现有函数进行修缮。当待回收区块与前面的相邻空闲块合并完成后，不执行 `return OK` 返回函数，而是先设置一个回收成功的标志变量 `success`，然后继续顺序执行后面的代码块，以检查该区块后面是否还有空闲块相连，若有则与其进一步合并成更大的空闲区块。最后根据合并成功的情况，也即标志变量的取值进行回收成功 `OK` 或失败 `Error` 的返回。

原代码存在的问题示意图：



改进后的代码解决了问题：



实验三　请求调页存储管理方式的模拟

一、 实验目的

通过对页面、页表、地址转换和页面转换过程的模拟，加深对请求调页系统的原理和实现过程的理解。

二、 实验内容

（1） 用 C 语言模拟一作业的执行过程，该作业共有 320 条指令。假设每个页面中可以存放 10 条指令，即该作业的逻辑地址空间 为 32 页。分配给这个作业的内存块数为 4 个，目前它的所有页都还 没有调入内存。

（2） 模拟如下请求分页存储管理方式：

- 如果所访问的指令已在内存中，则显示其物理地址，并转下一 条指令。
- 如果所访问的指令还未装入内存，则发生缺页，此时须记录缺页的次数，并将相应页调入内存；如果调页时所有内存均已装入，则 需要按照置换算法进行页面置换。最后显示其物理地址，并转下一条指令。

（3） 作业中指令的执行次序按如下原则随机生成：

- 50%的概率顺序执行下一条指令。
- 25%的概率跳转执行前面的指令。
- 25%的概率跳转执行后面的指令。

具体实现办法：

1） 在[0, 319] 间随机选取一条指令开始启动执行，设其序号 为 m；

2） 顺序执行下一条指令，即序号为 m + 1 的指令；

3） 通过随机数跳转到[0, m]中的某条指令执行，设其序号为 m1；

4） 顺序执行下一条指令，即序号为 m1 + 1 的指令；

5） 通过随机数跳转到[m1 + 2, 319]中的某条指令处，设其序 号为 m2；

6） 顺序执行下一条指令，即序号为 m2 + 1 的指令；

7） 重复执行 2）-6）步，直到执行完 320 条指令为止。

（4） 编写程序分别使用 OPT、FIFO、LRU 置换算法模拟上述 流程，计算缺页率。

三、实现思路

在地址映射过程中，若在页面中发现所要访问的页面不在内存中，则产生缺页中断。当发生缺页中断时，如果操作系统内存中没有空闲页面，则操作系统必须在内存选择一个页面将其移出内存，以便为即将调入的页面让出空间。这一选择换出页面的算法即为页面置换算法。

本实验模拟实现了先进先出（FIFO）、最近最久未使用（LRU）、最佳（OPT）三种置换算法。

3.1 先进先出（FIFO）页面置换算法

置换最先调入内存的页面，即置换在内存中驻留时间最久的页面。按照进入内存的先后次序排列成队列，从队尾进入，从队首删除。但是该算法会淘汰经常访问的页面，不适应进程实际运行的规律，目前已经很少使用。流程图如下。

具体实现过程为：定义变量 ptr。一开始先预调页填满内存。在这一部分，ptr 指向下一个要存放的位置。之后继续执行剩下的指令。此时，ptr 表示队列最前面的位置，即最先进来的位置，也就是下一个要被替换的位置。ptr 用循环加，即模拟循环队列。

3.2 最近最久未使用（LRU）置换算法

置换最近一段时间以来最长时间未访问过的页面。根据程序局部性原理，刚被访问的页面，可能马上又要被访问；而较长时间内没有被访问的页面，可能最近不会被访问。 流程图如下。

LRU 算法普偏地适用于各种类型的程序，但是系统要时时刻刻对各页的访问历史情况加以记录和更新，开销太大，因此 LRU 算法必须要有硬件的支持。

具体实现如下：定义数组 ltu[],即 last_time_use 来记录该页最近被使用的时间。定义变量 ti 模拟时间的变化，每执行一次加一。本次实现未预调页，而是直接执行所有指令。若当前需要的页没在内存里，就寻找最近最少使用的页，也就是 ltu[]最小的页，即最近一次使用时间离现在最久的页，然后替换掉它。或者在内存还未满时，直接写入，这个以初始化内存里所有页为-1 来实现。若已经在内存里了，则只遍历内存内的页，把当前页的最近使用时间改一下即可。

3.3 最佳（OPT）页面置换算法

置换以后不再被访问，或者在将来最迟才回被访问的页面，缺页中断率最低。

但是该算法需要依据以后各业的使用情况，而当一个进程还未运行完成是，很难估计哪一个页面是以后不再使用或在最长时间以后才会用到的页面。所以该算法在实际情况中几乎是不能实现的。但该算法仍然有意义，作为很亮其他算法优劣的一个标准。流程图如下。

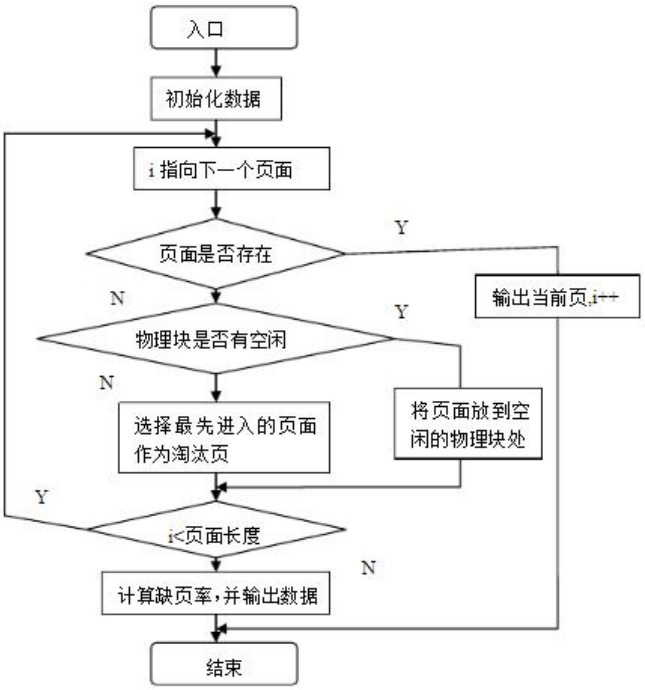
具体实现如下：定义数组 ntu[]，即 next_time_use 来记录下一次被使用的时间，即将来最快使用时间。初始化为-1。开始时预调页填满内存里的页。同样利用变量 ptr 来表示下一个要存放的位置从而控制预调页的过程。接着初始化 ntu 数组为-1。然后求出每一页下一次被使用的指令号，以此代替使用时间。如果所有剩下的序列都没有用该页时，则还是-1。这种值为-1 的页显然是最佳替换对象。然后执行所有剩下的指令。当该页不在内存里时，遍历 ntu 数组，遇到-1 的直接使用该页，没有则用 ntu[]值最大的，也就是最晚使用的。无论该页在不在内存里，因为这一次已经被使用了，所以都应该更新这个页的 ntu[]，只需往前看要执行的页流，记录下第一个遇到的该页即可。如果没有找到同样填入 -1 即可。

四、主要的数据结构

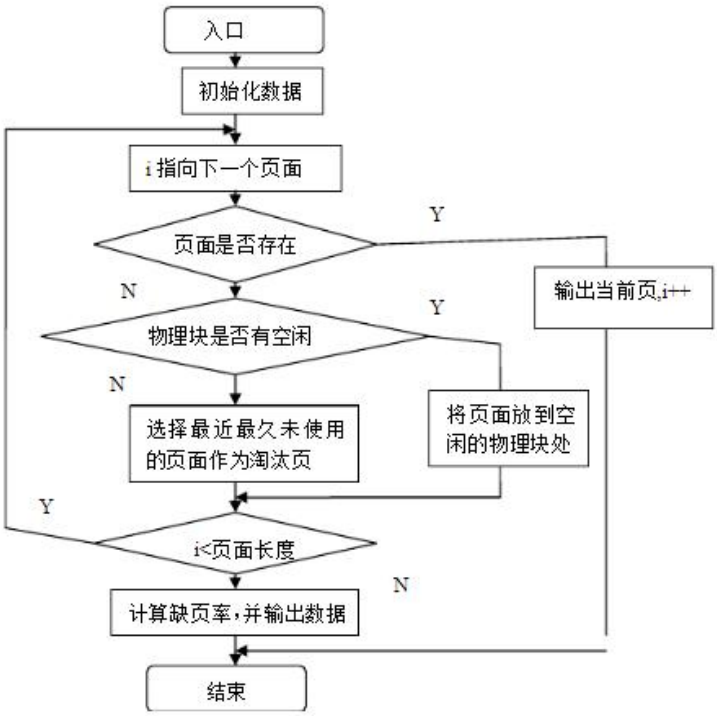
```
#define maxn 320          //序列个数
#define max (maxn + 20)       //数组大小
#define maxp (max / 10)   //最大页数

int inst[max];            //指令序列
int page[max];            //页地址流
int size;              //分配给该作业的物理块数

int in[maxp];             //记录某页是否在内存内
int pin[maxp];            //记录内存中的页面
```

五、算法流程图

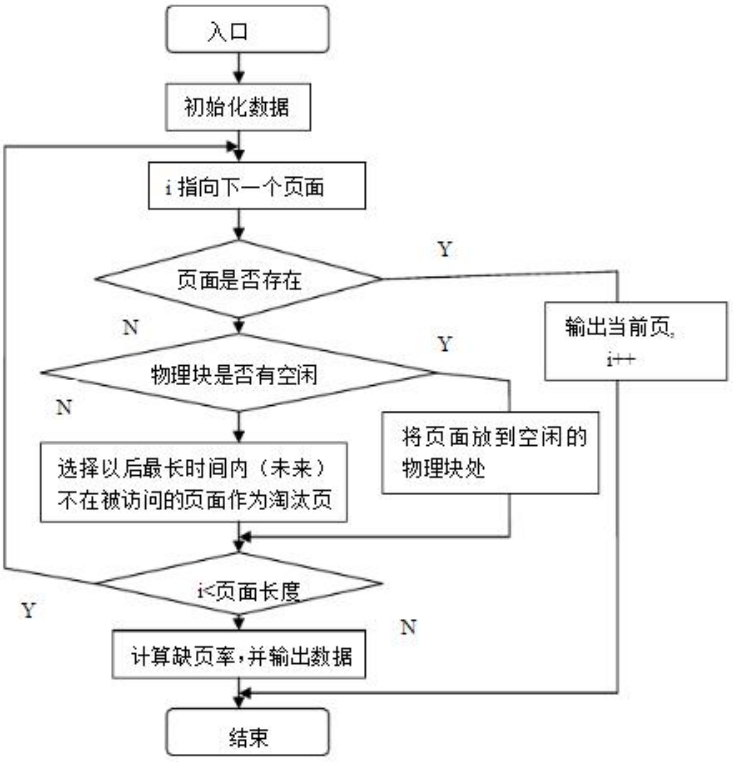
5.1 先进先出（FIFO）页面置换算法流程图



5.2 最近最久未使用（LRU）置换算法流程图



5.3 最佳（OPT）页面置换算法流程图



六、运行与测试

6.1 初始化（生成随机页地址流并设定内存大小为 4）

```
*****************************************
**           操作系统模拟实验          **
**            页式存储管理             **
*****************************************


1--create new instruction sequence      2--set memory page number(4 to 32)
3--solve by FIFO algorithm              4--solve by LRU algorithm
5--solve by OPT algorithm               0--exit
*********Please input Your choice:  1

New page address sequence is set OK!!!

1--create new instruction sequence      2--set memory page number(4 to 32)
3--solve by FIFO algorithm              4--solve by LRU algorithm
5--solve by OPT algorithm               0--exit
*********Please input Your choice:  2

Please input the size of memory page number:  4
```

6.2 选择使用 FIFO 算法进行模拟

```
1--create new instruction sequence      2--set memory page number(4 to 32)
3--solve by FIFO algorithm              4--solve by LRU algorithm
5--solve by OPT algorithm               0--exit
*********Please input Your choice:  3


By FIFO algorithm, the fault-page number is:  264
                   the hit ratio is :  0.18
```

6.3 选择使用 LRU 算法进行模拟

```
1--create new instruction sequence      2--set memory page number(4 to 32)
3--solve by FIFO algorithm              4--solve by LRU algorithm
5--solve by OPT algorithm               0--exit
*********Please input Your choice:  4


By LRU algorithm, the fault-page number is:  270
                  the hit ratio is :  0.16
```

6.4 选择使用 OPT 算法进行模拟

```
1--create new instruction sequence      2--set memory page number(4 to 32)
3--solve by FIFO algorithm              4--solve by LRU algorithm
5--solve by OPT algorithm               0--exit
*********Please input Your choice:  5


By OPT algorithm, the fault-page number is:  203
                  the hit ratio is :  0.37
```

6.5 附细节

使用 FIFO 算法时内存块中存储的页面情况（部分）

（指令序号：页面号，内存块 1 内存块 2 内存块 3 内存块 4）

```
3--solve by FIFO algorithm              4--solve by LRU algorithm
5--solve by OPT algorithm               0--exit
*********Please input Your choice:  3

1:      12,     12      0       0       0
2:      5,      12      5       0       0
3:      13,     12      5       13      0
4:      12,     12      5       13      0
5:      1,      12      5       13      1
6:      29,     29      5       13      1
7:      14,     29      14      13      1
8:      3,      29      14      3       1
9:      5,      29      14      3       5
10:     29,     29      14      3       5
11:     4,      4       14      3       5
12:     24,     4       24      3       5
13:     31,     4       24      31      5
14:     10,     4       24      31      10
15:     21,     21      24      31      10
16:     30,     21      30      31      10
17:     27,     21      30      27      10
18:     30,     21      30      27      10
19:     15,     21      30      27      15
20:     14,     14      30      27      15
21:     18,     14      18      27      15
22:     30,     14      18      30      15
23:     11,     14      18      30      11
24:     31,     31      18      30      11
25:     14,     31      14      30      11
26:     12,     31      14      12      11
27:     19,     31      14      12      19
28:     1,      1       14      12      19
29:     0,      1       0       12      19
30:     12,     1       0       12      19
31:     12,     1       0       12      19
32:     12,     1       0       12      19
33:     25,     1       0       25      19
34:     24,     1       0       25      24
```

使用 LRU 算法时内存块中存储的页面情况（部分）

（指令序号：页面号，内存块 1 内存块 2 内存块 3 内存块 4）

```
3--solve by FIFO algorithm              4--solve by LRU algorithm
5--solve by OPT algorithm               0--exit
*********Please input Your choice:  4

1:      12,     12      -1      -1      -1
2:      5,      12      5       -1      -1
3:      13,     12      5       13      -1
4:      12,     12      5       13      -1
5:      1,      12      5       13      1
6:      29,     12      29      13      1
7:      14,     12      29      14      1
8:      3,      3       29      14      1
9:      5,      3       29      14      5
10:     29,     3       29      14      5
11:     4,      3       29      4       5
12:     24,     24      29      4       5
13:     31,     24      29      4       31
14:     10,     24      10      4       31
15:     21,     24      10      21      31
16:     30,     30      10      21      31
17:     27,     30      10      21      27
18:     30,     30      10      21      27
19:     15,     30      15      21      27
20:     14,     30      15      14      27
21:     18,     30      15      14      18
22:     30,     30      15      14      18
23:     11,     30      11      14      18
24:     31,     30      11      31      18
25:     14,     30      11      31      14
26:     12,     12      11      31      14
27:     19,     12      19      31      14
28:     1,      12      19      1       14
29:     0,      12      19      1       0
30:     12,     12      19      1       0
31:     12,     12      19      1       0
32:     12,     12      19      1       0
33:     25,     12      25      1       0
34:     24,     12      25      24      0
```

使用 OPT 算法时内存块中存储的页面情况（部分）

（指令序号：页面号，内存块 1 内存块 2 内存块 3 内存块 4）

```
5--solve by OPT algorithm                0--exit
*********Please input Your choice:  5

1:      12,     12      0       0       0
2:      5,      12      5       0       0
3:      13,     12      5       13      0
4:      12,     12      5       13      0
5:      1,      12      5       13      1
6:      29,     12      5       29      1
7:      14,     12      5       29      14
8:      3,      3       5       29      14
9:      5,      3       5       29      14
10:     29,     3       5       29      14
11:     4,      3       4       29      14
12:     24,     3       24      29      14
13:     31,     31      24      29      14
14:     10,     31      24      10      14
15:     21,     31      24      21      14
16:     30,     31      24      30      14
17:     27,     31      27      30      14
18:     30,     31      27      30      14
19:     15,     31      15      30      14
20:     14,     31      15      30      14
21:     18,     31      18      30      14
22:     30,     31      18      30      14
23:     11,     31      11      30      14
24:     31,     31      11      30      14
25:     14,     31      11      30      14
26:     12,     31      11      30      12
27:     19,     31      19      30      12
28:     1,      31      19      1       12
29:     0,      0       19      1       12
30:     12,     0       19      1       12
31:     12,     0       19      1       12
32:     12,     0       19      1       12
33:     25,     0       19      1       25
34:     24,     0       19      1       24
35:     6,      0       19      1       6
```

中间省略

```
291:    8,      6       28      16      8
292:    17,     17      28      16      8
293:    14,     14      28      16      8
294:    28,     14      28      16      8
295:    27,     27      28      16      8
296:    11,     11      28      16      8
297:    16,     11      28      16      8
298:    20,     20      28      16      8
299:    8,      20      28      16      8
300:    13,     20      13      16      8
301:    10,     20      10      16      8
302:    6,      20      10      6       8
303:    25,     20      10      25      8
304:    26,     26      10      25      8
305:    1,      1       10      25      8
306:    21,     21      10      25      8
307:    23,     23      10      25      8
308:    8,      23      10      25      8
309:    25,     23      10      25      8
310:    10,     23      10      25      8
311:    10,     23      10      25      8
312:    17,     17      10      25      8
313:    15,     15      10      25      8
314:    12,     12      10      25      8
315:    20,     20      10      25      8
316:    19,     19      10      25      8
317:    4,      4       10      25      8
318:    22,     22      10      25      8
319:    31,     31      10      25      8
320:    16,     16      10      25      8

By OPT algorithm, the fault-page number is:  203
                  the hit ratio is :  0.37
```

七、改进的方向

7.1 对思考题的回答

（1） 如果增加分配给作业的内存块数，将会对作业运行过程中的缺页率产生什么影响？

答：当分配给作业的内存块数增加到一定程度前，作业运行过程中的缺页

率会随着分配的内存数的增加而减少。

（2） OPT、FIFO、LRU 哪种置换算法的性能好？为什么一般情况下 LRU 具有比 FIFO 更好的性能？

答：OPT 置换算法的性能最好，可以保证获得最低的缺页率，但它只是在理想状态下的理论算法，不能被实际应用。FIFO 置换算法之所以一般情况下较 LRU 差，是因为它所依据的条件是各个页面调入内存的时间，而页面调入的先后并不能很好地反映页面的使用情况。而 LRU 是根据页面调入内存后的使用情况做出决策的，选择最近最久未使用的页面予以淘汰，利用“最近的过去”作为“最近的将来”的近似，一般情况下具有比 FIFO 更好的性能。

7.2 改进的方向

教材中提到，LRU 算法要求有较多的硬件支持（因为需要记录和快速检索最久未使用的页面），使得其实现所需的成本较高，因而在实际应用中，大多采用 LRU 的近似算法，即 Clock 置换算法或称最近未用（NRU）算法，以减少实现的开销。同时，也存在使用链表数据结构（空闲页面链表与修改页面链表）进行页面置换选择的页面缓冲算法 PBA，并且能显著降低页面换进换出的频率。后续改进可以考虑增加这两种页面置换算法的实现，并测试其性能，与本实验中的三种算法进行比较。

实验四　简单文件系统的模拟实现

一、 实验目的

通过具体的文件存储空间的管理、文件的物理结构、目录结构和文件操作的实现，加深对文件系统内部功能和实现过程的理解。

二、 实验内容

（1） 在内存中开辟一块连续的空间作为一个虚拟磁盘，在其上实现一个简单的单用户文件系统。在退出这个文件系统时，应将该虚拟文件系统保存到磁盘上，以便下次可以再将它恢复到内存的虚拟磁盘空间中。

（2） 文件存储空间的分配可采用显式链接分配或其它方法。

（3） 磁盘空间的管理可采用位示图或其它方法。（注：如果采用位示图管理磁盘空间，并采用显式链接分配方式，可以将位示图合并到 FAT 中）。

（4） 文件目录结构采用多级目录结构。为了简单起见，可以不使用索引结点，其中的每个目录项必须包括文件名、物理地址、长度等信息，还可以包括文件的读写保护权限。

（5） 要求必须实现如下文件操作：

1. format：对文件存储器进行格式化，即按照文件系统的结构对虚拟磁盘空间进行布局，并在其上创建根目录以及用于管理文件存储空间等的数据结构。
2. mkdir：创建子目录。
3. rm：删除子目录。
4. dir：显示目录。
5. cd：更改当前目录。

6. create：创建文件。
7. delete：删除文件。

（6） 可选实现如下相关操作：

1. open：打开文件。
2. close：关闭文件。
3. write：写文件。
4. read：读文件。

三、 实现思路

本实验实现了一个树形多级目录结构的文件系统，通过模拟操作系统的文件管理功能，加深了对文件系统内部功能和实现过程的理解。实验的核心在于构建一个以根目录为起点的文件树，每个节点代表一个文件或目录，通过指针相互链接，形成层次分明的树状结构。

系统初始化时，创建根目录"/"，并以此为基础，实现了创建目录、创建文件、读取、写入、删除文件和目录等基本操作。路径解析功能允许用户通过绝对路径或相对路径访问文件系统中的任何位置，而目录遍历功能则可以列出当前目录下的所有内容。

为了有效管理内存，系统在删除文件或目录时，在希冀平台提供的代码框架的基础上，我额外编写了用于回收内存的递归函数，在删除非空目录时能够递归释放其所有子节点的内存，从而避免了内存泄漏。

用户交互通过命令行实现，用户可以输入不同的命令来执行相应的文件系统操作，直至退出系统。

具体实现代码细节如下：

1. 树形结构的建立：文件系统使用树形结构来组织文件和目录。每个节点代表一个文件或目录，根节点"/"代表整个文件系统的根目录。

2. 节点表示：系统中的每个文件或目录都用一个 filenode 结构体来表示，包含文件名、类型（文件或目录）、内容、以及指向父节点、子节点、前驱节点

和后继节点的指针。

3. 目录操作：通过 mkdir 和 rm 命令创建和删除目录。创建目录时，会在父目录的子节点链表中添加新的目录节点。删除目录时，会从父目录的子节点链表中移除相应节点，并释放相关资源，这一部分类似于链表的删除操作。

4. 文件操作：通过 create、read、write 和 del 命令来创建、读取、写入和删除文件。文件的创建和删除与目录类似，且更为简单。

5. 路径解析：使用 findpath 函数解析用户输入的路径，支持绝对路径和相对路径（如".."表示父目录，"."表示当前目录）。路径解析过程中，会更新当前工作目录的指针 recent 到目标节点。

6. 目录遍历：dir 命令用于列出当前目录下的所有文件和目录，通过遍历当前目录节点的子节点链表实现。

7. 当前工作目录管理：通过 cd 命令实现更改当前目录的功能，更新 recent 指针到新的目录节点，并相应地更新路径 path。

8. 内存管理：为避免内存泄露，在删除节点时，通过额外添加的 recycle_memory 函数递归释放节点及其子节点占用的内存。

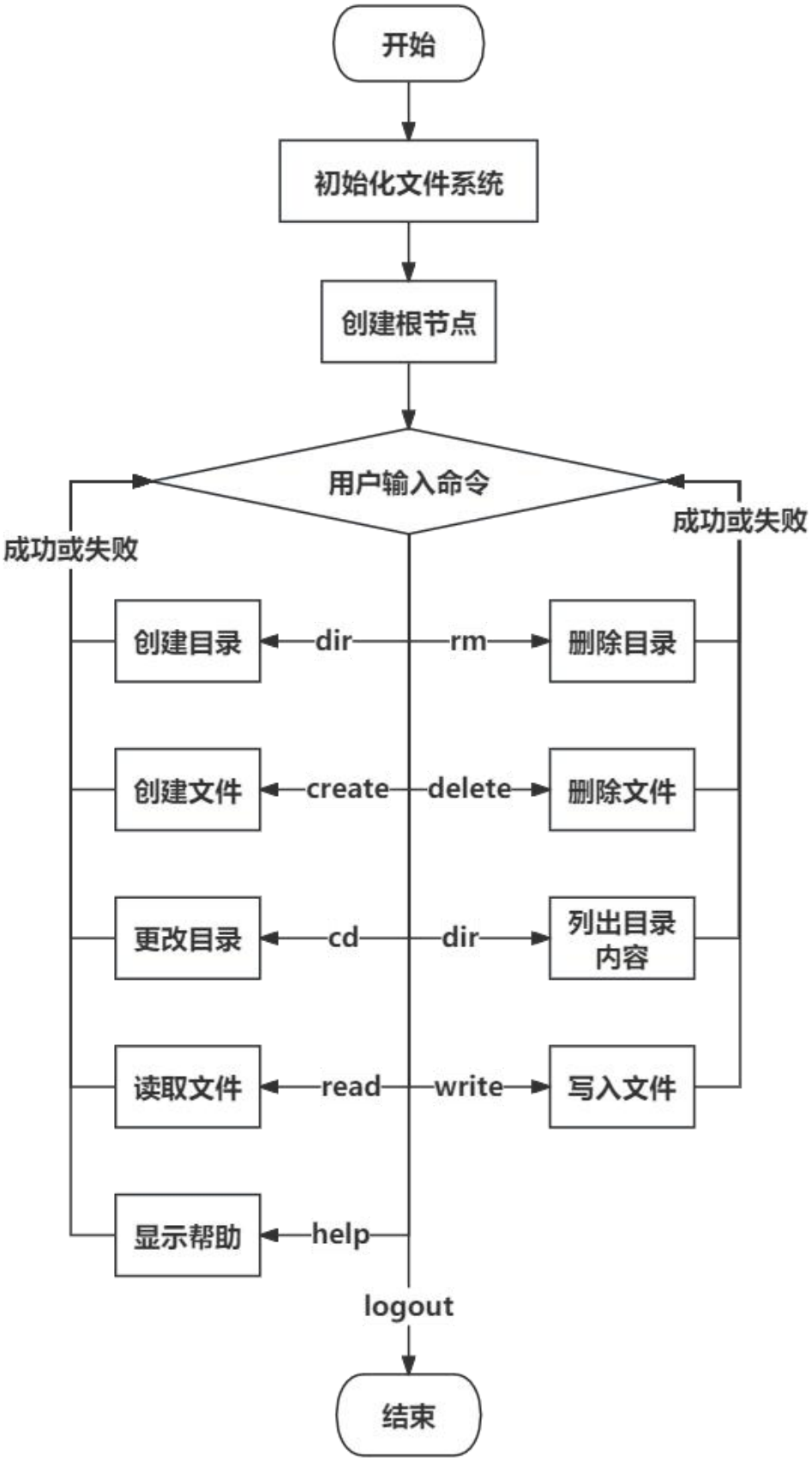
9. 用户交互：程序通过 run 函数实现用户交互循环，接收用户命令并调用相应的函数执行操作，直到用户执行 logout 命令退出系统。

综上所述，通过这种实现，本程序能够通过维护一个用孩子兄弟表示法存储的多叉树，来实现对文件管理中树形目录结构的模拟，从而实现对文件和目录的各种基本操作。并通过命令行提供了用户交互功能，使得文件和目录的管理更加直观和有序。

四、 主要的数据结构

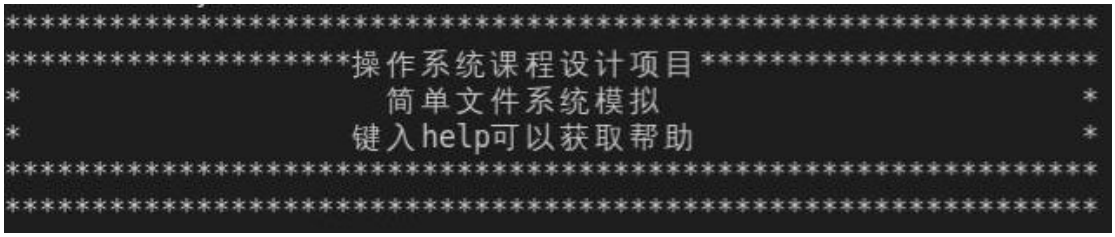
```
#define FILENAME_LENGTH 10  // 文件名称长度
#define COMMAND_LENGTH 10   // 命令行长度
#define PATH_LENGTH 30      // 参数长度

struct filenode              //定义多叉树节点数据类型
{
    char filename[FILENAME_LENGTH];  //文件或目录名
    int isdir;                       //标识位，为 1 时表示该节点存储目录
    char content[255];               //文件内容
    struct filenode *parent;         //父节点指针
    struct filenode *child;          //首个孩子节点指针
    struct filenode *prev;           //前驱兄弟节点指针
    struct filenode *next;           //后继兄弟节点指针
};

//功能函数声明
struct filenode *initnode(char filename[], int isdir);
void createroot();
int run();
int findpath(char *topath);
void help();
int mkdir();
int create();
int read();
int write();
int del();
int rm();
int cd();
int dir();

//根节点等全局变量
struct filenode *root, *recent, *temp, *ttemp, *temp_child;
char path[PATH_LENGTH], command[COMMAND_LENGTH], temppath[PATH_LENGTH],
recentpath[PATH_LENGTH];
```

五、程序流程图

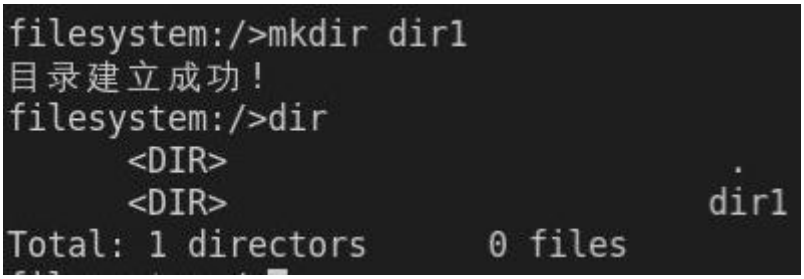

开始
初始化文件系统
创建根节点
用户输入命令
成功或失败
成功或失败
创建目录
dir
rm
删除目录
创建文件
create
delete
删除文件
更改目录
cd
dir
列出目录内容
读取文件
read
write
写入文件
显示帮助
help
logout
结束

六、运行与测试

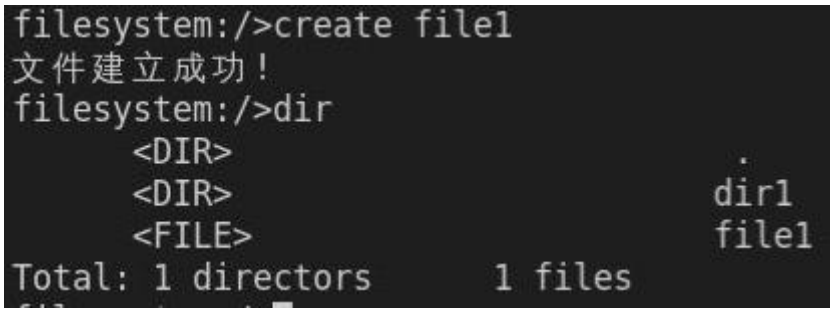
6.1 初始界面



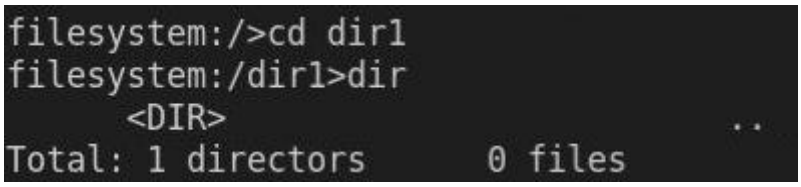
6.2 创建目录 mkdir



6.3 创建文件 create



6.4 通过 cd 命令进入刚创建的目录 dir1



6.5 创建多级目录和文件

```
filesystem:/dir1>create file2
文件创建成功!
filesystem:/dir1>mkdir dir2
目录建立成功!
filesystem:/dir1>dir
      <DIR>                          ..
      <FILE>                        file2
      <DIR>                         dir2
Total: 2 directors      1 files
filesystem:/dir1>cd dir2
filesystem:/dir1/dir2>mkdir dir3
目录建立成功!
filesystem:/dir1/dir2>create file3
文件建立成功!
filesystem:/dir1/dir2>dir
      <DIR>                          ..
      <DIR>                         dir3
      <FILE>                        file3
Total: 2 directors      1 files
filesystem:/dir1/dir2>
```

6.6 返回上级根目录准备读写

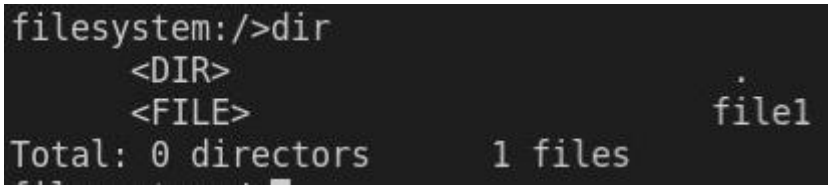
```
filesystem:/dir1/dir2>cd ..
filesystem:/dir1>cd ..
```

6.7 读写文件 write/read

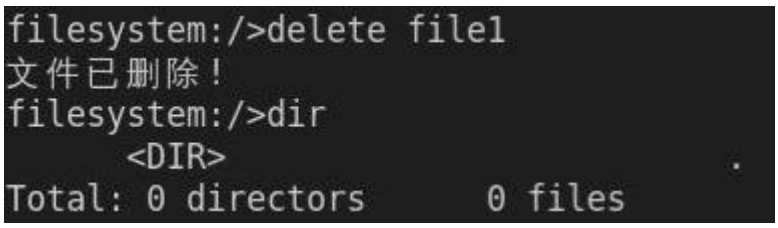
```
filesystem:/>write file1
gdut_3122008883_cyq
文件写入成功!
filesystem:/>read file1
gdut_3122008883_cyq
```

6.8 删除目录 rm 和递归回收内存函数 recycle_memory

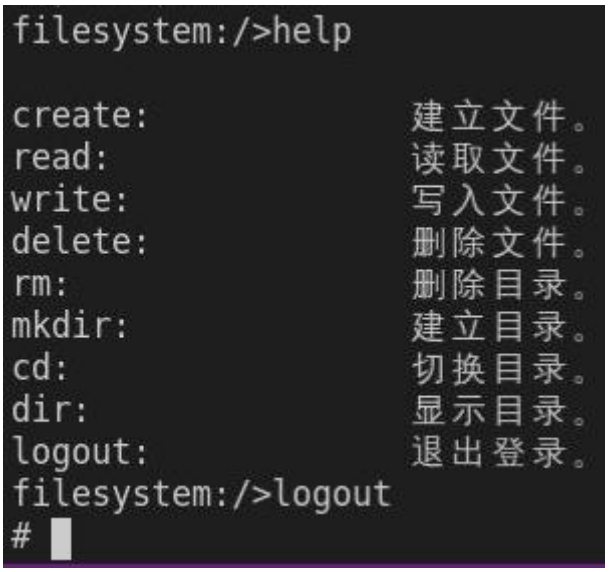
```
filesystem:/>rm dir1
Deleting dir3 ...
Deleting file3 ...
Deleting dir2 ...
Deleting file2 ...
Deleting dir1 ...
目录已删除!
```



6.9 删除文件 delete



6.10 帮助命令 help 与退出命令 logout



七、改进的方向

对于一个已不再需要的目录，如何删除其目录项，需要视情况而定。如果所要删除的目录是空的，即在该目录中已不再有任何文件，就可简单地将该目录项删除。如果要删除的目录不空，即其中尚有几个文件或子目录，则可以采用“不删除非空目录”和“可删除非空目录”两种方法处理。

本实验使用的希冀平台提供的实验代码架构中，采取了可删除非空目录的策略，但在删除时没有提供对应的处理函数，仅仅简单地调用 free 函数将该目录项节点释放，而其子节点项则未作处理。这样虽然能够通过测试样例，但极大概率出现内存泄漏的情况。

应对这种情况，在采用可删除非空目录策略的情况下，应该额外添加编写一

个释放非空目录子节点的函数，通过递归遍历该子树的方式，逐个释放其中的节点。以下是我对该功能函数的实现。

```
void recycle_memory(struct filenode *now)
{
    struct filenode *p, *q;
    p=now->child;
    while(p!=NULL)
    {
        q=p->next;
        recycle_memory(p);
        p=q;
    }
    printf("Deleting %s ...\n", now->filename);
    free(now);
}
```

对应地，应在 rm 函数中将 free 函数的调用改成对该函数 recycle_memory 的调用。（见选中部分）

```
int rm()
{
    char filename[FILENAME_LENGTH];
    scanf("%s", filename);
    temp = (struct filenode *)malloc(sizeof(struct filenode));

    if (recent->child)
    {…
    }
    else
    {…
    }

    if (temp->parent == NULL)
    {…
    }
    else
    {…
    }
    recycle_memory(temp);
    printf("目录已删除!\n");
    return 0;
}
```

此外，本次实验中未包含 open 和 close 操作的部分。这也是后续改进的一个方向。

本实验中文件采用无结构方式组织，仅被看作一个字符流。而在有结构文件中，文件由若干个相关记录组成，同时记录中又可以包含多个数据项。

后续改进添加时可以考虑修改代码中的数据结构类型定义，在节点数据类型 filenode 中添加文件属性（文件类型、长度等）和用于存储记录的数组或链表，采用顺序文件或索引文件等方式进行文件中多个记录的组织。

实现细节上，可在调用 open 函数打开文件时，展示文件中各记录的组织情况，并允许 write 和 read 函数执行；在调用 close 函数关闭文件时，则返回到文件所在的目录一级，并且此时不再允许 read/write 函数对文件的读写操作。