页面置换算法实验报告

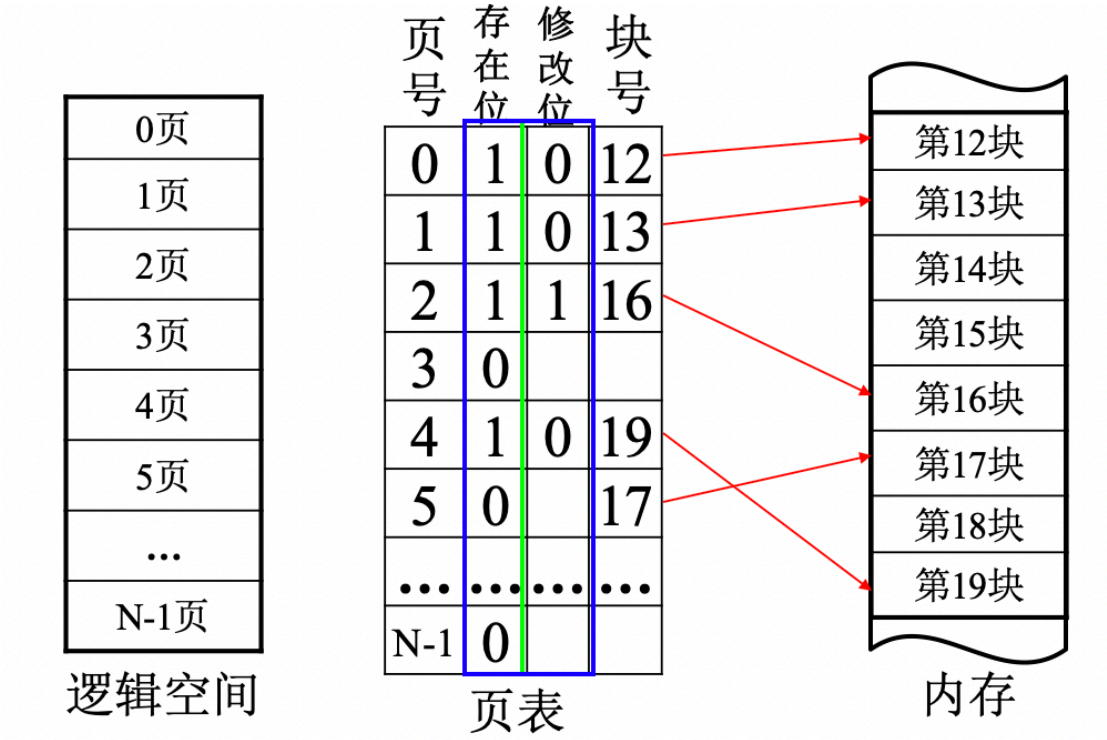
16281153 张天悦

一 实验目的

设计和实现最佳置换算法、先进先出置换算法、最近最久未使用置换算法、页面缓冲置换算法 ;通过页面访问序列随机发生器实现对上述算法的测试及性能比较。

二 背景知识

1.请求分页虚拟内存管理



2.工作集

多数程序都显示出高度的局部性，也就是说，在一个时间段内，一组页面被反复引用。这组被反复引用的页面随着时间的推移，其成员也会发生变化。有时这种变化是剧烈的，有时这种变化则是渐进的。我们把这组页面的集合称为工作集

3.缺页率

缺页率 = 缺页中断次数/页面访问次数

4.五种替换方法

最佳置换算法

最佳置换算法

基本思想：

在发生页面替换时，被替换的对象应该满足，在以后的页面访问中，该对象不会再次被访问或者较晚被访问。

评价：

是一种理想化算法，具有最好性能（对于固定分配页面方式，本法可保证获得最低的缺页率），但实际上却难于实现，故主要用于算法评价参照。

先进先出置换算法

基本思想：

先进先出置换算法的主要思想是，在发生页面替换时，被替换的对象应该是最早进入内存的。

评价：

简单直观，但不符合进程实际运行规律，性能较差，故实际应用极少。

最近最久未使用置换算法

主要思想：

在发生页面替换时，被替换的页面应该满足，在之前的访问队列中，该对象截止目前未被访问的时间最长。

评价：

适用于各种类型的程序，性能较好，但需要较多的硬件支持

改进型 Clock 置换算法

主要思想：

在每次页面替换时，总是尽可能地先替换掉既未被访问又未被修改的页面。

评价：

与简单 Clock 算法相比，可减少磁盘的 I/O 操作次数，但淘汰页的选择可能经历多次扫描，故实现算法自身的开销增大。

页面缓冲算法 PBA

基本思想：

设立空闲页面链表和已修改页面链表采用可变分配和基于先进先出的局部置换策略，并规定被淘汰页先不做物理移动，而是依据是否修改分别挂到空闲页面链表或已修改页面链表的末尾,空闲页面链表同时用于物理块分配,当已修改页面链表达到一定长度如 Z 个页面时，一起将所有已修改页面写回磁盘，故可显著减少磁盘 I/O 操作次数。1.确定虚拟内存的尺寸 N，工作集的起始位置 p，工作集中包含的页数 e，工作集移动率 m（每处理 m 个页面访问则将起始位置 p +1），以及一个范围在 0 和 1 之间的值 t；

三 实验要求

1.页面访问序列随机生成说明

符合局部访问特性的随机生成算法：

1.确定虚拟内存的尺寸 N，工作集的起始位置 p，工作集中包含的页数 e，工作集移动率 m（每处理 m 个页面访问则将起始位置 p +1），以及一个范围在 0 和 1 之间的值 t；

2.生成 m 个取值范围在 p 和 p + e 间的随机数，并记录到页面访问序列串中；

3.生成一个随机数 r，0 ≤ r ≤ 1；

4.如果 r < t，则为 p 生成一个新值，否则 p = (p + 1) mod N；

5.如果想继续加大页面访问序列串的长度，请返回第 2 步，否则结束。

2.内存

模拟的虚拟内存的地址为 16 位，页面大小为 1K

模拟的物理内存有 32K

3.页面

页表用整数数组或结构数组来表示

页面访问序列串是一个整数序列，整数的取值范围为 0 到 N - 1。页面访问序列串中的每个元素 p 表示对页面 p 的一次访问

四　实验过程

1.代码：

```
#include<iostream>
#include<windows.h>
#include<time.h>
#include<malloc.h>
#include<conio.h>
using namespace std;
#define ref_size 20
#define phy_size 3
int ref[ref_size];
float interrupt[6]={0.0};
//int ref[ref_size]={0};
int phy[phy_size];

void set_rand_num()//产生具有局部特性的随机数列
{
    cout<<"页面访问序列："<<endl;
    int p=12;
    int e=4;
    int m=4;
```

```
    int i=0;
    int j=0;
    int n=0;
    double t=0.6;
    int temp;
    for(i=0;i<m;i++,j++)
    {
        Sleep(1000*i);
        srand(time(NULL));
        temp=rand()%e+p;
        ref[j]=temp;
        cout<<ref[j]<<"    ";
    }
    for(n=0;n<4;n++)
    {
        Sleep(1000*n);
        srand(time(NULL));
        double r=(double)(rand()%10)/10.0;
        //cout<<r<<endl;
        if(r<t) p=p+int(10*r);
        else
            p=(p+1)%20;
        for(i=0;i<m;i++,j++)
        {
            Sleep(1000*i);
            srand(time(NULL));
            temp=rand()%e+p;
            ref[j]=temp;
            cout<<ref[j]<<"    ";
        }
    }
    cout<<endl;
}
```

```
typedef struct QNode    //定义队列数据结构
{
    int data;
    struct QNode *next;
}QNode,*QueuePtr;
typedef struct
{
    QueuePtr front;          //头指针
    QueuePtr rear;           //尾指针
}LinkQueue;
//定义链表结点
typedef struct LNode     //定义循环链表数据结构
{
    int data;
    int flag;          //访问位
    int modify;            //修改位
    struct LNode *next;
}LNode,*LinkList;
```

对循环链表的一些操作

```
int CreatList(LinkList &L)//创建循环带有头结点的链表
{
    L=(LinkList)malloc(sizeof(LNode));
    if(!L) exit(-1);
    L->next=L;
    L->flag=0;
    return 1;
}
```

int Exchange_LNode(LinkList &L,int e,int i)//将链表 L 中序号为 i 的结点替换为内容为 e 的结点

```
{
    if(L->next==L) exit(-1);
    LinkList p,q;
    int j=0;
    p=(LinkList)malloc(sizeof(LNode));
    q=(LinkList)malloc(sizeof(LNode));
    q->data=e;
    p=L;
    for(j=0;j<i;j++)//使 p 为待更换结点的前一个结点，故应保证，删除第一个非头结点时 i=0，以此类推
        p=p->next;
    q->next=p->next->next;
    p->next=q;
    q->flag=1;      //设置新结点的访问位为 1
    q->modify=0;   //设置新结点的修改位为 0
    return 1;
}

int Insert_LNode(LinkList &L,int e)//在循环链表中插入新的结点，从 L 头结点开始依次向后插入
{
    LinkList p,q;
    p=(LinkList)malloc(sizeof(LNode));
    q=(LinkList)malloc(sizeof(LNode));
    q->data=e;
    q->flag=1;      //设置新结点的访问位为 1
    q->modify=0;   //设置新结点的修改位为 0
    p=L;
    while(p->next!=L)
    {
        p=p->next;
    }
    p->next=q;
```

```
    q->next=L;
    return 1;
}

bool Search_LinkList(LinkList &L,int e,int &i)//找到链表 L 中内容为 e 的结点，并用 i 返回其位置，i=1 表示第一个非头结点，依次类推
{
    i=1;
    if(L->next==L) exit(-1);
    LinkList p;
    p=(LinkList)malloc(sizeof(LNode));
    if(!p) exit(-1);
    p=L->next;      //p 指向链表的第一个结点（非头结点）
    while(p!=L && p->data!=e)
    {
        p=p->next;
        i++;
    }
    if(p==L)    //没有找到符合要求的结点
        return false;
    return true;
}

void Search_LL_Flag(LinkList &L,int &i)//用 i 返回第一个 flag 为 0 的结点的位置,i=1 表示第一个非头结点，以此类推
{
    i=1;
    LinkList p;
    p=(LinkList)malloc(sizeof(LNode));
    if(!p) exit(-1);
    p=L->next;
    while(p->flag!=0)
    {
```

```
            p->flag=0;        //修改访问标志位为 0
            p=p->next;
            if(p==L)      //跳过头结点
                p=p->next;
            i++;
            if(i==4)      //跳过头结点
                i=1;
        }
        //return 1;
}

void Set_LL_Flag(LinkList &L,int i)//设置链表 L 中的序号为 i 的结点的 flag 标志为
1；
{
        LinkList p;
        p=(LinkList)malloc(sizeof(LNode));
        if(!p) exit(-1);
        p=L->next;
        if(i==1)
            p->flag=1;
        if(i==2)
        {
            p=p->next;
            p->flag=1;
        }
        if(i==3)
        {
            p=p->next;
            p=p->next;
            p->flag=1;
        }
}
```

```
int Search_LL_ModifyClock(LinkList &L,int &modify_num)//找到改进的 CLOCK 算法所需要淘汰的页，用 modify_num 返回其位置
{
    modify_num=1;
    if(L->next==L) exit(-1);
    LinkList p;
    p=(LinkList)malloc(sizeof(LNode));
    if(!p) exit(-1);
    p=L->next;      //p 指向链表的第一个结点（非头结点）
    while(p!=L)     //第一轮扫描 A=0 并且 M=0 的结点
    {
        if(p->flag==0 && p->modify==0)
            break;  //找到
        p=p->next;
        modify_num++;
    }
    if(p==L)
    {
        modify_num=1;
        p=L->next;
        while(p!=L)     //第二轮扫描 A=0 并且 M=1 的结点，同时修改访问过的结点的访问位为 0
        {
            if(p->flag!=0)
                p->flag=0;
            else if(p->modify==1)
                break;
            p=p->next;
            modify_num++;

        }
    }
    if(p==L)
```

```
    {
        modify_num=1;
        p=L->next;
        while(p!=L)       //第三轮扫描 A=0 并且 M=0 的结点
        {
            if(p->flag==0 && p->modify==0)
                break;
            p=p->next;
            modify_num++;
        }
        if(p==L)
        {
            modify_num=1;
            p=L->next;
            while(p!=L)       //第四轮扫描 A=0 并且 M=1 的结点
            {
                if(p->flag!=0)
                    p->flag=0;
                else if(p->modify==1)
                    break;
                p=p->next;
                modify_num++;

            }
        }

    }
    return 1;
}

void Set_LL_modify(LinkList &L,int i)  //设置链表 L 中的序号为 i 的结点的 modify 标志为 1；
{
```

```
    LinkList p;
    p=(LinkList)malloc(sizeof(LNode));
    if(!p) exit(-1);
    p=L->next;
    if(i==0)
        p->modify=1;
    if(i==1)
    {
        p=p->next;
        p->modify=1;
    }
    if(i==2)
    {
        p=p->next;
        p=p->next;
        p->modify=1;
    }
}

int DestroyLinkList(LinkList &L)        //删除链表，并释放链表空间
{
    LinkList p,q;
    p=(LinkList)malloc(sizeof(LNode));
    if(!p) exit(-1);
    q=(LinkList)malloc(sizeof(LNode));
    if(!q) exit(-1);
    p=L->next;
    while(p!=L)
    {
        q=p->next;
        free(p);
        p=q;
    }
```

```
    free(q);
    return 1;
```
对队列操作
```
int InitQueue(LinkQueue &Q)          //队列初始化
{
    Q.front=Q.rear=(QueuePtr)malloc(sizeof(QNode));
    if(!Q.front) exit(-1);
    Q.front->next=NULL;
    return 1;
}

int EnQueue(LinkQueue &Q,int e)          //插入元素 e 为 Q 的新的队尾元素
{
    QueuePtr p;
    p=(QueuePtr)malloc(sizeof(QNode));
    if(!p) exit(-1);
    p->data=e;
    p->next=NULL;
    Q.rear->next=p;
    Q.rear=p;
    return 1;
}

int DeQueue(LinkQueue &Q,int &e) //若队列不空，则删除 Q 的队头元素，用 e
返回其值
{
    if(Q.front==Q.rear) return -1;
    QueuePtr p;
    p=(QueuePtr)malloc(sizeof(QNode));
    p=Q.front->next;
    e=p->data;
    Q.front->next=p->next;
    if(Q.rear==p)
```

```
        Q.rear=Q.front;
    free(p);
    return 1;
}

bool SearchQueue(LinkQueue &Q,int e,int &i)    //寻找队列 Q 中结点 data 域等于 e 的结点，并用 i 返回其在 Q 中的位置
{
    i=1;
    if(Q.front==Q.rear) exit(-1);
    QueuePtr p;
    p=(QueuePtr)malloc(sizeof(QNode));
    if(!p) exit(-1);
    p=Q.front->next;          //p 指向队列的第一个节点（非头结点）
    while(p!=NULL && p->data!=e)
    {
        p=p->next;
        i++;
    }
    if(!p)
        return false;
    return true;
}

int DelMid_Queue(LinkQueue &Q,int &e)          //删除 Q 的中间元素，并用 e 返回其值
{
    if(Q.front==Q.rear) return -1;
    QueuePtr p;
    p=(QueuePtr)malloc(sizeof(QNode));
    if(!p) exit(-1);
    p=Q.front->next;
    e=p->next->data;
```

```
    p->next=p->next->next;
    return 1;
}

int DestroyQueue(LinkQueue &Q)              //删除队列并释放空间
{
    while(Q.front)
    {
        Q.rear=Q.front->next;
        free(Q.front);
        Q.front=Q.rear;
    }
    return 1;
}
int max1(int a,int b, int c)     //返回 a,b,c 中的最大值
{
    if(a<b) a=b;
    if(a<c) a=c;
    return a;
}
int getnum(int a,int b)              //用 b 返回元素 a 在被引用数列中的下一个位置
{
    for(;b<ref_size;b++)
    {
        if(a==ref[b])
            break;
    }
    return b;
}
void ORA()            //////////////最佳置换算法
{
    cout<<"\n**********************     最     佳     置     换     算     法
************************"<<endl;
```

```
int i,j;
int num_0,num_1,num_2,num_max;
int interrupt_num=0;
//num_0=num_1=num_2=0;
for(i=0;i < phy_size;i++) //前三个数进内存
    phy[i]=ref[i];
for(i=0;i<phy_size;i++)        //输出最初的三个数
    cout<<phy[i]<<"\t";
cout<<endl;
for(j=phy_size;j<ref_size;j++)
{

    if(!(ref[j]==phy[0] || ref[j]==phy[1] || ref[j]==phy[2]))  //若产生缺页中断,选择最久不会被使用的页被替换
    {
        num_0=getnum(phy[0],j+1);
        num_1=getnum(phy[1],j+1);
        num_2=getnum(phy[2],j+1);
        num_max=max1(num_0,num_1,num_2);
        if(num_0==num_max)
            phy[0]=ref[j];
        else
            if(num_1==num_max)
                phy[1]=ref[j];
            else
                if(num_2==num_max)
                    phy[2]=ref[j];
        interrupt_num++;

    }
    cout<<"进入页："<<ref[j]<<endl;
    for(i=0;i<phy_size;i++)        //输出内存状态
        cout<<phy[i]<<"\t";
```

```
            cout<<endl<<endl;
        }
        cout<<"最佳置换算法缺页中断次数："<<interrupt_num<<endl;
        interrupt[0]=((float)interrupt_num/20.0)*100.0;
}
void RAND()                 //////////////随机置换算法
{
        cout<<"\n**********************      随      机      置      换      算      法
**********************"<<endl;
        int i,j,temp;
        int interrupt_num=0;
        //num_0=num_1=num_2=0;
        Sleep(1000);
        srand(time(NULL));          //设置时间种子
        for(i=0;i < phy_size;i++)
            phy[i]=ref[i];
        for(i=0;i<phy_size;i++)
            cout<<phy[i]<<"\t";
        cout<<endl;
        for(j=phy_size;j<ref_size;j++)
        {

            if(!(ref[j]==phy[0] || ref[j]==phy[1] || ref[j]==phy[2]))  //产生缺页中断，随
机选择页被替换
            {
                temp=rand()%3;
                //cout<<temp<<endl;
                phy[temp]=ref[j];
                interrupt_num++;

            }
            cout<<"进入页："<<ref[j]<<endl;
            for(i=0;i<phy_size;i++)
```

```
            cout<<phy[i]<<"\t";
        cout<<endl<<endl;
    }
    SetConsoleTextAttribute(GetStdHandle(STD_OUTPUT_HANDLE),FOREGROU
ND_INTENSITY | FOREGROUND_GREEN);
    cout<<"随机置换算法缺页中断次数："<<interrupt_num<<endl;
    interrupt[1]=((float)interrupt_num/20.0)*100.0;
}

void FIFO()
{
    cout<<"\n**********  先  进  先  出  置  换  算  法
**********"<<endl;
    LinkQueue L;
    QueuePtr p;
    int i,j,e,m;
    int interrupt_num=0;
    InitQueue(L);
    for(i=0;i<phy_size;i++)
    {
        EnQueue(L,ref[i]);
    }
    p=(QueuePtr)malloc(sizeof(QNode));
    p=L.front->next;
    for(j=0;p!=NULL && j<phy_size;j++)//前三个数进内存
    {
        cout<<p->data<<"\t";
        p=p->next;
    }
    cout<<endl;
    for(i=phy_size;i<ref_size;i++)
    {
```

```
        if(!SearchQueue(L,ref[i],m))  //产生缺页中断，选择最先进入的页被替换
        {
            DeQueue(L,e);
            //cout<<e<<endl;
            EnQueue(L,ref[i]);
            interrupt_num++;

    SetConsoleTextAttribute(GetStdHandle(STD_OUTPUT_HANDLE),FOREGROUND_INTENSITY | FOREGROUND_BLUE);
        }
        cout<<"进入页："<<ref[i]<<endl;
        p=L.front->next;
        for(j=0;p!=NULL && j<phy_size;j++)
        {
            cout<<p->data<<"\t";
            p=p->next;
        }
        cout<<endl<<endl;
    }
    cout<<"先进先出置换算法缺页中断次数："<<interrupt_num<<endl;

    interrupt[2]=((float)interrupt_num/20.0)*100.0;
    free(p);
    DestroyQueue(L);
}
////////////////////////////////////////////////////////////////////////////////////////
/////
void LRU()                              /////////最近最久未使用算法
{
    cout<<"\n********************** 最 近 最 久 未 使 用 置 换 算 法 **********************"<<endl;
    int QNode_num=0;
        LinkQueue L;
```

```
QueuePtr p;
int i,j,e;
int interrupt_num=0;
InitQueue(L);
for(i=0;i<phy_size;i++)
{
    EnQueue(L,ref[i]);
}
p=(QueuePtr)malloc(sizeof(QNode));
p=L.front->next;
for(j=0;p!=NULL && j<phy_size;j++)
{
    cout<<p->data<<"\t";
    p=p->next;
}
cout<<endl;
for(i=phy_size;i<ref_size;i++)
{

    if(!SearchQueue(L,ref[i],QNode_num))    //产生缺页中断，选择最“老”的页面被替换
    {
        DeQueue(L,e);
        //cout<<e<<endl;
        EnQueue(L,ref[i]);
        interrupt_num++;

    }
    else if(QNode_num==1)//如果接下来是内存中的第一个，则将它移到最后一个，即标志为最近使用的页
    {
        EnQueue(L,ref[i]);
        DeQueue(L,e);
```

```
            }
            else if(QNode_num==2)//如果接下来是内存中的第二个，则将它删除，并在队列尾部添加相同元素，即标志为最近使用的页
            {
                DelMid_Queue(L,e);
                EnQueue(L,e);
            }
            cout<<"进入页："<<ref[i]<<endl;
            p=L.front->next;
            for(j=0;p!=NULL && j<phy_size;j++)//输出内存状况
            {
                cout<<p->data<<"\t";
                p=p->next;
            }
            cout<<endl<<endl;
        }
        cout<<"最近最久未使用置换算法缺页中断次数："<<interrupt_num<<endl;
        interrupt[3]=((float)interrupt_num/20.0)*100.0;

        DestroyQueue(L);
        free(p);
}
void CLOCK()
{
        cout<<"\n**********CLOCK 置 换 算 法 **********"<<endl;
        int interrupt_num=0;
        int i;
        int LNode_hit_num=0;        //标记带内存中与带进入页面相同的页面的位置
        int LNode_flag_num=0;       //标记访问位为 0 的页面在内存中的位置
        LinkList L;
        CreatList(L);
```

```
LinkList p;
p=(LinkList)malloc(sizeof(LNode));
for(i=0;i<phy_size;i++)
{
    Insert_LNode(L,ref[i]);
}
if(L->next==L) exit(-1);
p=L->next;
for(;p!=L;p=p->next)
{
    cout<<p->data<<"\t";
    //p->flag=1;
}
cout<<endl;
p=L->next;
while(p!=L)
{
    cout<<"A:"<<p->flag<<"\t";
    p=p->next;
}
cout<<endl<<endl;
for(i=phy_size;i<ref_size;i++)
{

SetConsoleTextAttribute(GetStdHandle(STD_OUTPUT_HANDLE),FOREGROUND_INTENSITY | FOREGROUND_INTENSITY);
    if(!Search_LinkList(L,ref[i],LNode_hit_num))
    {
        Search_LL_Flag(L,LNode_flag_num);//找到第一个 flag 标志为 0 的结点，其序号记录在 LNode_flag_num 中
        LNode_flag_num--;
        Exchange_LNode(L,ref[i],LNode_flag_num);// 将链表 L 中序号为 LNode_flag_num 的结点替换为内容为 ref[i]的结点
```

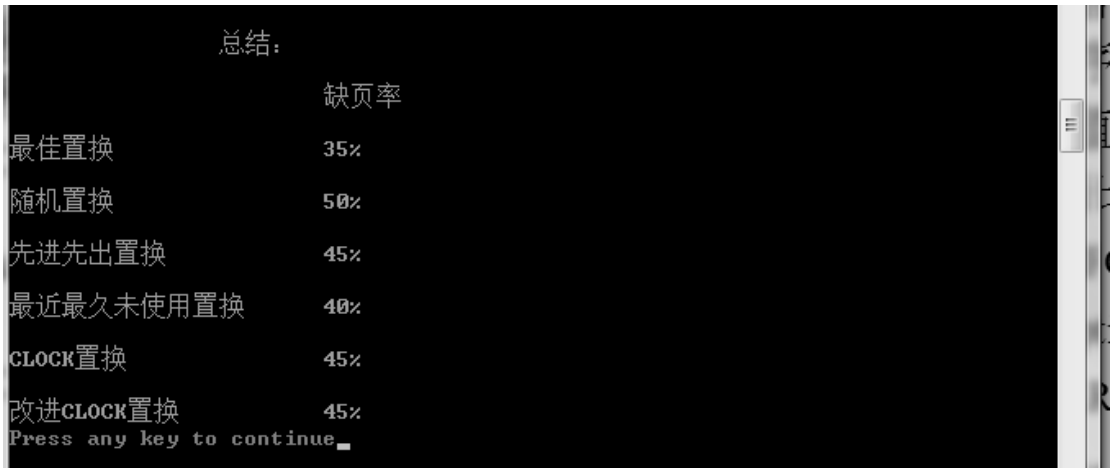
```
                interrupt_num++;

    SetConsoleTextAttribute(GetStdHandle(STD_OUTPUT_HANDLE),FOREGROU
ND_INTENSITY | FOREGROUND_BLUE);
            }
            else
                Set_LL_Flag(L,LNode_hit_num);
            cout<<"进入页："<<ref[i]<<endl;
            p=L->next;
            for(;p!=L;p=p->next)
            {
                cout<<p->data<<"\t";
                //p->flag=1;
            }
            cout<<endl;
            p=L->next;
            while(p!=L)
            {
                cout<<"A:"<<p->flag<<"\t";
                p=p->next;
            }
            cout<<endl<<endl;
        }
        cout<<"CLOCK 置换算法缺页中断次数："<<interrupt_num<<endl;        //
        interrupt[4]=((float)interrupt_num/20.0)*100.0;
        DestroyLinkList(L);
        //free(L);
}
////////////////////////////////////////////////////////////////////////////////
/////
void Modified_Clock()
{
    cout<<"\n******************  改  进  的  CLOCK  置  换  算  法
```

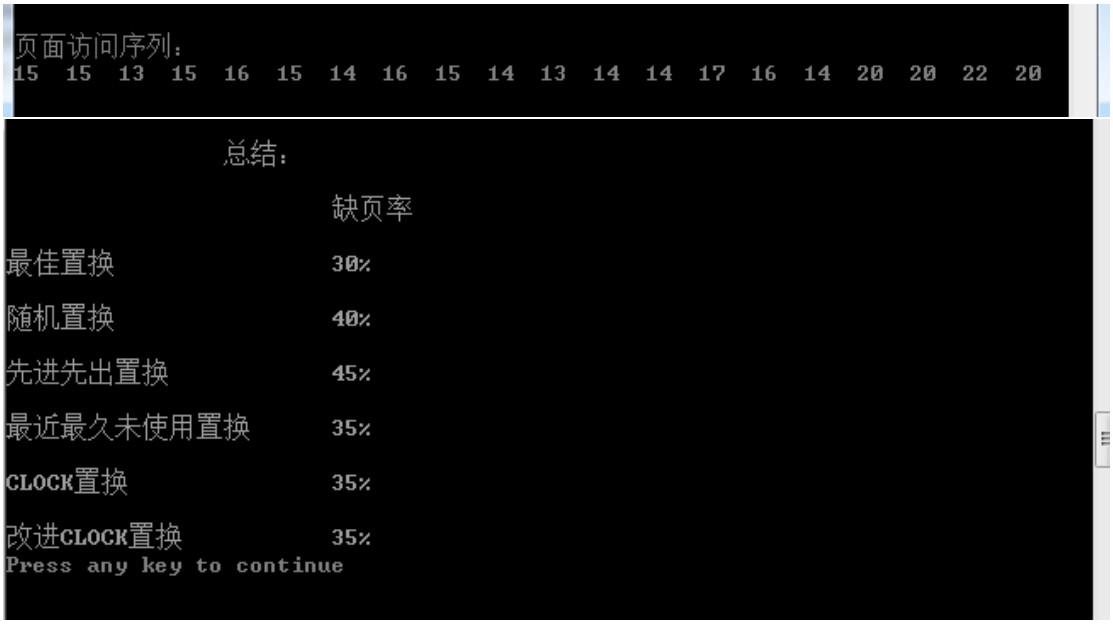
```
**********"<<endl;
    int interrupt_num=0;
    int i,temp;
    int LNode_hit_num=0;
    int LNode_flag_num=0;
    int LNode_modify_num=0;
    LinkList L;
    CreatList(L);
    LinkList p;
    p=(LinkList)malloc(sizeof(LNode));
    for(i=0;i<phy_size;i++)
    {
        Insert_LNode(L,ref[i]);
    }
    if(L->next==L) exit(-1);
    p=L->next;
    for(;p!=L;p=p->next)
    {
        cout<<p->data<<"\t\t";
        //p->flag=1;
    }
    cout<<endl;
    Sleep(1000);
    srand(time(NULL));        //设置时间种子
    temp=rand()%3;
    cout<<"修改页（内存中序号）："<<temp<<endl;
    Set_LL_modify(L,temp);
    p=L->next;
    while(p!=L)
    {
        cout<<"A:"<<p->flag<<"\tM:"<<p->modify<<"\t";
        p=p->next;
    }
```

```
cout<<endl<<endl;
for(i=phy_size;i<ref_size;i++)
{
    if(!Search_LinkList(L,ref[i],LNode_hit_num))
    {
        Search_LL_ModifyClock(L,LNode_modify_num);
        //Search_LL_Flag(L,LNode_flag_num);
        LNode_modify_num--;
        Exchange_LNode(L,ref[i],LNode_modify_num);
        interrupt_num++;
    }
    else
        Set_LL_Flag(L,LNode_hit_num);
    cout<<"进入页："<<ref[i]<<endl;
    p=L->next;
    for(;p!=L;p=p->next)
    {
        cout<<p->data<<"\t\t";
        //p->flag=1;
    }
    cout<<endl;
    Sleep(1000);
    srand(time(NULL));        //设置时间种子
    temp=rand()%3;
    cout<<"修改页（内存中序号）："<<temp<<endl;
    Set_LL_modify(L,temp);

    p=L->next;
    while(p!=L)
    {
        cout<<"A:"<<p->flag<<"\tM:"<<p->modify<<"\t";
        p=p->next;
    }
```

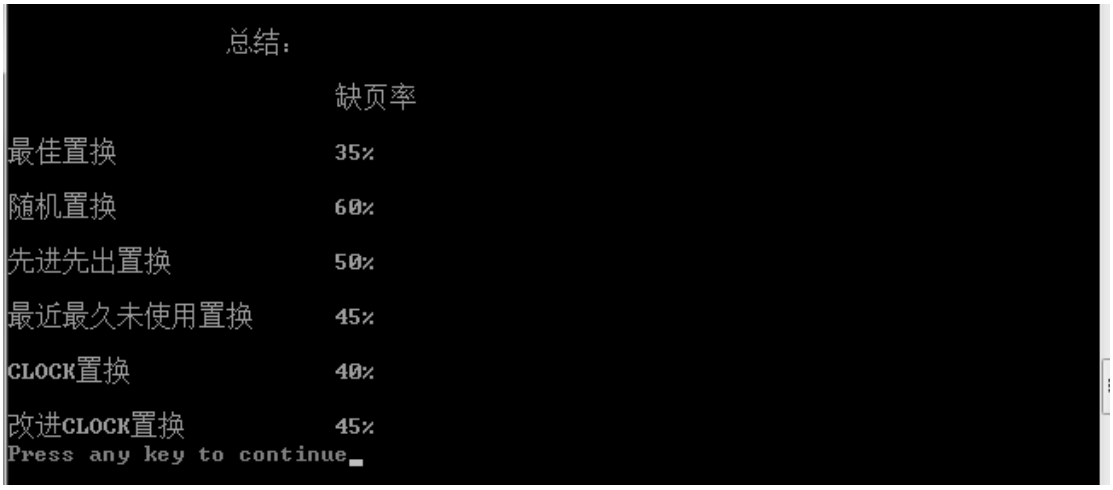
```
            cout<<endl<<endl;
        }
        cout<<"改进的 CLOCK 置换算法缺页中断次数："<<interrupt_num<<endl;
            interrupt[5]=((float)interrupt_num/20.0)*100.0;
        DestroyLinkList(L);
        //free(L);
}
int main()
{
        set_rand_num();
        ORA();
        RAND();
        FIFO();
        LRU();
        CLOCK();
        Modified_Clock();//*/
        cout<<"\n\n\t\t 总结："<<endl;
        cout<<"\n\t\t\t 缺页率"<<endl;
        cout<<"\n 最佳置换\t\t"<<interrupt[0]<<"%"<<endl;
        cout<<"\n 先进先出置换\t\t"<<interrupt[2]<<"%"<<endl;
        cout<<"\n 最近最久未使用置换\t"<<interrupt[3]<<"%"<<endl;
        cout<<"\nCLOCK 置换\t\t"<<interrupt[4]<<"%"<<endl;
        cout<<"\n 改进 CLOCK 置换\t\t"<<interrupt[5]<<"%"<<endl;
        SetConsoleTextAttribute(GetStdHandle(STD_OUTPUT_HANDLE),FOREGROU
ND_INTENSITY | FOREGROUND_INTENSITY);
        return 1;
}
```

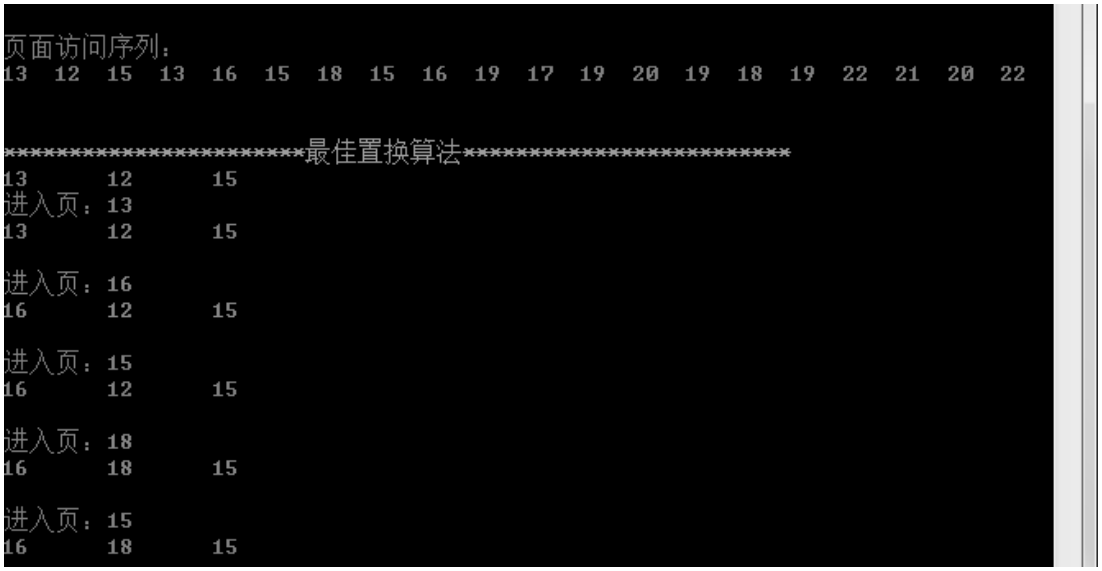
2.实验结果截图

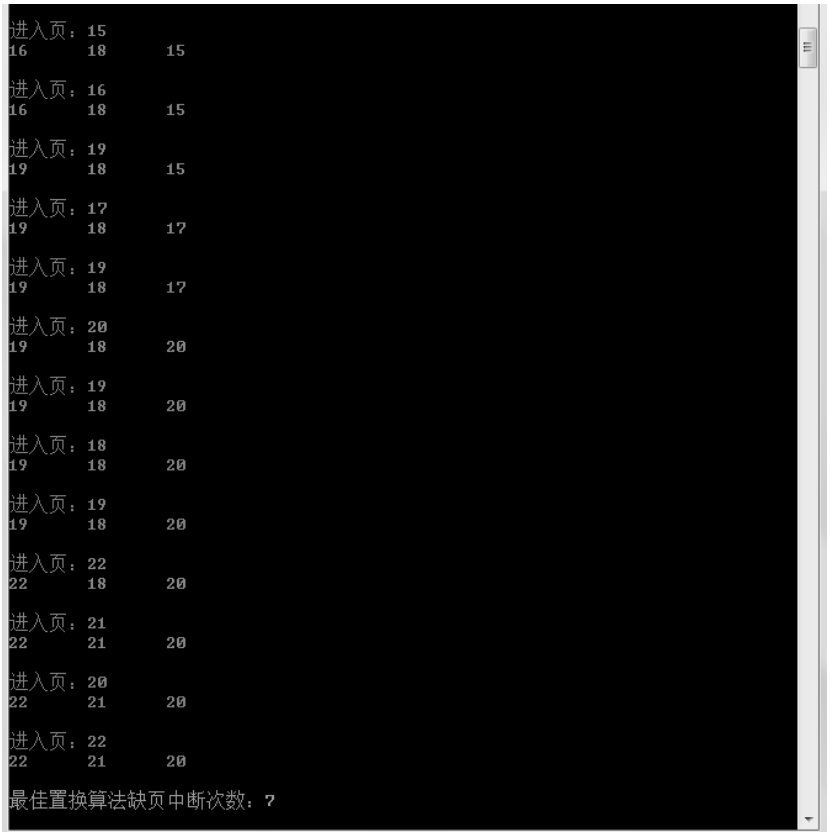

总结：
缺页率
最佳置换 35%
随机置换 50%
先进先出置换 45%
最近最久未使用置换 40%
CLOCK置换 45%
改进CLOCK置换 45%
Press any key to continue

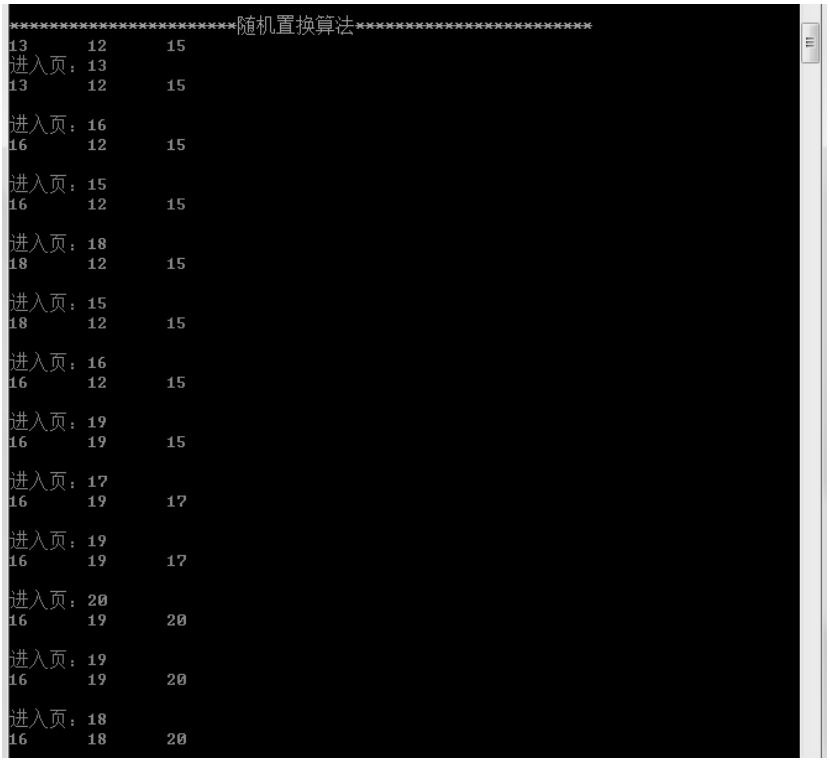

页面访问序列：
15 15 13 15 16 15 14 16 15 14 13 14 14 17 16 14 20 20 22 20
总结：
缺页率
最佳置换 30%
随机置换 40%
先进先出置换 45%
最近最久未使用置换 35%
CLOCK置换 35%
改进CLOCK置换 35%
Press any key to continue


页面访问序列：
13 12 15 13 16 15 18 15 16 19 17 19 20 19 18 19 22 21 20 22


总结：
缺页率
最佳置换 35%
随机置换 60%
先进先出置换 50%
最近最久未使用置换 45%
CLOCK置换 40%
改进CLOCK置换 45%
Press any key to continue


页面访问序列：
13 12 15 13 16 15 18 15 16 19 17 19 20 19 18 19 22 21 20 22
**********最佳置换算法**********
13 12 15
进入页：13
13 12 15
进入页：16
16 12 15
进入页：15
16 12 15
进入页：18
16 18 15
进入页：15
16 18 15


进入页：15
16 18 15
进入页：16
16 18 15
进入页：19
19 18 15
进入页：17
19 18 17
进入页：19
19 18 17
进入页：20
19 18 20
进入页：19
19 18 20
进入页：18
19 18 20
进入页：19
19 18 20
进入页：22
22 18 20
进入页：21
22 21 20
进入页：20
22 21 20
进入页：22
22 21 20
最佳置换算法缺页中断次数：7

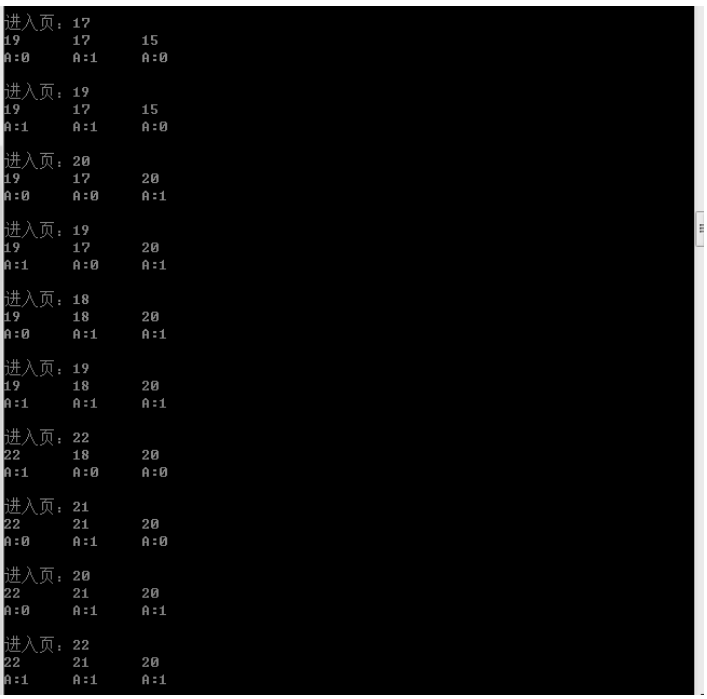

**********随机置换算法**********
13 12 15
进入页：13
13 12 15
进入页：16
16 12 15
进入页：15
16 12 15
进入页：18
18 12 15
进入页：15
18 12 15
进入页：16
16 12 15
进入页：19
16 19 15
进入页：17
16 19 17
进入页：19
16 19 17
进入页：20
16 19 20
进入页：19
16 19 20
进入页：18
16 18 20

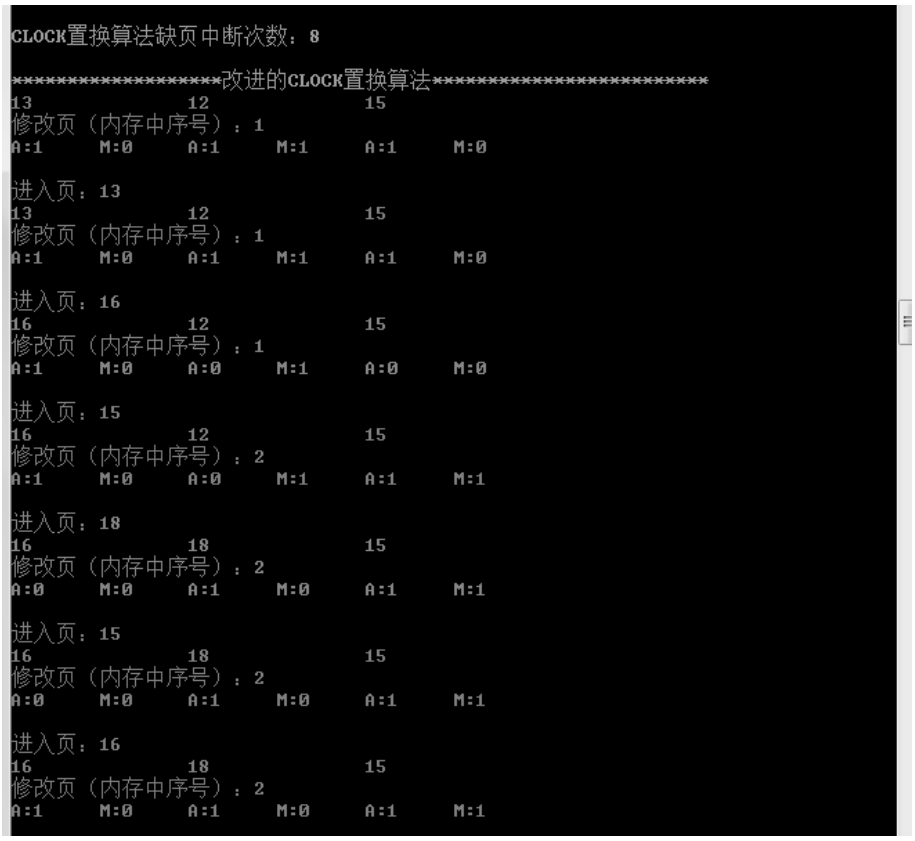
```
进入页：19
19      18      20

进入页：22
19      18      22

进入页：21
19      21      22

进入页：20
19      21      20

进入页：22
19      21      22

随机置换算法缺页中断次数：12

**********先进先出置换算法**********
13      12      15
进入页：13
13      12      15

进入页：16
12      15      16

进入页：15
12      15      16

进入页：18
15      16      18

进入页：15
15      16      18

进入页：16
15      16      18

进入页：19
16      18      19
```

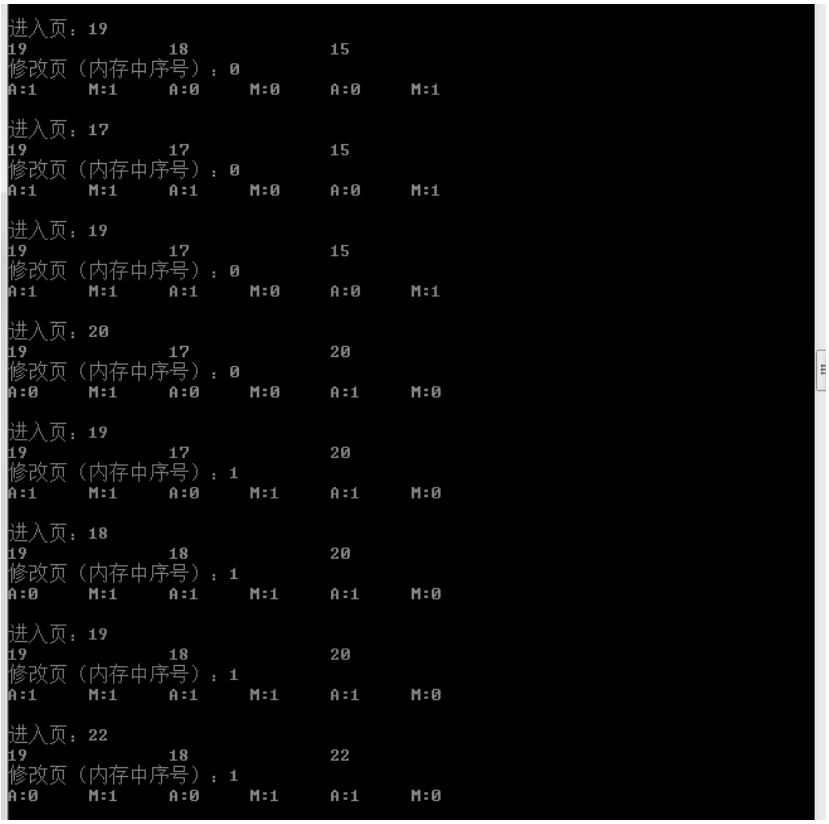
```
进入页：17
18      19      17

进入页：19
18      19      17

进入页：20
19      17      20

进入页：19
19      17      20

进入页：18
17      20      18

进入页：19
20      18      19

进入页：22
18      19      22

进入页：21
19      22      21

进入页：20
22      21      20

进入页：22
22      21      20

先进先出置换算法缺页中断次数：10

**********最近最久未使用置换算法**********
13      12      15
进入页：13
12      15      13

进入页：16
15      13      16

进入页：15
13      16      15
```

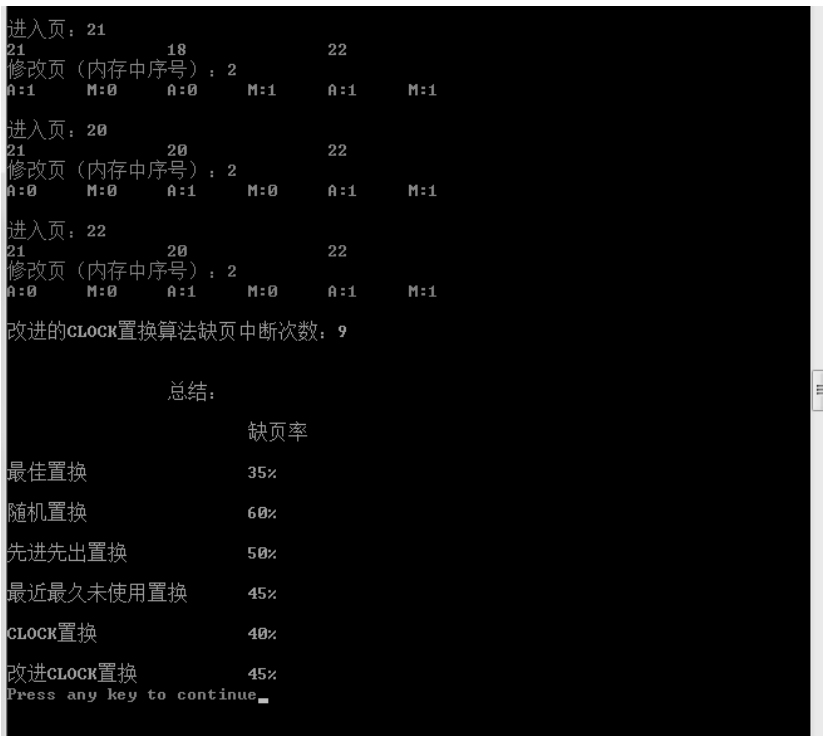
```
进入页：15
13      16      15

进入页：18
16      15      18

进入页：15
16      18      15

进入页：16
18      15      16

进入页：19
15      16      19

进入页：17
16      19      17

进入页：19
16      17      19

进入页：20
17      19      20

进入页：19
17      20      19

进入页：18
20      19      18

进入页：19
20      18      19

进入页：22
18      19      22

进入页：21
19      22      21
```

```
进入页：20
22      21      20

进入页：22
21      20      22

最近最久未使用置换算法缺页中断次数：9

**********CLOCK置换算法**********
13      12      15
A:1     A:1     A:1

进入页：13
13      12      15
A:1     A:1     A:1

进入页：16
16      12      15
A:1     A:0     A:0

进入页：15
16      12      15
A:1     A:0     A:1

进入页：18
16      18      15
A:0     A:1     A:1

进入页：15
16      18      15
A:0     A:1     A:1

进入页：16
16      18      15
A:1     A:1     A:1

进入页：19
19      18      15
A:1     A:0     A:0
```


进入页：17
19 17 15
A:0 A:1 A:0
进入页：19
19 17 15
A:1 A:1 A:0
进入页：20
19 17 20
A:0 A:0 A:1
进入页：19
19 17 20
A:1 A:0 A:1
进入页：18
19 18 20
A:0 A:1 A:1
进入页：19
19 18 20
A:1 A:1 A:1
进入页：22
22 18 20
A:1 A:0 A:0
进入页：21
22 21 20
A:0 A:1 A:0
进入页：20
22 21 20
A:0 A:1 A:1
进入页：22
22 21 20
A:1 A:1 A:1


CLOCK置换算法缺页中断次数：8
**********改进的CLOCK置换算法**********
13 12 15
修改页（内存中序号）：1
A:1 M:0 A:1 M:1 A:1 M:0
进入页：13
13 12 15
修改页（内存中序号）：1
A:1 M:0 A:1 M:1 A:1 M:0
进入页：16
16 12 15
修改页（内存中序号）：1
A:1 M:0 A:0 M:1 A:0 M:0
进入页：15
16 12 15
修改页（内存中序号）：2
A:1 M:0 A:0 M:1 A:1 M:1
进入页：18
16 18 15
修改页（内存中序号）：2
A:0 M:0 A:1 M:0 A:1 M:1
进入页：15
16 18 15
修改页（内存中序号）：2
A:0 M:0 A:1 M:0 A:1 M:1
进入页：16
16 18 15
修改页（内存中序号）：2
A:1 M:0 A:1 M:0 A:1 M:1





3. 实验结论

1、最佳置换算法效果最佳

性能最高。但通过课堂上的学习，我们知道这只是一种理想化算法，但实际上却难于实现，故主要用于算法评价参照。

2、随机算法的性能总是最不好的

这是由于随机算法每次总是从所有页面中随机挑一个置换出去，但页面的访问存在着局部性的原理，并不是随机的，因此它的性能较差。

3、最近最久未使用算法的性能较好

相较于先进先出和两种 clock 算法，最近最久未使用算法的性能略好，我们测试的数据规模相对较小，相信如果采用更大规模的数据，其优势会更加明显。

要想在实际的应用中实现本算法，用软件的方法速度太慢，影响程序执行效率，如果采用硬件方法实现，则需要增加大量的硬件设备。

4、先进先出与 clock 算法的性能基本相同

这是由于两种 clock 算法遍历链表采用的就是 FIFO 的方法，而改进的 clock 算法相比于简单 clock 算法的优势主要体现在会根据是否被修改进行选择，以减少写回所花费的时间。