页面替换算法

学生姓名：王昊飞

学号：16281269

1. 算法原理及过程：
   1. 最佳置换算法（OPT）：
      1. 原理：所选择的被淘汰的页面将是以后永不使用的，或是在将来最长时间内不再被访问的页面。在这样理想的情况下，OPT算法能够达到最低的缺页率，但是其条件是理想的无法实现的，一般用该算法评价其他算法。
      2. 过程：当发生页面置换时，检查页面是否包含在内存中，是的话直接显示，否则检查内存是否满，在满的情况下淘汰引用串和。
   2. 随机置换算法（PRA）：
      1. 原理：所选择的被淘汰的页面将是内存里随机的一个页面。
      2. 过程：当发生页面置换时，查询页面是否在内存中，在的话直接显示页面，否则检验内存是否满，在满的情况下随机淘汰一个内存中页面，不满的情况下直接插入内存空位。
   3. 先进先出置换算法（FIFO）：
      1. 原理：所选择的被淘汰页面是最先进入内存的页面。
      2. 过程：当发生页面置换时，查询页面是否在内存中，在的话直接显示页面，否则查询内存是否满，在满的情况下淘汰一个内存中最早进入的页面，不满的情况下直接插入内存空位。
   4. 最近最久未使用置换算法（LRU）：
      1. 原理：所选择的被淘汰的页面是最近最久未使用的页面。
      2. 过程：当发生页面置换时，查询页面是否在内存中，在的话直接显示页面，否则查询内存是否满，在满的情况下淘汰一个内存中最近最久未使用的页面，不满的情况下直接插入内存空位。
   5. 简单时钟置换算法（First-Clock）：
      1. 原理：所选择的被淘汰的页面是访问位为0的页面。
      2. 过程：当发生页面置换时，查询页面是否在内存中，在的话直接显示页面，否则查询内存是否满，在满的情况下，依次访问内存中的页面访问位，为1则置为0访问下一位；为0则淘汰页面，将新插入的页面访问位置为1，指针指向下一位。在不满的情况下直接插入内存空位，且页面访问位置为1，指针指向内存下一位.
   6. 改进型时钟置换算法（Second-Clock）：
      1. 原理：所选择的最佳被淘汰页面是访问位为0，修改位也为0的页面。
      2. 过程：当发生页面置换时，首先判断是否为修改程序：如果是则修改插入页面的修改位为1否则为0，接着查询页面是否在内存中，在的话直接显示页面，否则查询内存是否满，在满的情况下，
         * 1. 依次访问内存页面的修改位和访问位，如果修改位与访问位同时为0则淘汰该页面，同时将新插入的页面访问位置为1，指针指向下一位;
           2. 遍历后不存在同时为0的情况，那么重新依次查找访问位为0修改位为1的页面，如果指针位置的访问位为1，则将其置为0，为0则淘汰该页面，并且将新插入的页面访问位置为1，指针指向下一位.
           3. 这时遍历如果还未找到,则再次重新遍历，按1)与2)的过程就能找到合适页面，淘汰该页面并且将新插入的页面访问位置为1。
   7. 用于测试的引用串
      1. 原理：引用串的关键在于如何模拟程序的局部性。
      2. 过程：
         1. 确定虚拟内存的尺寸P，工作面的起始位置p，工作面中包含的页数e，工作面移动率m，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，并记录到引用串中
         3. 生成一个随机数r，0 ≤ r ≤ 1， 如果r < t，则为p生成一个新值，否则p=（p+1）modP
         4. 如果想继续加大引用串的长度，请返回第二步，否则结束
2. 、算法流程图

开始

引用串生成

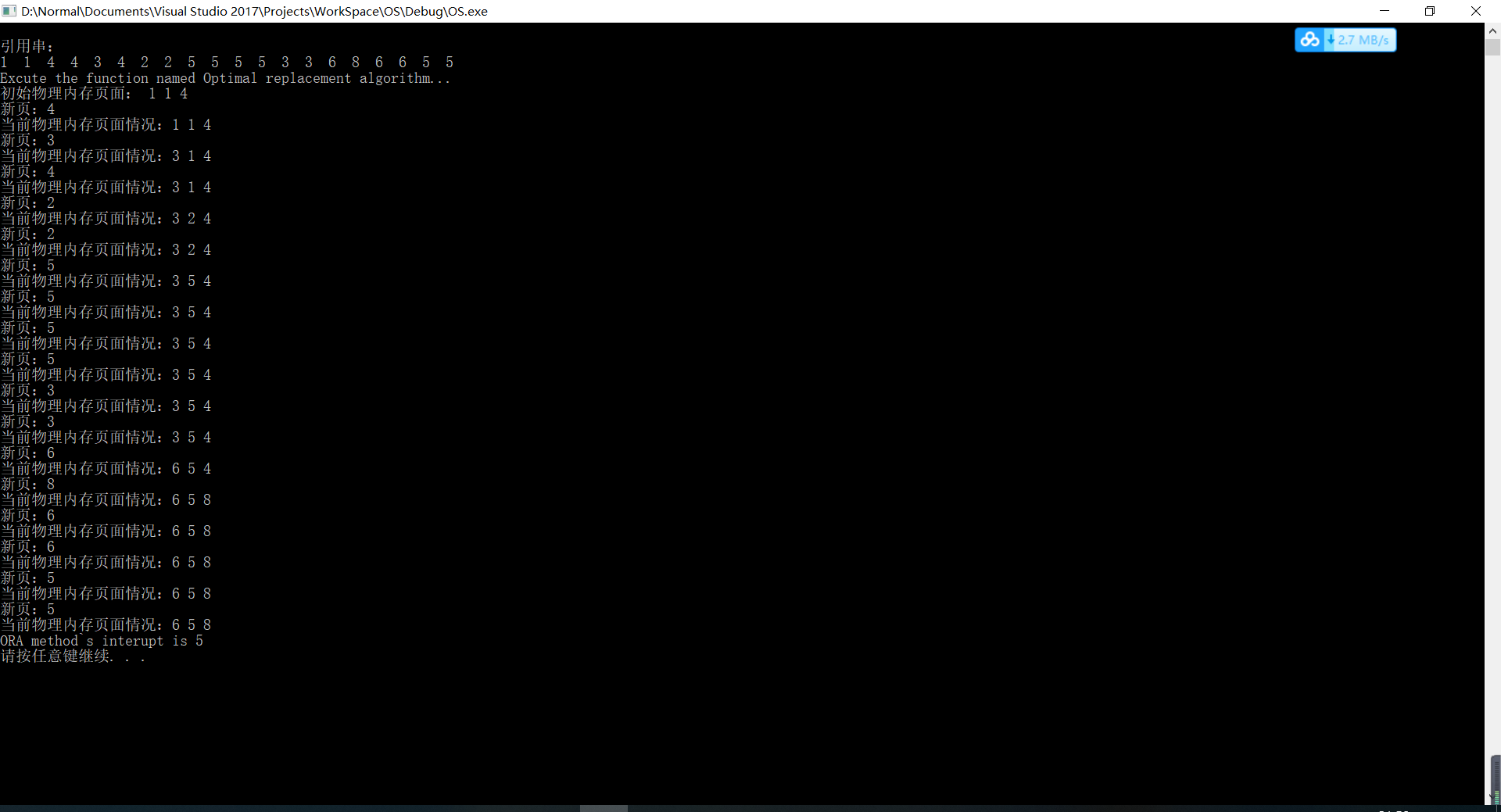
调用置换算法

初始化相关数据结构

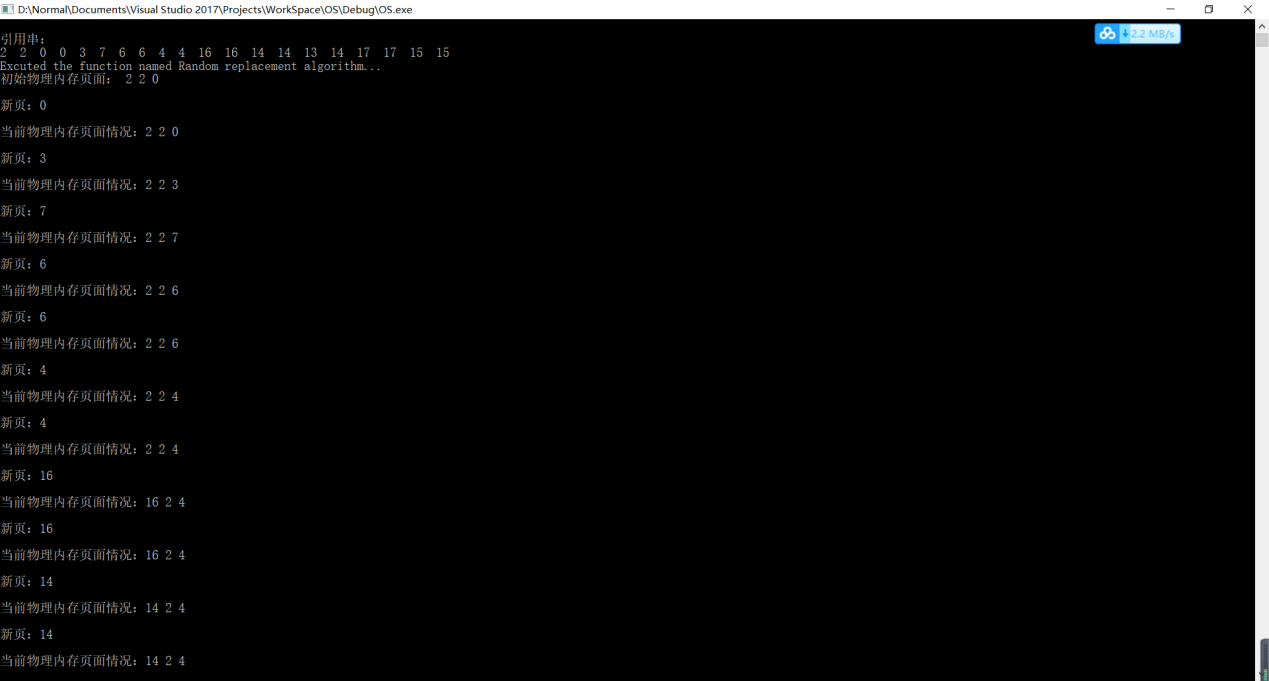
执行算法逻辑

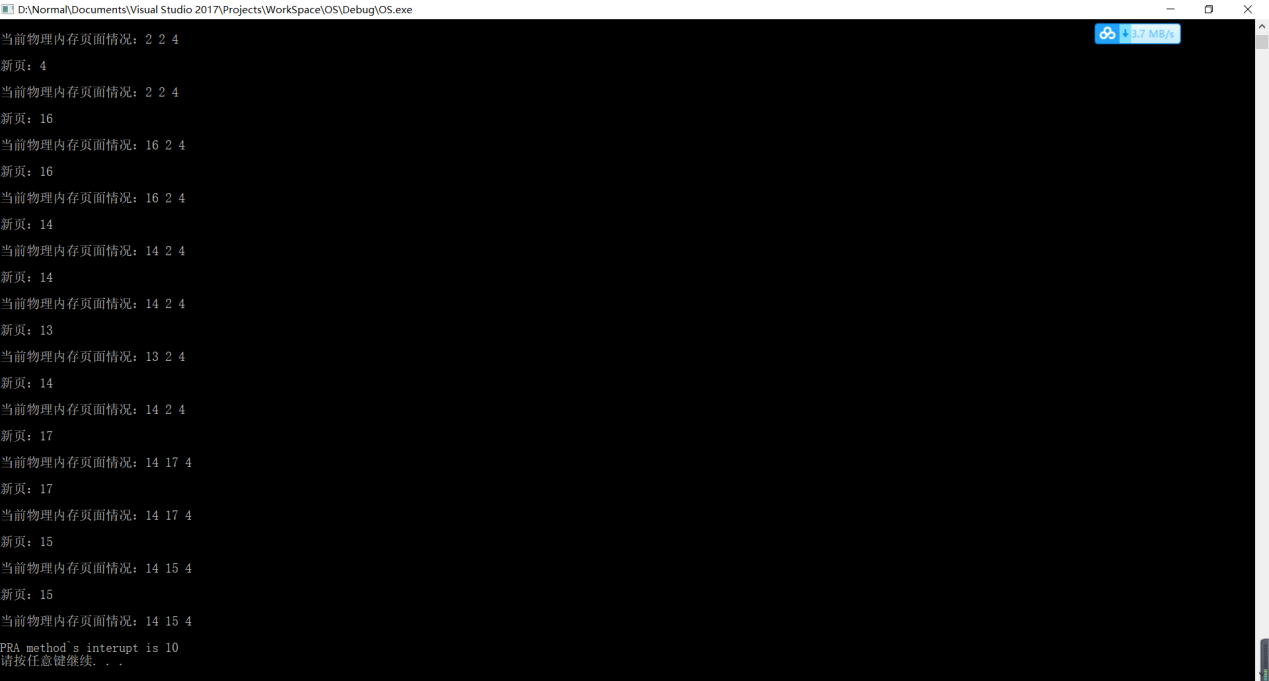
输出中断次数

1. 结果展示

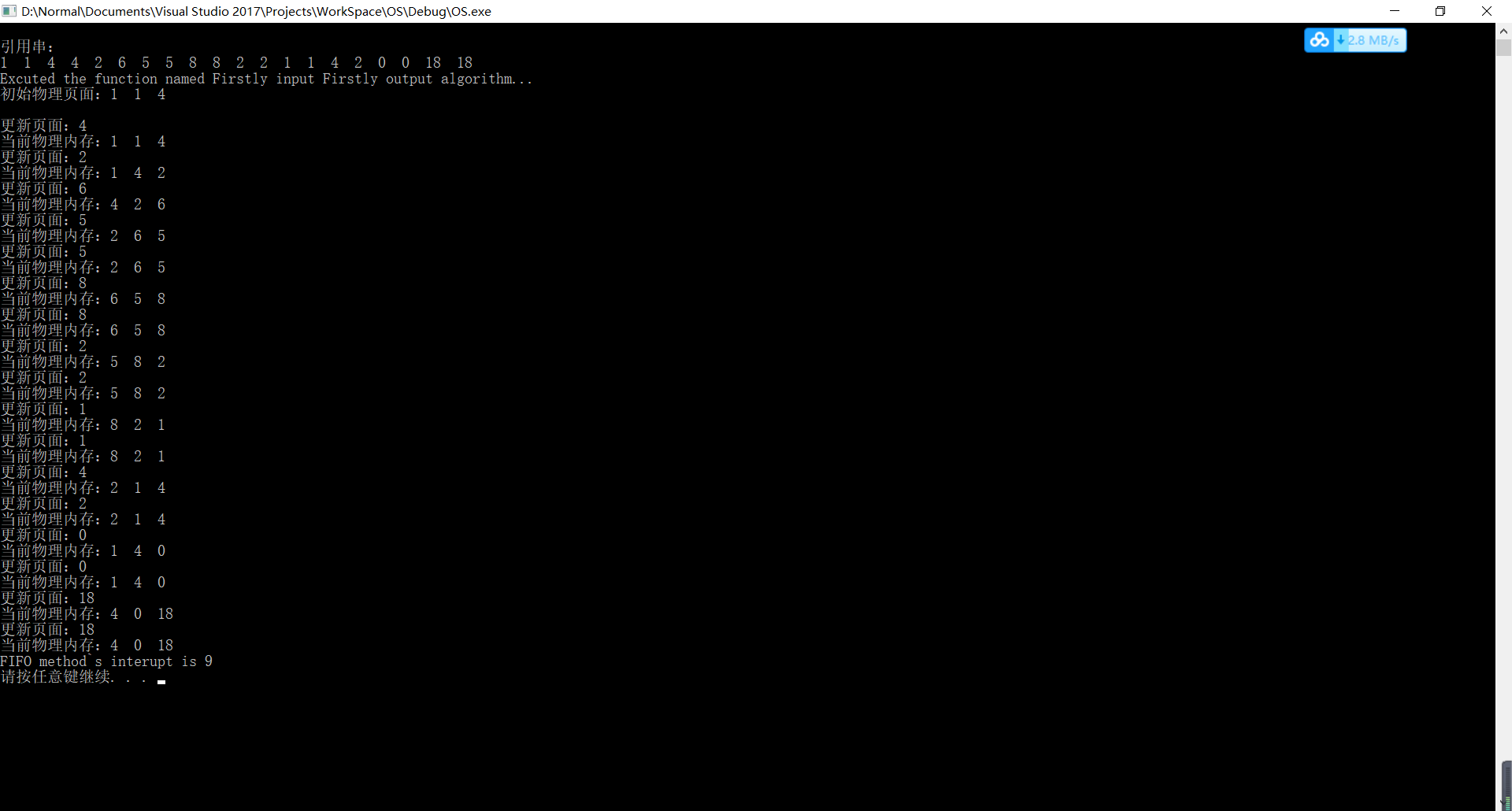


图（最佳置换）

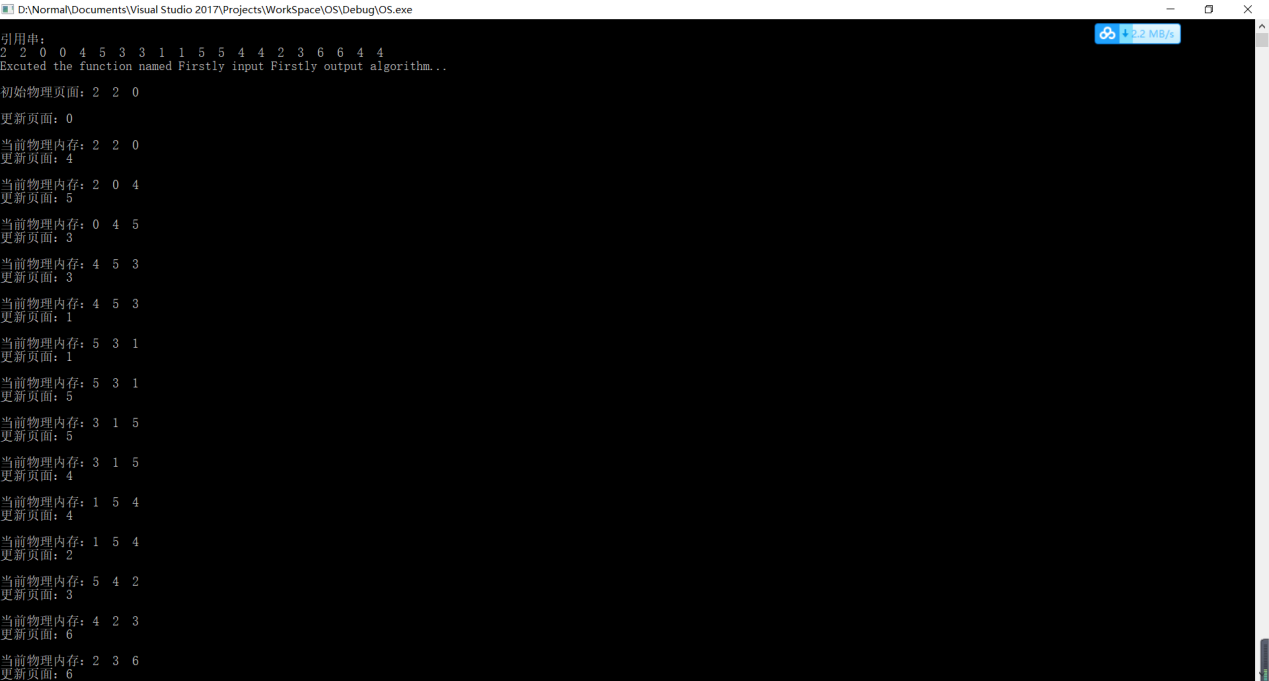


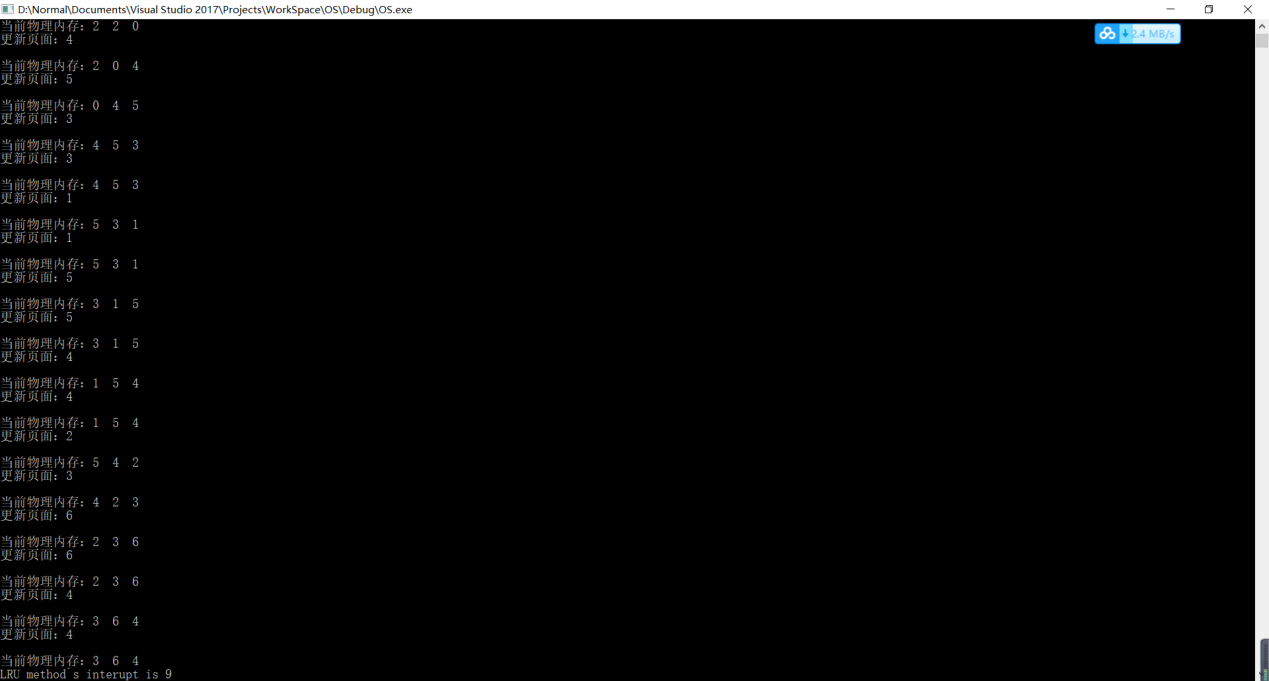


图（随机置换）

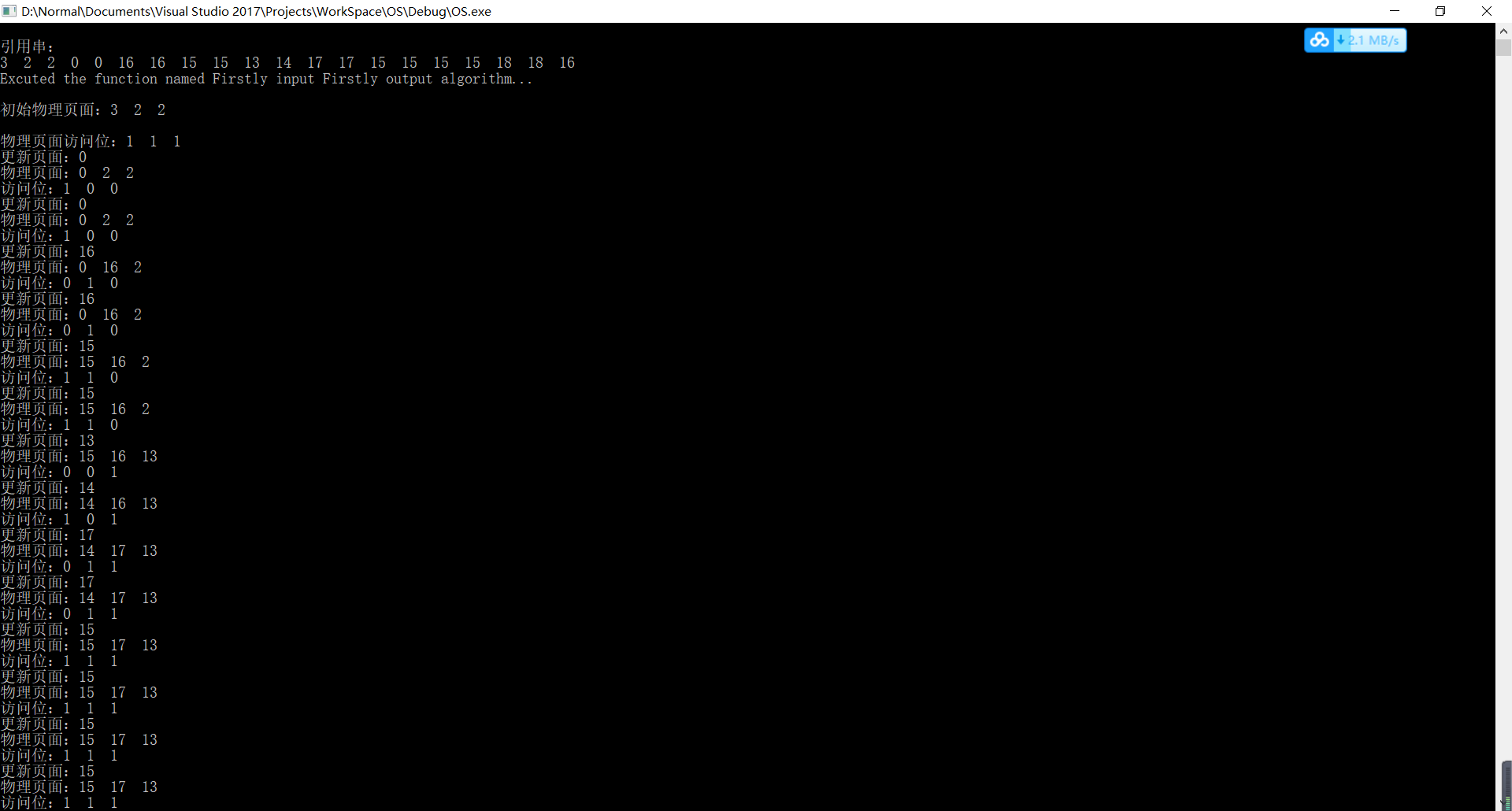


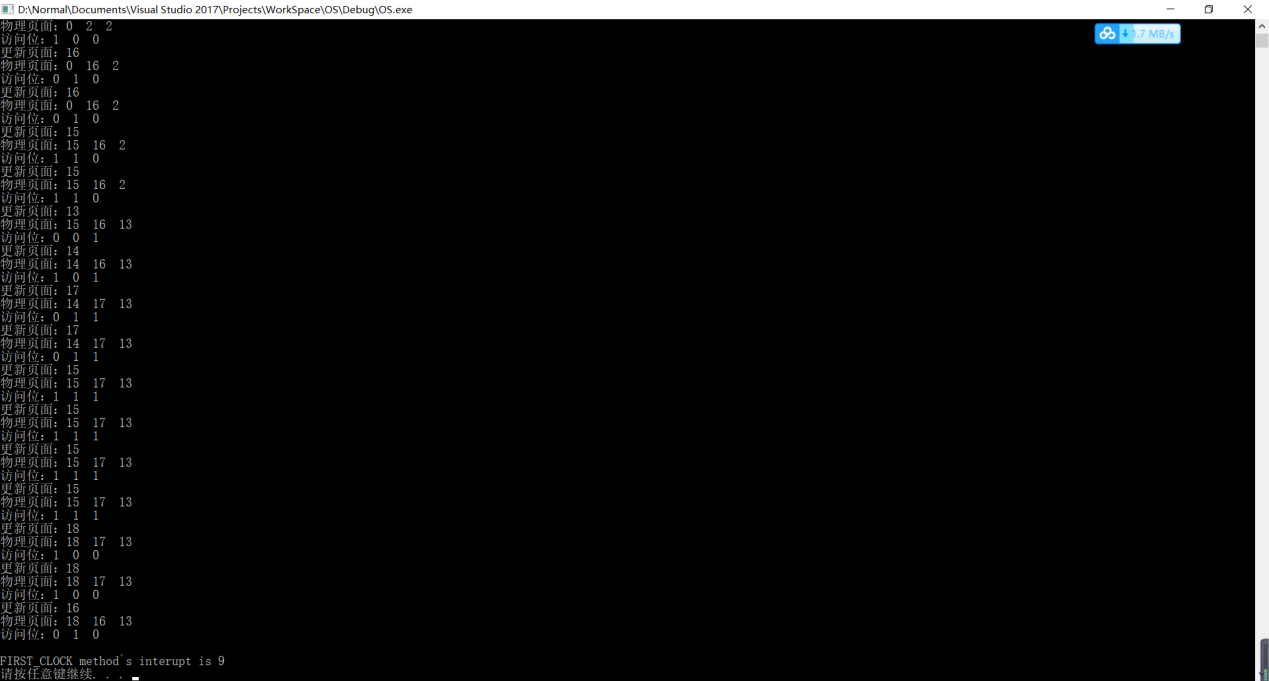
图（先进先出置换）



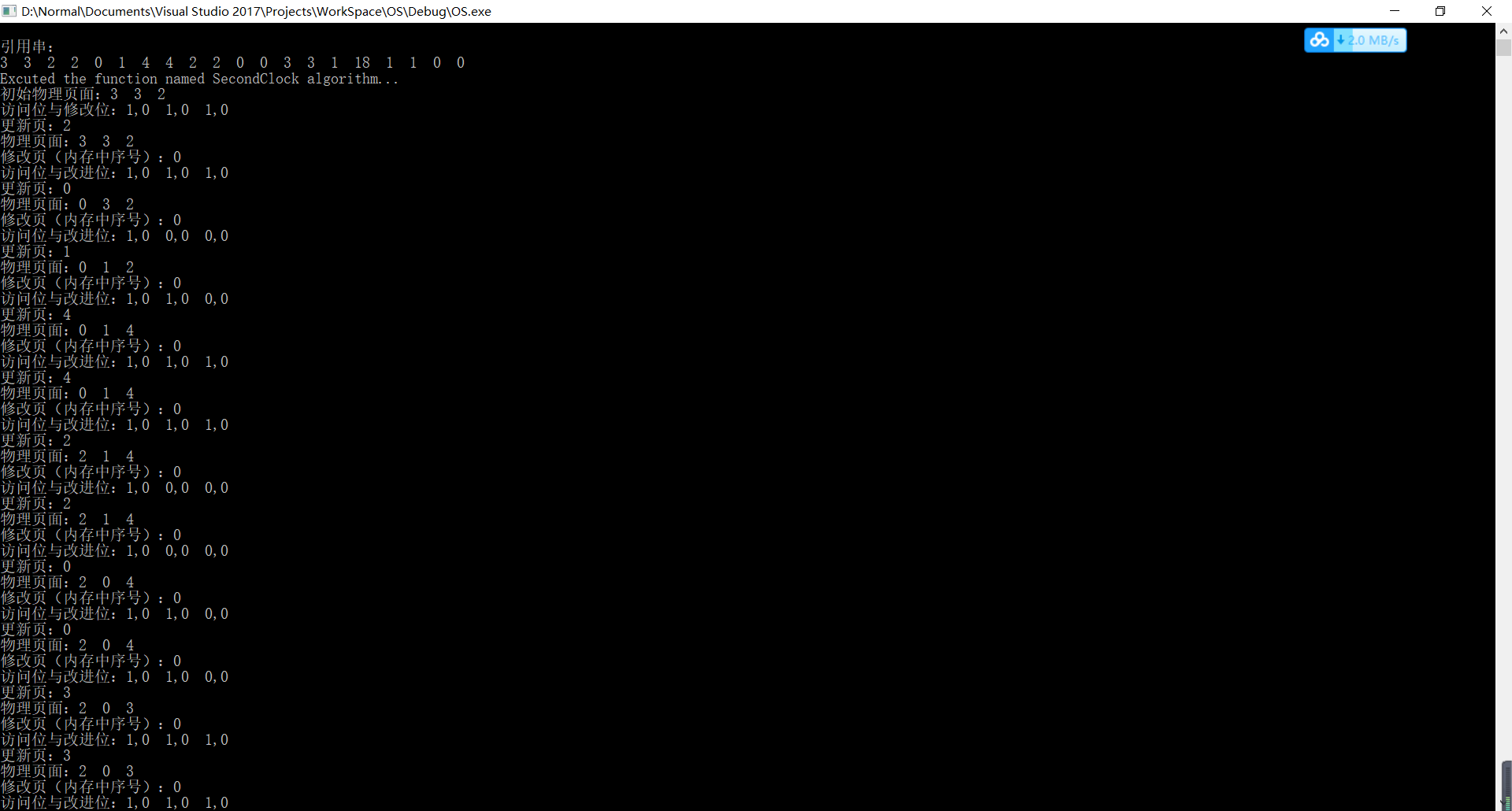


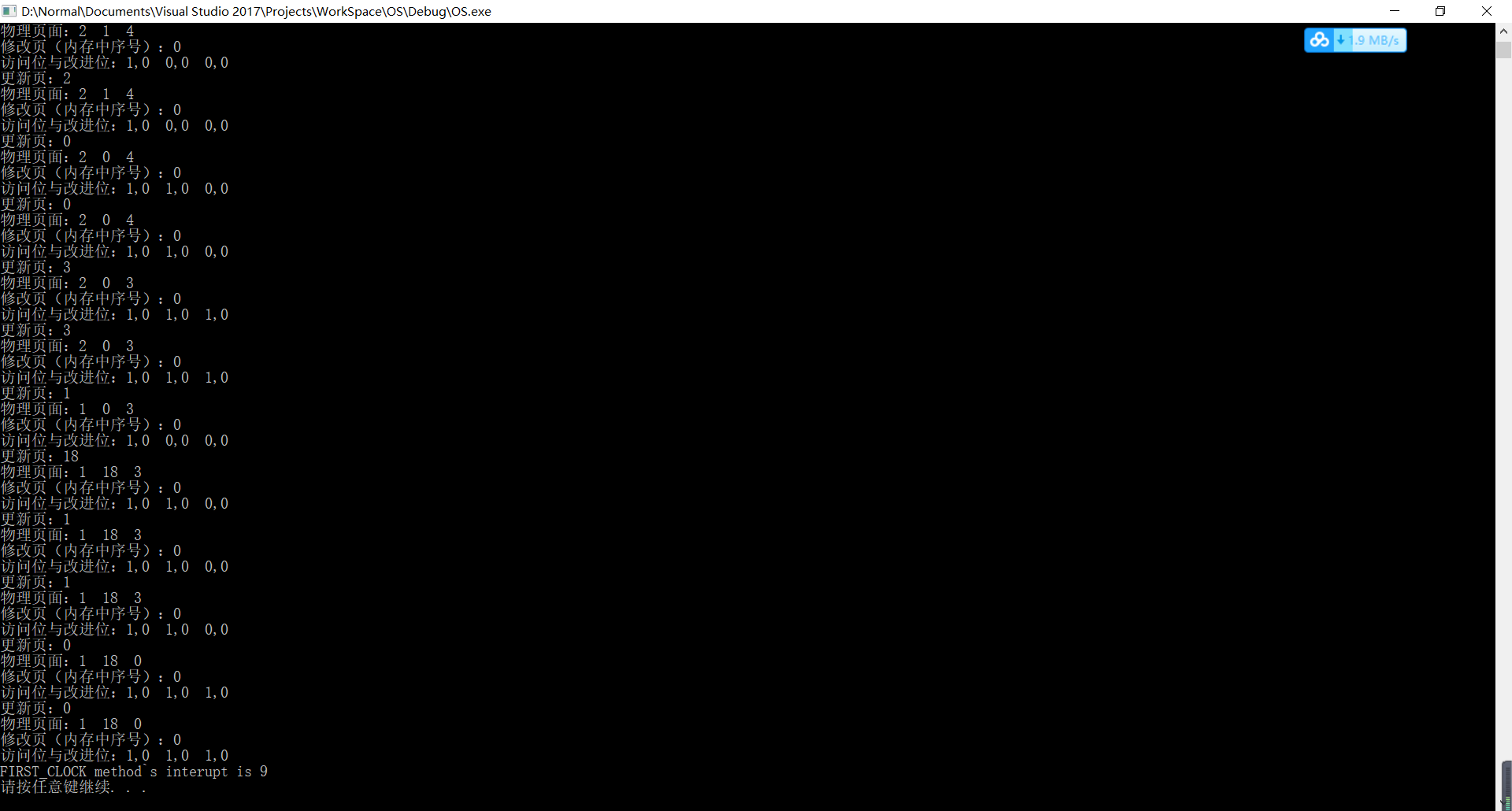
图（最近最久未使用置换）





图（时钟置换）





图（改进的时钟置换）

1. 结果分析
   1. FIFO算法是否比随机替换算法优越？LRU算法比FIFO 算法优越 多少？

FIFO算法在总体上比随机替换算法优先，但是在某些特定的引用串的情况下，随即替换算法比FIFO算法更加优越，LRU算法的置换次数是FIFO置换算法低一些，所以效率提升不是很明显，但是在具体实现的过程中，却需要付出很大的代价，即为每页配置的寄存器，或者一个栈，这样计算下来，LRU置换算法在现实中的应用价值很低，其只存在理论上的在置换次数方面的优越性。

* 1. LRU算法和最优算法有何差距？

LRU算法与最优算法的差距体现在两个方面，一个是磁环效率方面的差距，最优置换算法的置换次数大概是LRU置换算法低较多，但是在某些引用串时，LRU的置换效率与OPT的差别不大；另一个方面是在其具体实现的时候，需要付出的硬件代价比较大。

* 1. Clock算法和LRU算法有何差距？

Clock置换算法与LRU置换算法的差距主要体现在置换效率方面，其置换次数比LRU略多，差距并不大；而在具体实现方面，Clock实现的硬件代价比LRU小，因为Clock只需要一位标志位，而LRU需要配置一个和页面数相等的n位寄存器。

1. 实验问题

主要解决页面替换算法的逻辑和相关数据结构的适配。

1. 心得体会

本次实验完成了对页面替换算法的实现，分别用了不同的数据结构诸如链表和队列实现相关的模拟，程序流程相互比较类似，实现思路比较清楚。

1. 程序清单

#include<iostream>

#include<windows.h>

#include<time.h>

#include<malloc.h>

#include<conio.h>

using namespace std;

#define P 20 // 虚拟内存大小

#define F 3 // 物理内存大小

int referenceString[P]; // 引用串

int M[F]; // 物理内存数组

float interrupt[6] = { 0.0 };

// 页面数据结构

typedef struct Page

{

int data; // 页面数据内容

bool access; // 访问位

bool modify; // 修改位

struct Page \*next; // 下一页面

} \*PageList; // 页面单元和链表

typedef struct

{

PageList front;

PageList rear;

}PageQueue; // 页面队列

// 生成具有局部性质的引用串

/\*

@param: int p 工作面的起始位置

@param: int e 工作面包含页数

@param: int m 工作面移动率

@param: double t 随机阈值

\*/

void getReferenceString(int p, int e, int m, double t)

{

int pos = 0;

int last = 0;

int i = 0;

cout << "\n引用串：" << endl;

while (1)

{

if (last + m > P)

{

break;

}

// 生成m个取值范围在p和p+e之间的随机数，记录到引用串中

for (i = last; i <last + m; i++, pos++)

{

Sleep(500);

srand((unsigned)time(NULL));

referenceString[i] = (rand() % e + p) % P;

cout << referenceString[pos] << " ";

}

// 生成一个随机数在（0，1）的r

srand((unsigned)time(NULL));

double r = (double)(rand() % 100) / 100.0;

// 比较r与t的值

if (r < t) // r < t 创造一个新值

{

p = rand() % P;

}

else // r 大于等于 t p = (p+1) mod P

{

p = (p + 1) % P;

}

last = i;

}

}

/\*

-----------------------------------------链表操作-------------------------------------------

\*/

// 创建链表

bool creatList(PageList &L)

{

L = (PageList)malloc(sizeof(Page));

if (!L)

{

cout << "Error with establish the List!" << endl;

return false;

}

else

{

L->next = L;

L->access = 0;

L->modify = 0;

return true;

}

}

// 插入节点

bool insertList(PageList &L, int data)

{

// 初始化节点链表

PageList p = (PageList)malloc(sizeof(Page)); // 遍历链表

PageList q = (PageList)malloc(sizeof(Page)); // 插入节点

// 设置插入节点信息

q->data = data; // 插入数据内容

q->access = 1; // 设置访问位

q->modify = 0; // 设置修改位

// 设置遍历链表模板

p = L; // 指向目标链表

// 遍历到最后一个链表节点

while (p->next != L)

{

p = p->next;

}

// 插入q节点

p->next = q;

q->next = L;

return true;

}

// 设置节点属性

bool setAccessBit(PageList &L, int pos, bool access)// 设置节点修改位

{

// 检验数据的正确性

if (L->next == L)

{

cout << "Error with List" << endl;

return false;

}

// 初始化遍历

PageList p = (PageList)malloc(sizeof(Page));

p = L->next;

// 查找下标为pos的节点

int i = 1;

while (p != L && i != pos)

{

p = p->next;

}

// 检查遍历结果

if (p == L)

{

return false;

}

else

{

p->access = access;

}

return true;

}

bool setModifyBit(PageList &L, int pos, bool modify)// 设置节点修改位

{

// 检验数据的正确性

if (L->next == L)

{

cout << "Error with List" << endl;

return false;

}

// 初始化遍历

PageList p = (PageList)malloc(sizeof(Page));

p = L->next;

// 查找下标为pos的节点

int i = 1;

while (p != L && i != pos)

{

p = p->next;

}

// 检查遍历结果

if (p == L)

{

return false;

}

else

{

p->modify = modify;

}

return true;

}

// 查询节点

int searchListwithData(PageList &L, int data)

{

int pos = 1;

// 初始化遍历链表

PageList p = (PageList)malloc(sizeof(Page));

p = L->next;

// 遍历链表

while (p != L && data != p->data)

{

p = p->next;

pos++;

}

// 判断查询是否有结果

if (p == L)

{

return -1;

}

else

{

return pos;

}

}

// 销毁节点

bool destroyList(PageList &L)

{

// 初始化临时变量

PageList p = (PageList)malloc(sizeof(Page)); // 初始化临时遍历变量

PageList q = (PageList)malloc(sizeof(Page)); // 初始化临时节点变量

if (!p || !q)

{

cout << "Error with the temp List or Node" << endl;

return false;

}

p = L->next;

// 释放

while (p != L->next)

{

q = p->next;

free(p);

p = q;

}

free(q);

return true;

}

/\*

---------------------------------------------------------------------------------------------

\*/

/\*

-----------------------------------------队列操作-------------------------------------------

\*/

bool initQueue(PageQueue &Q)

{

// 初始化队列指针

Q.front = Q.rear = (PageList)malloc(sizeof(Page));

if (!Q.front || !Q.rear)

{

cout << "Error with QueueInitial" << endl;

return false;

}

Q.front->next = NULL;

return true;

}

bool EnQueue(PageQueue &Q, int data)

{

// 初始化插入元素

PageList p = (PageList)malloc(sizeof(Page));

if (!p)

{

cout << "Error with malloc in EnQueue" << endl;

return false;

}

// 设置插入队伍元素

p->data = data;

p->next = NULL;

// 插入目标队列

Q.rear->next = p;

Q.rear = p;

return true;

}

bool DeQueue(PageQueue &Q, int &data)

{

if (Q.front == Q.rear)

{

cout << "Error with Queue in DeQueue" << endl;

return false;

}

PageList p = (PageList)malloc(sizeof(Page));

if (!p)

{

cout << "Error with malloc in DeQueue" << endl;

return false;

}

p = Q.front->next;

data = p->data;

Q.front->next = p->next;

if (Q.rear == p)

Q.rear = Q.front;

free(p);

return true;

}

bool DeQueueOne(PageQueue &Q, int &data, int pos, PageQueue &PQ)

{

if (Q.front == Q.rear)

{

cout << "Error with Queue in DeQueue" << endl;

return false;

}

PageList p = (PageList)malloc(sizeof(Page));

if (!p)

{

cout << "Error with malloc in DeQueue" << endl;

return false;

}

p = Q.front->next;

int i = 1;

while (i != pos && p->next != NULL)

{

EnQueue(PQ, p->data);

p = p->next;

}

data = p->data;

if (p == NULL)

{

cout << "Error with DeQueueOne" << endl;

return false;

}

while (p->next != NULL)

{

EnQueue(PQ, p->next->data);

p = p->next;

}

return true;

}

int searchQueue(PageQueue &Q, int data)

{

int pos = 1;

// 检验队列合理性

if (Q.front == Q.rear)

{

cout << "Error with Queue in DeQueue" << endl;

exit(-1);

}

// 初始化队列临时元素

PageList p = (PageList)malloc(sizeof(Page));

if (!p)

{

cout << "Error with malloc in searchQueue" << endl;

exit(-1);

}

// 遍历寻找目标元素队列下标

p = Q.front->next;

while (p != NULL && data != p->data)

{

p = p->next;

pos++;

}

if (p == NULL)

{

return -1;

}

return pos;

}

bool destroyQueue(PageQueue &Q)

{

while (Q.front)

{

Q.rear = Q.front->next;

free(Q.front);

Q.front = Q.rear;

}

return true;

}

/\*

---------------------------------------------------------------------------------------------

\*/

/\*

-----------------------------------------过程方法-------------------------------------------

\*/

// ORA的方法

// 检验元素在数组是否出现

template<class T>

int checkShowUp(T&v, int target, int begin, int end) // 检验元素在数组是否出现

{

// 获取元素集长度

size\_t length = sizeof(v) / sizeof(int);

if (end > length || begin < 0 || begin > end)

{

cout << "Error with the pos" << endl;

return -1;

}

// 设置showup标志

bool flag = false;

// 遍历检验最早出现的

for (int i = begin; i < end; i++)

{

if (v[i] == target)

{

return i;

}

}

return -1;

}

// 得到最远的页面出现距离

template<class T>

int getMostFar(T&v, int pos, int target)

{

// 获取元素集长度

size\_t length = sizeof(v) / sizeof(int);

// 判断是否有目标

if (checkShowUp(v, target, pos, length) != -1)

{

return checkShowUp(v, target, pos, length) - pos;

}

else

{

return length - pos + 1;

}

}

// 获取数组中最大数的下标

template<class T>

int getPosOfMost(T&v)

{

// 获取元素集长度

size\_t length = sizeof(v) / sizeof(int);

// 得到最大的元素

int max = 0;

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

{

if (v[i] > max)

{

max = v[i];

}

}

// 找到最大元素的下标

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

{

if (v[i] == max)

{

return i;

}

}

return -1;

}

// PRA的方法

template<class T>

int getRand(T&v)

{

size\_t length = sizeof(v) / sizeof(int);

Sleep(1000);

srand(time(NULL));

return rand() % length;

}

// FIRST\_CLOCK方法

int searchAndSetAccess(PageList &Q)

{

int pos = 1;

PageList p = (PageList)malloc(sizeof(Page));

if (!p)

{

cout << "Error with malloc in searchAndSetAccess" << endl;

return -1;

}

p = Q->next;

while (p->access != 0)

{

p->access = 0;

p = p->next;

if (p == Q)

{

p = p->next;

}

pos++;

if (pos > F)

{

pos = 1;

}

}

return pos;

}

bool replacePage(PageList &L, int pos, int data)

{

if (L->next == L)

{

cout << "Error with the input in replace page" << endl;

exit(-1);

}

PageList p = (PageList)malloc(sizeof(Page));

PageList q = (PageList)malloc(sizeof(Page));

q->data = data;

p = L;

for (int j = 0; j < pos - 1; j++)

{

p = p->next;

}

q->next = p->next->next;

p->next = q;

q->access = true;

q->modify = false;

return true;

}

// SECOND\_CLOCK方法

int searchDatawithModify(PageList &L)

{

int pos\_modify = 1;

PageList p = (PageList)malloc(sizeof(Page));

p = L->next;

// 扫描访问位与改订位同时为0

while (p != L)

{

if (p->access == 0 && p->modify == 0)

break; //找到

p = p->next;

pos\_modify++;

}

if (p == L)

{

pos\_modify = 1;

p = L->next;

//扫描访问位为0，改进位为1的结点，同时修改访问过的结点的访问位为0

while (p != L)

{

if (p->access != 0)

p->access = 0;

else if (p->modify == 1)

break;

p = p->next;

pos\_modify++;

}

}

if (p == L)

{

pos\_modify = 1;

p = L->next;

//扫描A=0并且M=0的结点

while (p != L)

{

if (p->access == 0 && p->modify == 0)

break;

p = p->next;

pos\_modify++;

}

if (p == L)

{

pos\_modify = 1;

p = L->next;

//扫描A=0并且M=1的结点

while (p != L)

{

if (p->access != 0)

p->access = 0;

else if (p->modify == 1)

break;

p = p->next;

pos\_modify++;

}

}

}

return pos\_modify;

}

/\*

---------------------------------------------------------------------------------------------

\*/

/\*

-----------------------------------------过程函数-------------------------------------------

\*/

void OPT()

{

cout << "\nExcute the function named Optimal replacement algorithm..." << endl;

int total\_interupt = 0;

// 物理内存空位读入引用串

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

{

M[i] = referenceString[i];

}

cout << "初始物理内存页面： ";

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

{

cout << M[i] << " ";

}

cout << endl;

for (int i = F; i < P; i++)

{

bool flag = true; // 缺页中断标志

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

{

if (referenceString[i] == M[j])

{

flag = false;

break;

}

}

// 判断缺页中断标志是否发生

if (flag)

{

total\_interupt++;

// 获取物理内存中最远的

int dist[F];

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

{

dist[j] = getMostFar(referenceString, i, M[j]);

}

// 判断抛弃页面下标

int drop = getPosOfMost(dist);

// 置换页面

M[drop] = referenceString[i];

}

cout << "新页：" << referenceString[i] << endl;

cout << "当前物理内存页面情况：";

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

{

cout << M[j] << " ";

}

cout << endl;

}

cout << "ORA method`s interupt is " << total\_interupt << endl;

interrupt[0] = ((float)total\_interupt / 20.0)\*100.0;

}

void PRA()

{

cout << "\nExcuted the function named Random replacement algorithm..." << endl;

int total\_interrupt = 0;

// 物理内存空位读入引用串

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

{

M[i] = referenceString[i];

}

cout << "初始物理内存页面： ";

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

{

cout << M[i] << " ";

}

cout << endl;

for (int i = F; i < P; i++)

{

bool flag = true; // 缺页中断标志

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

{

if (referenceString[i] == M[j])

{

flag = false;

break;

}

}

// 判断缺页中断标志是否发生

if (flag)

{

total\_interrupt++;

// 判断抛弃页面下标

int drop = getRand(M);

// 置换页面

M[drop] = referenceString[i];

}

cout << "\n新页：" << referenceString[i] << endl;

cout << "\n当前物理内存页面情况：";

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

{

cout << M[j] << " ";

}

cout << endl;

}

cout << "\nPRA method`s interupt is " << total\_interrupt << endl;

interrupt[1] = ((float)total\_interrupt / 20.0)\*100.0;

}

void FIFO()

{

cout << "\nExcuted the function named Firstly input Firstly output algorithm..." << endl;

// 缺页中断计数

int total\_interrupt = 0;

// 初始化页面队列元素

PageQueue Q;

if (!initQueue(Q))

{

cout << "FIFO method interrupted by initQueue" << endl;

system("pause");

}

// 初始进入物理内存空位页面

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

{

if (!EnQueue(Q, referenceString[i]))

{

cout << "FIFO method interrupted by EnQueue" << endl;

system("pause");

}

}

cout << "初始物理页面：";

// 初始化临时输出

PageList p = (PageList)malloc(sizeof(Page));

p = Q.front->next;

// 输出初始物理页面

while(p!=NULL)

{

cout << p->data << " ";

p = p->next;

}

cout << endl;

for (int i = F; i < P; i++)

{

// 查询队列中是否有存在页面

if (searchQueue(Q, referenceString[i]) == -1)

{

int resultPage = -1;

// 删除物理内存页面

DeQueue(Q, resultPage);

// 添加虚拟内存页面

EnQueue(Q, referenceString[i]);

total\_interrupt++;

}

cout << "\n更新页面：" << referenceString[i] << endl;

// 更新队列

p = Q.front->next;

// 输出当前物理内存

cout << "当前物理内存：";

for (int j = 0; p != NULL && j < F; j++)

{

cout << p->data << " ";

p = p->next;

}

}

cout << "\nFIFO method`s interupt is " << total\_interrupt << endl;

interrupt[2] = ((float)total\_interrupt / 20.0)\*100.0;

free(p);

destroyQueue(Q);

}

void LRU()

{

cout << "\nExcuted the function named Firstly input Firstly output algorithm..." << endl;

// 缺页中断计数

int total\_interrupt = 0;

// 初始化页面队列元素

PageQueue Q;

int QNode\_num = 0;

// 初始化队列

initQueue(Q);

if (!initQueue(Q))

{

cout << "LRU method interrupted by initQueue" << endl;

system("pause");

}

// 初始化物理内存

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

{

if (!EnQueue(Q, referenceString[i]))

{

cout << "LRU method interrupted by EnQueue" << endl;

system("pause");

}

}

cout << "\n初始物理页面：";

PageList p = (PageList)malloc(sizeof(Page));

p = Q.front->next;

for (int i = 0; i < F&&p != NULL; i++)

{

cout << p->data << " ";

p = p->next;

}

cout << endl;

for (int i = F; i < P; i++)

{

// 没有找到目标元素

int pos = searchQueue(Q, referenceString[i]);

if (pos == -1)

{

int resultPage = -1;

DeQueue(Q, resultPage);

EnQueue(Q, referenceString[i]);

total\_interrupt++;

}

// 找到同页面将其删除并调到最近

else

{

int resultPage = -1;

PageQueue temp;

if (!initQueue(temp))

{

cout << "LRU method interrupted by initQueue" << endl;

system("pause");

}

if (!DeQueueOne(Q, resultPage, pos, temp))

{

cout << "Error with DeQueueOne in LRU" << endl;

system("pause");

}

else

{

Q = temp;

EnQueue(Q, resultPage);

}

}

cout << "\n更新页面：" << referenceString[i] << endl;

// 更新队列

p = Q.front->next;

// 输出当前物理内存

cout << "\n当前物理内存：";

for (int j = 0; p != NULL && j < F; j++)

{

cout << p->data << " ";

p = p->next;

}

}

cout << "\nLRU method`s interupt is " << total\_interrupt << endl;

interrupt[3] = ((float)total\_interrupt / 20.0)\*100.0;

free(p);

destroyQueue(Q);

}

void FIRST\_CLOCK()

{

cout << "\nExcuted the function named Firstly input Firstly output algorithm..." << endl;

// 缺页中断计数

int total\_interrupt = 0;

PageList L;

creatList(L);

PageList p = (PageList)malloc(sizeof(Page));

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

{

insertList(L, referenceString[i]);

setAccessBit(L, i + 1, true);

setModifyBit(L, i + 1, false);

}

cout << "\n初始物理页面：";

for (p = L->next; p != L; p = p->next)

{

cout << p->data << " ";

}

cout << endl;

cout << "\n物理页面访问位：";

for (p = L->next; p != L; p = p->next)

{

cout << p->access << " ";

}

cout << endl;

for (int i = F; i < P; i++)

{

int searchPos = searchListwithData(L, referenceString[i]);

if (searchPos == -1)

{

int pos = searchAndSetAccess(L);

if (!replacePage(L, pos, referenceString[i]))

{

cout << "Error in the replacePage" << endl;

}

total\_interrupt++;

}

else

setAccessBit(L, searchPos, true);

cout << "更新页面：" << referenceString[i] << endl;

cout << "物理页面：";

for (p = L->next; p != L; p = p->next)

{

cout << p->data << " ";

}

cout << endl;

cout << "访问位：";

for (p = L->next; p != L; p = p->next)

{

cout <<p->access << " ";

}

cout << endl;

}

cout << "\nFIRST\_CLOCK method`s interupt is " << total\_interrupt << endl;

interrupt[4] = ((float)total\_interrupt / 20.0)\*100.0;

free(p);

destroyList(L);

}

void SECOND\_CLOCK()

{

cout << "\nExcuted the function named SecondClock algorithm..." << endl;

// 缺页中断计数

int total\_interrupt = 0;

PageList L;

creatList(L);

PageList p = (PageList)malloc(sizeof(Page));

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

{

insertList(L, referenceString[i]);

}

cout << "初始物理页面：";

for (p = L->next; p != L; p = p->next)

{

cout << p->data << " ";

}

cout << endl;

// 随机modify页面

Sleep(1000);

srand(time(NULL)); //设置时间种子

int m = rand() % F;

setModifyBit(L, m, true);

cout << "访问位与修改位：";

for (p = L->next; p != L; p = p->next)

{

cout <<p->access<<","<< p->modify << " ";

}

cout << endl;

for (int i = F; i < P; i++)

{

int searchPos = searchListwithData(L, referenceString[i]);

if (searchPos == -1)

{

int pos\_modify = searchDatawithModify(L);

if (!replacePage(L, pos\_modify, referenceString[i]))

{

cout << "Error with replacePage in SECOND\_CLOCK" << endl;

exit(-1);

}

total\_interrupt++;

}

else

setAccessBit(L, searchPos, true);

cout << "更新页：" << referenceString[i] << endl;

cout << "物理页面：";

for (p = L->next; p != L; p = p->next)

{

cout << p->data << " ";

}

cout << endl;

Sleep(1000);

cout << "修改页（内存中序号）：" << m << endl;

setModifyBit(L, m, true);

cout << "访问位与改进位：";

p = L->next;

while (p != L)

{

cout << p->access << "," << p->modify << " ";

p = p->next;

}

cout << endl;

}

cout << "FIRST\_CLOCK method`s interupt is " << total\_interrupt << endl;

interrupt[5] = ((float)total\_interrupt / 20.0)\*100.0;

destroyList(L);

}

/\*

---------------------------------------------------------------------------------------------

\*/

int main()

{

getReferenceString(0,5,5,0.6);

OPT();

PRA();

FIFO();

LRU();

FIRST\_CLOCK();

SECOND\_CLOCK();

cout << "缺页率:" << endl;

cout << "\n最佳置换: " << interrupt[0] << "%" << endl;

cout << "\n随机置换: " << interrupt[1] << "%" << endl;

cout << "\n先进先出置换: " << interrupt[2] << "%" << endl;

cout << "\n最近最久未使用置换: " << interrupt[3] << "%" << endl;

cout << "\nCLOCK置换: " << interrupt[4] << "%" << endl;

cout << "\n改进CLOCK置换: " << interrupt[5] << "%" << endl;

system("pause");

return 1;

}