****

**实 验 报 告**

**实验课程： 操作系统课程实践**

**学生姓名： 马星**

**学 号： 5418122020**

**专业班级： 计算机科学与技术(卓越)221班**

**2024年11月1日**

** 南昌大学实验报告**

**---（5）存储管理的模拟实现**

学生姓名： 马星 学 号： 541812020 专业班级：计算机科学与技术(卓越)221班

实验类型：□验证 □ 综合 ■设计 □ 创新 实验日期： 2024/12/6 实验成绩：

# 一、实验目的

存储管理的主要功能之一是合理地分配空间。请求页式管理是一种常用的虚拟存储管理技术。本实验的目的是通过请求页式存储管理中页面置换算法模拟设计，了解虚拟存储技术的特点，掌握请求页式管理的页面置换算法。

# 二、实验内容

1．过随机数产生一个指令序列，共320条指令。其地址按下述原则生成：

①50%的指令是顺序执行的；

②25%的指令是均匀分布在前地址部分；

③25%的指令是均匀分布在后地址部分；

#具体的实施方法是：

* + 1. 在[0，319]的指令地址之间随机选区一起点M;
    2. 顺序执行一条指令，即执行地址为M+1的指令；
    3. 在前地址[0，M+1]中随机选取一条指令并执行，该指令的地址为M’;
    4. 顺序执行一条指令，其地址为M’+1；
    5. 在后地址[M’+2，319]中随机选取一条指令并执行；
    6. 重复A—E，直到执行320次指令。

2．指令序列变换成页地址流

设：（1）页面大小为1K；

* 1. 用户内存容量为4页到32页；
  2. 用户虚存容量为32K。

在用户虚存中，按每K存放10条指令排列虚存地址，即320条指令在虚存中的存放方式为：

第0条—第9条指令为第0页（对应虚存地址为[0，9]）；

第10条—第19条指令为第1页（对应虚存地址为[10，19]）；

。。。。。。。。。。。。。。。。。。。。。

第310条—第319条指令为第31页（对应虚存地址为[310，319]）；

按以上方式，用户指令可组成32页。

3. 计算并输出下述各种算法在不同内存容量下的命中率。

* + 1. FIFO先进先出的算法
    2. LRU最近最少使用算法

C．LFU最少访问页面算法

# 三、实验要求

1、需写出设计说明；

2、设计实现代码及说明

3、运行结果；

# 四、主要实验步骤

## (1) 生成320条指令

// (1) 生成320条随机指令序列

int ICnt = 320;// 指令数  
vector<int> ISeq;// 指令序列

srand((unsigned) time(nullptr)); // 初始化随机数种子  
while (ISeq.size() < ICnt) {  
 int m = randInt(1, ICnt);  
 ISeq.push\_back(m);  
 ISeq.push\_back(randInt(1, m + 1));  
 ISeq.push\_back(randInt(m + 1, ICnt));  
}  
cout << "生成指令序列:";  
for (int i: ISeq) cout << i << ",";  
cout << "\b\n";

## (2) 指令序列转换为地址流

// (2) 指令序列转换为页地址

int pageSize = 1 \* 1024;// 页大小  
int virtualCapacity = 32 \* 1024;// 虚存容量  
vector<int> page;// page[i]:第i条指令存在的页号

int pageCnt = virtualCapacity / pageSize;// 总页数 = 容量 / 页大小  
int per = ICnt / pageCnt; // 每页指令数 = 总指令数 / 总页数  
for (int i: ISeq) page.push\_back(i / per); // 指令的页地址 = 指令地址/每页指令数

## (3) 页面置换算法

父类:

class PageAlgorithm {// 页面置换算法  
public:

int capacity;// 内存容量

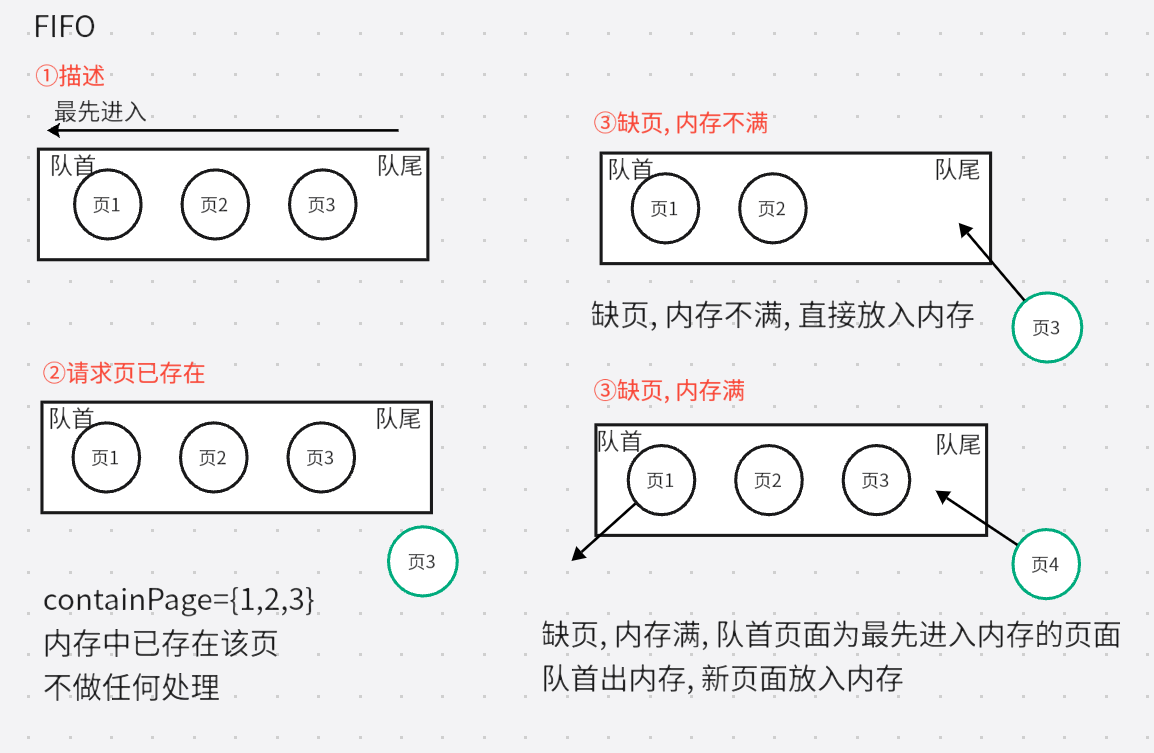
int total = 0, missingCnt = 0; // (统计) 总访问次数、缺页次数

explicit PageAlgorithm(int capacity) : capacity(capacity) {}

virtual bool get(int page) = 0;// 查询页是否在内存中  
 virtual void put(int page) = 0;// 将页放入内存, 若内存满则置换  
  
 double getPageFault() const {// 缺页率   
 return missingCnt \* 1.0 / total;  
 }  
};

### 1. FIFO算法

FIFO是先进先出的置换算法, 可以采用队列实现, 当内存满时, 移除最先进入的页面(移除队首元素)



class FIFO : public PageAlgorithm {  
public:  
 queue<int> q;// 队列, 先进先出  
 set<int> s;// containPage, 页是否存在于内存   
  
 explicit FIFO(int capacity) : PageAlgorithm(capacity) {}  
  
 void put(int key) override {  
 if (s.size() == capacity) {// 内存已满  
 int &r = q.front();// 移除队首(先进先出)  
 q.pop();  
 s.erase(r);// 将页面r移出内存  
 }  
 s.insert(key);// 将页面key加入内存  
 q.push(key);  
 }  
  
 bool get(int key) override {  
 total++;  
 if (s.find(key) != s.end()) {// 命中  
 return true;  
 } else {//缺页  
 missingCnt++;  
 return false;  
 }  
 }  
};  
  
#endif //OP\_FIFO\_H

### 2. LRU算法

LRU:最久未使用

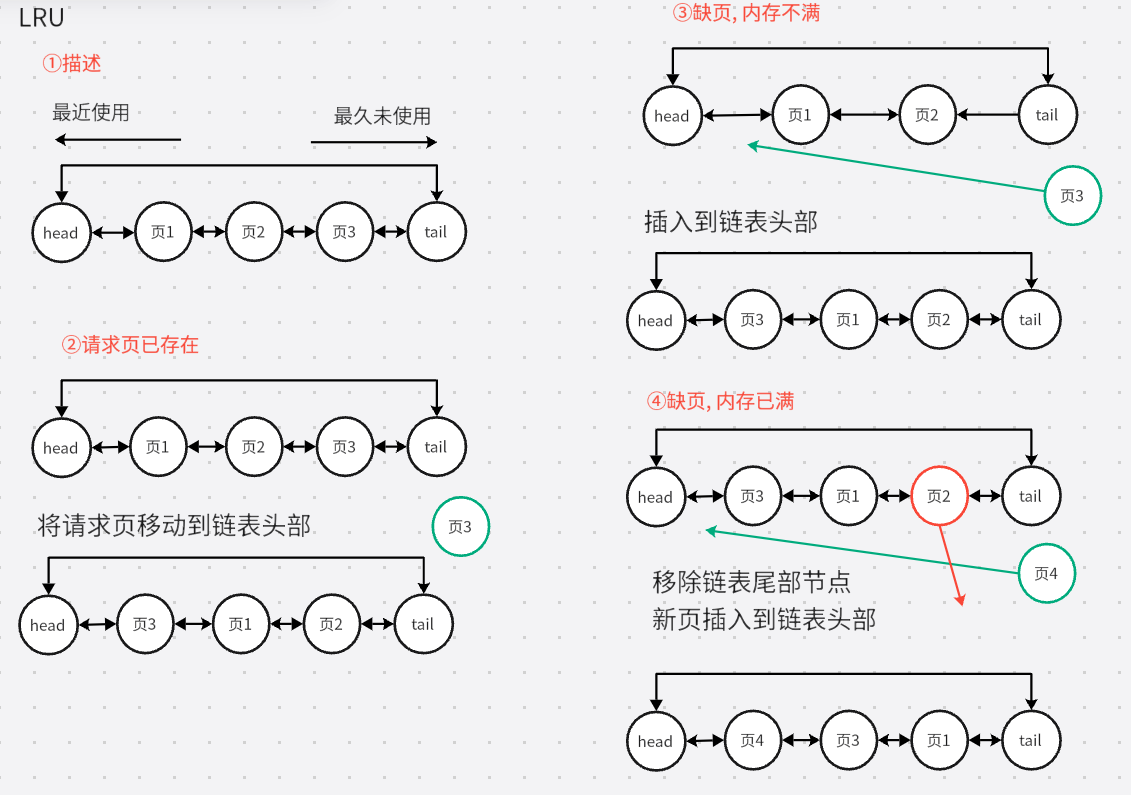
使用双向链表实现, 链表的头部表示最近使用的页面, 链表的尾部表示最久未使用的页面

可以类比为看书, 链表看作一摞书, 书堆的顶部就是最近看过的, 每次看一本书就是将它从书堆里抽出来, 看过后放在顶部

当请求一个页面时:

若页面已存在于内存中, 则将其放置在链表的头部, 表示该页是最近使用的

若缺页, 如果内存已满, 则将尾部节点(最久未使用的页面)移除, 将新页面放入链表头部



class LRU : public PageAlgorithm {  
public:  
 LinkedList linkedList;// 页节点链  
 map<int, LNode \*> nodeMap;// 页码 -> 页节点  
  
 explicit LRU(int capacity) : PageAlgorithm(capacity) {}  
  
 void put(int key) override {  
 if (nodeMap.size() == capacity) { // 内存已满  
 LNode \*r = linkedList.removeLast();// 移除链表尾端(最久未使用)  
 nodeMap.erase(r->key); // 移出内存  
 free(r);  
 }  
 auto \*node = new LNode(key); // 新页面放入内存  
 nodeMap[key] = node;  
 linkedList.addFirst(node);  
 }  
  
 bool get(int key) override {  
 total++;  
 if (nodeMap.find(key) != nodeMap.end()) { // 命中  
 LNode \*&pNode = nodeMap[key];  
 linkedList.remove(pNode);// 更新最近使用时间  
 linkedList.addFirst(pNode);  
 return true;  
 } else {// 缺页  
 missingCnt++;  
 return false;  
 }  
 }  
  
};

### 3. LFU算法

LFU:最不经常使用

使用多个链表, 第一个链表对应使用1次, 第二个链表对应使用2次, 且链表的头部表示最近使用的, 链表的尾部表示最久未使用的

继续用看书类别, 每个链表看作一摞书,且有对应的频次标记, 书堆的顶部是最近看过的, 每次看一本书就是将它从书堆里抽出来, 看过后放在右边一摞书的顶部

当访问某页时:

①该页已存在于内存中: 该页的使用次数+1, 从原频次的链表移动+1频次的链表

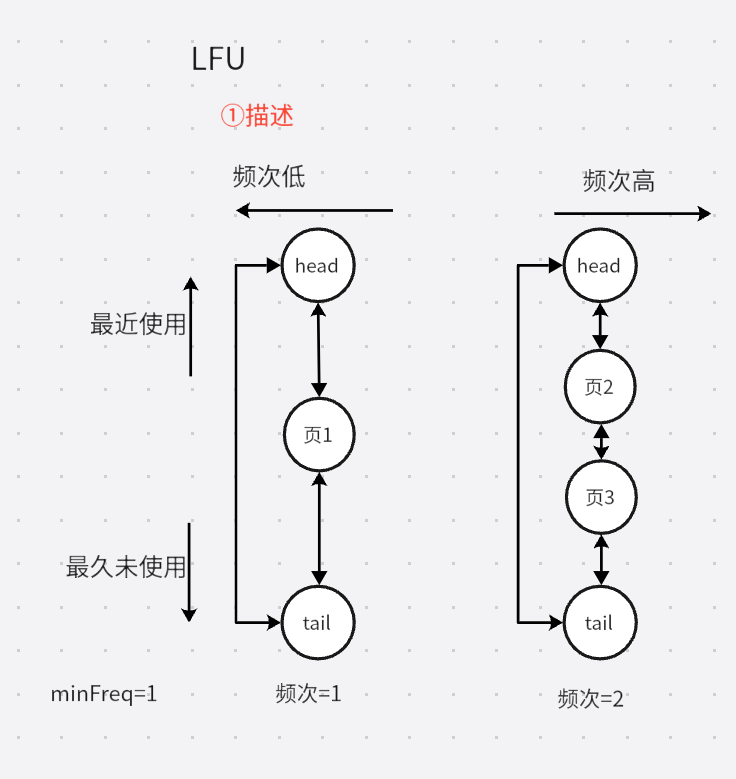
②缺页, 内存不满: 将页面放入”使用1次”对应的链表的头部

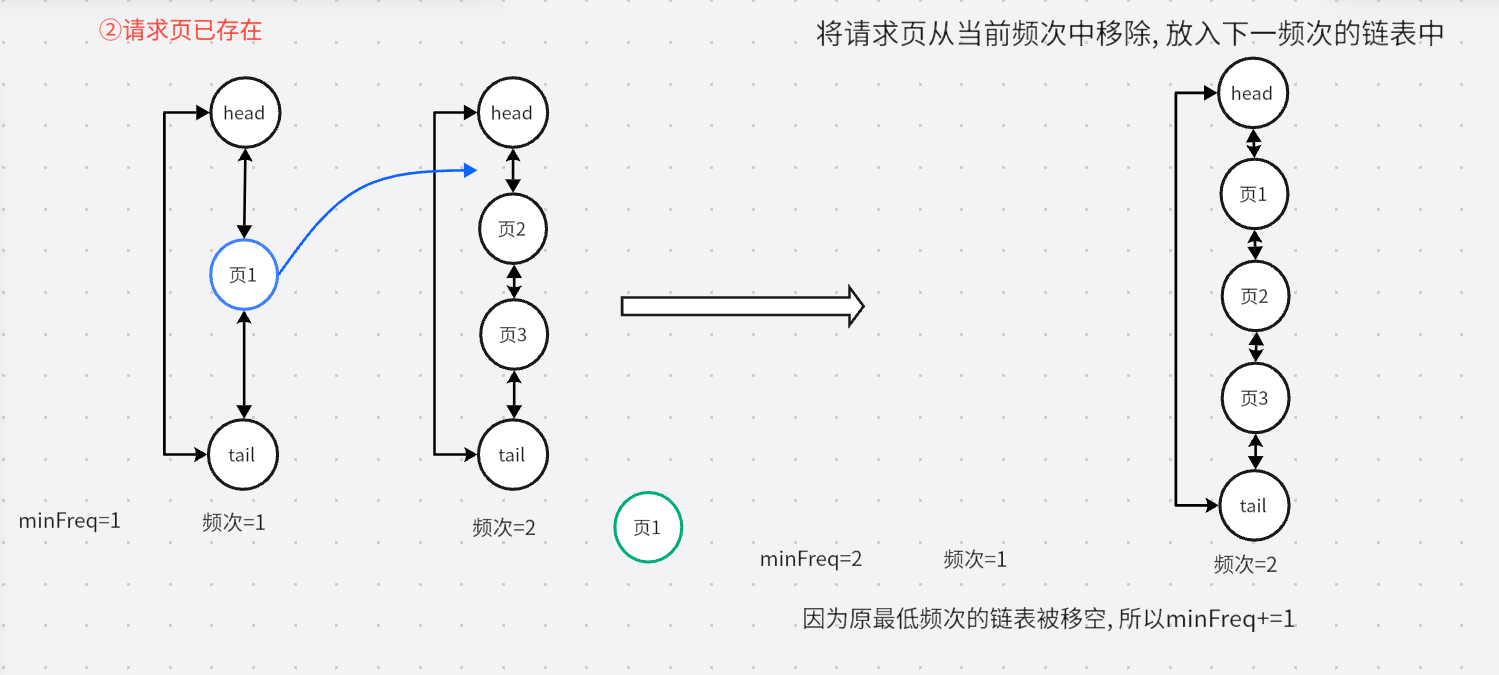
③缺页, 内存已满: 移出当前最低频次对应的链表的尾部节点, 再将页面放入”使用1次”对应的链表的头部

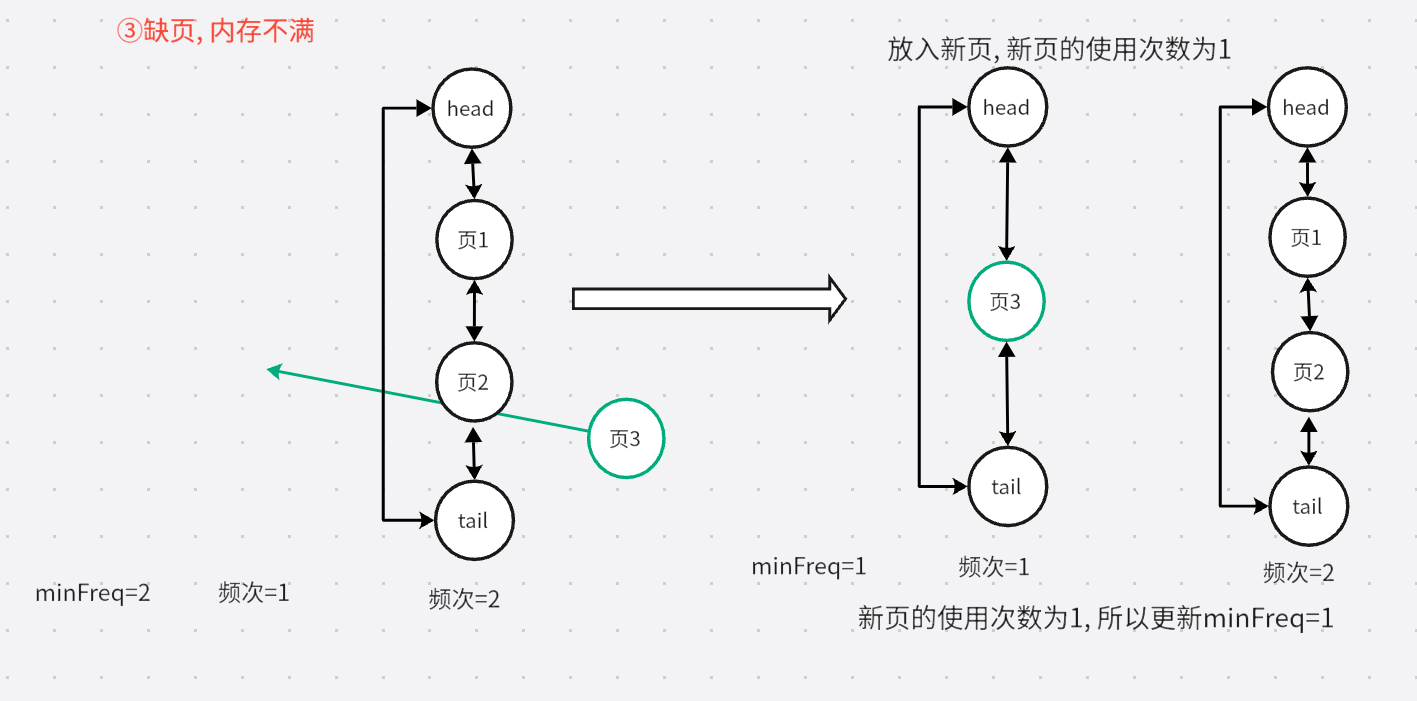
如何维护” 当前最低频次”: 用int minFreq记录当前最低频次

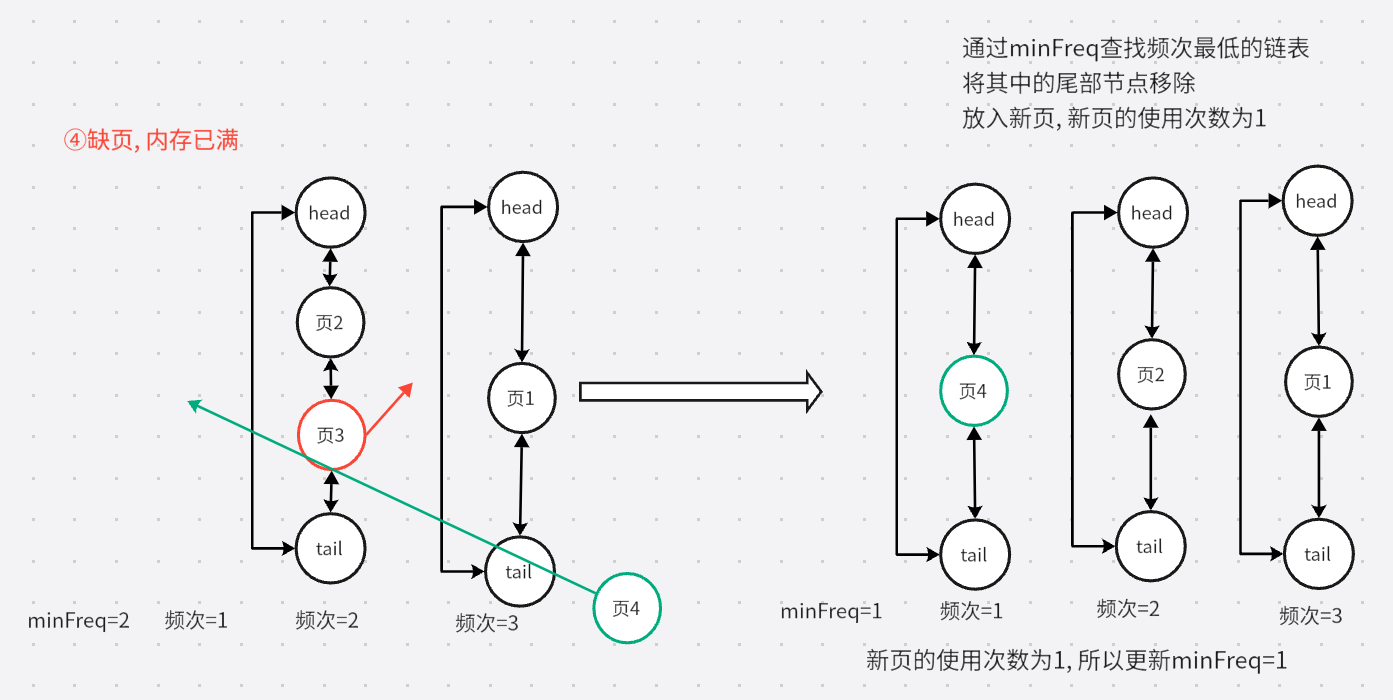
①页存在时: 需要将页面从当前链表移除并放置在下一个链表中, 如果此时minFreq恰好等于该页面使用次数, 且恰好当前链表被移空, 则minFreq+=1, 因为小于minFreq没有节点了, 下一个非空链表是刚移动的页面

②缺页时, 需要放入新页, 而新页的使用次数为1, 直接置minFreq=1









class LFU : public PageAlgorithm {  
public:  
 map<int, Node \*> keyToNode;// 页关键字 -> 页节点  
 map<int, Node \*> freqToDummy;// 频次 -> 页节点链  
 int minFreq = 0;// 最少的频次  
 explicit LFU(int capacity) : PageAlgorithm(capacity) {}  
  
 void put(int key) override {  
 if (keyToNode.size() == capacity) { // 内存已满, 移除freq最小的页面里最久不使用的  
 Node \*dummy = freqToDummy[minFreq];// 最小频次的页节点链  
 Node \*r = dummy->prev; // 尾端节点(最久不使用)  
 keyToNode.erase(r->key);// 移出内存  
 remove(r);  
 free(r);  
 if (dummy->prev == dummy) { // 被移空了  
 freqToDummy.erase(minFreq); // 移除空链表  
 // (最小频次置为1, 待会更新)  
 }  
 }  
 Node \*node = new Node(key); // 新页面放入内存  
 keyToNode[key] = node;  
 pushFront(1, node); // 访问1次, 最近访问(最上面)  
 minFreq = 1;// 最小频次置为1  
 }  
  
 bool get(int key) override {  
 total++;  
 if (keyToNode.find(key) != keyToNode.end()) { // 命中  
 Node \*node = keyToNode[key];  
 // 把页节点从当前频次的页节点链中移除, 放在 频次+1 的页节点链中  
 remove(node);  
 Node \*dummy = freqToDummy[node->freq];  
 if (dummy->prev == dummy) { // 频次为freq的链被移空了  
 freqToDummy.erase(node->freq);  
 if (minFreq == node->freq) {// freq是频次最低的, 更新最低频次  
 minFreq++;// 链一定非空 -> 刚放入的节点频次为minFreq+1  
 }  
 }  
 node->freq++;  
 pushFront(node->freq, node); // 最近访问, 放在同频次的最上面  
 return true;  
 } else {// 缺页  
 missingCnt++;  
 return false;  
 }  
  
 }  
  
  
 /\*\*  
 创建一个新的双向链表  
 \*/  
 static Node \*newList() {  
 Node \*dummy = new Node(0); // 哨兵节点  
 dummy->prev = dummy;  
 dummy->next = dummy;  
 return dummy;  
 }  
  
 /\*\*  
 在链表头添加一个节点（把一本书放在最上面）  
 \*/  
 void pushFront(int freq, Node \*x) {  
 if (freqToDummy.find(freq) == freqToDummy.end()) {  
 freqToDummy[freq] = newList();  
 }  
 Node \*dummy = freqToDummy[freq];//链表头  
 x->prev = dummy;  
 x->next = dummy->next;  
 x->prev->next = x;  
 x->next->prev = x;  
 }  
  
 /\*\*  
 删除一个节点（抽出一本书）  
 \*/  
 static void remove(Node \*x) {  
 Node \*prev = x->prev;  
 Node \*next = x->next;  
 prev->next = next;  
 next->prev = prev;  
 }  
};

## (4) 测试代码

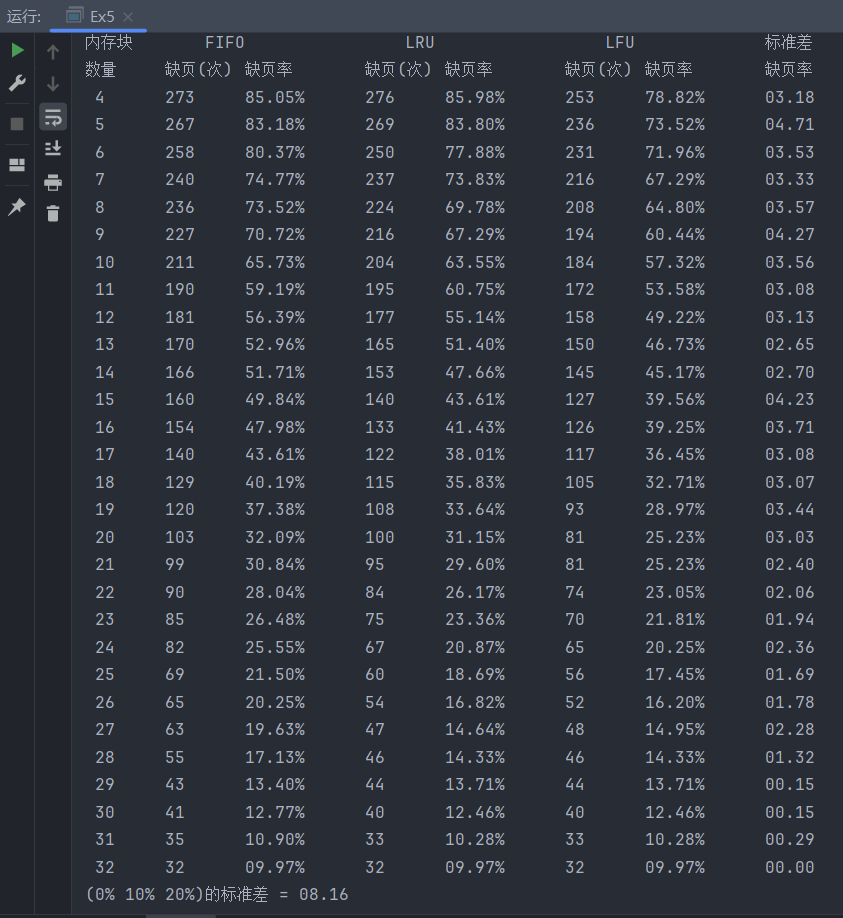
// 测试   
cout << "内存块" << "\t\t" << "FIFO" << "\t\t\t\t" << "LRU" << "\t\t\t\t\t"  
 << "LFU" << "\t\t\t\t" << "标准差" << endl;  
cout << "数量" << "\t\t";  
for (int i = 0; i < 3; i++) cout << "缺页(次)" << "\t缺页率\t\t";  
cout << "缺页率";  
cout << endl;  
for (int capacity = userRAM.first; capacity <= userRAM.second; capacity++) {  
 test(capacity);  
}  
cout << "(0% 10% 20%)的标准差 = " << percentage(standard(0.00, 0.10, 0.20));

# 五、实验数据及处理结果

运行结果如图

分别测试了内存大小为4至32的缺页情况, 统计了各算法的缺页次数与缺页率

并计算缺页率的标准差, 用标准差比较三个算法的缺页率离散程度



从表中可以看出, 在相同的指令序列输入下, 内存容量越大, 缺页率越低, 越不容易导致缺页

在各个内存容量下, 三个算法的缺页率标准差未超过5, 以缺页率为0%,10%,20%的标准差=8.16比较, 各算法的缺页率相差并不算大

更细致的比较, 可以发现在大多数情况下FIFO的缺页率 > LRU的缺页率 > LFU的缺页率

说明LFU能够更好地预测未来的页面访问模式，从而减少缺页次数

综上所述: 增加内存块的数量可以有效降低缺页率，提高系统性能；同时，选择合适的页面置换算法也是提高系统性能的关键因素之一。

# 六、实验体会或对改进实验的建议

在本次存储管理的模拟实现实验中，我深刻体会到了虚拟存储技术和页面置换算法的工作原理及其重要性。通过动手编写代码来模拟请求页式管理中的页面置换过程，不仅让我从理论层面理解了相关概念，而且通过实践操作加深了对这些知识的认识和应用能力。

首先，我认识到合理分配和管理内存空间的重要性。在有限的物理内存条件下，如何有效地利用虚拟内存和磁盘空间，确保程序运行的高效性和稳定性是存储管理的关键任务。通过模拟指令序列生成和地址变换的过程，我更加清楚地看到了操作系统在背后所做的大量复杂工作。

其次，通过实现FIFO、LRU和LFU三种页面置换算法，我了解到不同的算法适应不同的应用场景。每种算法都有其优缺点：FIFO算法简单易实现，但在特定情况下可能会出现Belady异常现象；LRU算法能较好地适应程序执行的局部性原理，但需要更多的系统开销来维护使用情况记录；而LFU算法则侧重于减少频繁访问页面的置换次数，但可能对新页面有偏见。

最后，实验中的数据收集与分析让我意识到系统性能优化是一个持续的过程。通过对比不同内存容量下各算法的命中率，我发现增加内存容量确实可以降低缺页率，但同时也要权衡成本和实际需求。选择合适的页面置换策略对于提升系统性能至关重要。

这次实验不仅增强了我对操作系统存储管理理论知识的理解，也锻炼了我的编程能力和问题解决能力。通过亲身体验和实际操作，我更加深刻地感受到了计算机系统的复杂性和精妙之处。在未来的学习中，我将更加注重理论与实践的结合，不断提升自己的专业技能。

# 七、参考资料

《计算机操作系统实验指导》