安徽大学人工智能学院《操作系统》实验报告

学号___WA2214014__ 姓名___杨跃浙___ 年级__大三__

【实验名称】_____实验四　虚拟存储_____

【实验内容】

1．先进先出置换算法

2．最佳置换算法

3．LRU 置换算法

1．先进先出算法

基本思想是淘汰最先进入内存的页面，即选择在内存驻留时间最长的页面予以淘汰。实现简单。按页面调入内存的先后链结为队列，设置一个替换指针，总是指向最先进入内存的页面。缺点在与进程实际运行规律不符，性能不好。

2.最佳页面置换算法

基本思想是所选择的被淘汰页面，将是以后永不使用的，或是在最长(未来)时间内不再被访问的页面。采用最佳置换算法，可保证获得最低的缺页率。

3.最近最久未使用页面置换算法

基本思想：最近未访问的页面，将来一段时间也不会访问。利用局部性原理，根据一个进程在执行过程中过去的页面访问踪迹来推测未来的行为。最近的过去 → 最近的将来　　思想：选择最近最久未使用的页面予以淘汰。利用页表中的访问字段，记录页面自上次被访问以来所经历的时间 t，需要淘汰页面时，选择在内存页面中 t 值最大的，即最近最久未使用的页面予以淘汰

【实验要求】

1、 根据上述算法的特点，分别设计数据结构。

2、 编写上述三种算法。

3、测试用例

在某请求分页管理系统中，一个作业共 5 页，作业执行时依次访问如下页面：1，4，3，1，2，5，1，4，2，1，4，5，若分配给该作业的主存块数为 3，分别采用最佳置换算法(OPT)、先进先出置换算法（FIFO）、最近最久未使用置换算法（LRU）

【实验原理】

虚拟存储技术是一种通过将程序运行时所需的内存从实际物理内存扩展到外存的机制，核心在于高效管理主存和外存之间的页面调度。其主要实现方式依赖于页面置换算法来决定在内存已满

的情况下，哪一页需要被换出。实验中涉及的三个页面置换算法（FIFO、OPT 和 LRU）各自基于不同的调度原则，性能差异取决于具体的访问序列和算法的局部性利用。

在虚拟存储系统中，内存被划分为固定大小的页面（Page），每次访问页面时，CPU 通过页表（Page Table）映射页面号到内存块号。如果所需的页面不在内存中，就会发生缺页异常（Page Fault）。缺页异常时，系统需要从外存中加载目标页面到主存，并可能根据页面置换算法的结果，淘汰当前内存中的一个页面以释放空间。页面置换的核心目标是减少缺页率，提高内存使用效率。

页面置换的性能衡量主要以缺页率（Page Fault Rate）为指标，公式如下：

$$\text{缺页率} = \frac{\text{缺页次数}}{\text{页面请求总次数}}$$

缺页率越低，说明页面置换算法性能越优。以下分块解释三种置换算法的原理。

最佳置换（OPT）算法

OPT 算法又称最优置换算法，理论上可获得最低的缺页率。它的基本思想是：淘汰未来访问时间最远的页面。在每次页面置换时，系统需要模拟未来的页面访问顺序，选择一段时间内不再使用的页面或未来使用时间最长的页面进行淘汰。然而，OPT 算法无法在实际系统中实现，因为无法预知未来的访问序列，因此通常用作评估其他算法性能的理论最优基准。

公式化描述：

$$\text{OPT 决策规则} = \arg\max_i(\text{页面未来访问时间}[i])$$

其中，页面未来访问时间表示当前页面到下次被访问的时间间隔。OPT 算法在局部性明显的访问模式下表现最优，但由于需要知道未来的访问序列，其实现仅存在于理论或模拟环境中。

先进先出（FIFO）算法

FIFO 算法是基于队列结构实现的页面置换方法，其核心思想是始终淘汰最早进入内存的页面。系统使用一个循环指针（或队列）记录页面调入的顺序，当发生缺页时，指针指向的页面即为被淘汰的目标页面。FIFO 算法实现简单且无需依赖复杂的历史信息，但是由于完全忽略页面的实际访问规律，可能导致频繁淘汰近期需要再次使用的页面。这种现象被称为“Belady 异常”。

公式化描述：

$$\text{FIFO 决策规则} = \arg\min_i(\text{页面调入时间}[i])$$

其中，页面调入时间表示每个页面进入内存的时间戳。FIFO 的优点是实现简单，缺点是可能无法适应程序的实际局部性规律。

最近最久未使用（LRU）算法

LRU 算法基于程序的“时间局部性”原理，即近期被访问的页面在未来一段时间内可能会被再次访问。LRU 通过记录每个页面的最近使用时间来实现，当需要置换页面时，选择最近最久未访问的页面进行淘汰。系统通常使用计时器或栈来实现 LRU 算法，分别

对应时间戳法和链表法。LRU 在实践中较为常用，缺点是实现复杂度较高，且需要额外存储最近使用时间的信息。

公式化描述：

$$\text{LRU 决策规则} = \arg\min_i(\text{页面最近访问时间}[i])$$

其中，页面最近访问时间表示当前页面上一次被访问的时间戳。LRU 算法在大多数程序访问模式下性能优异，但在某些极端情况下（如循环访问序列）可能表现欠佳。

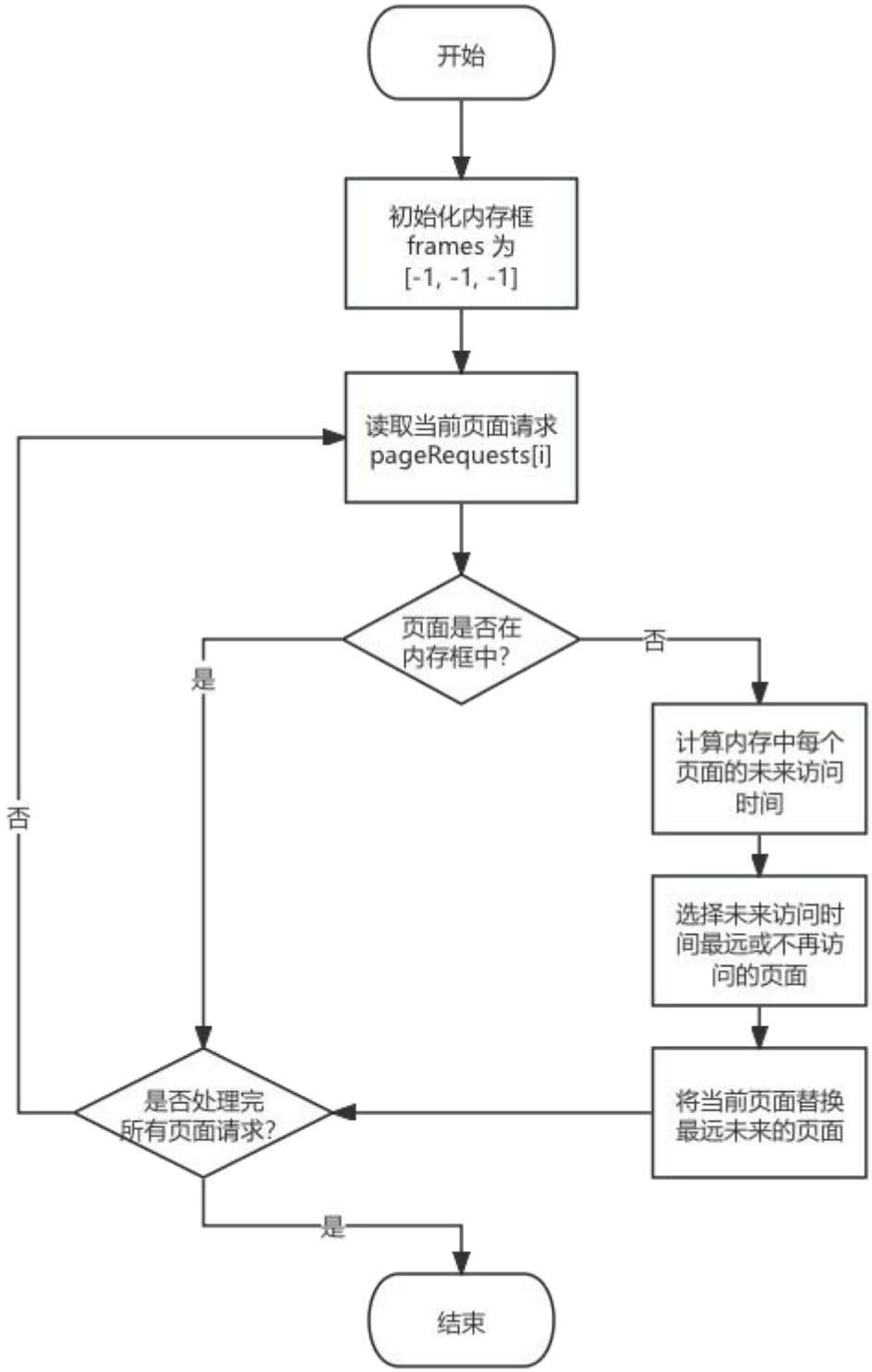
页面置换算法依赖于对程序访问规律的充分利用。程序访问规律通常表现为时间局部性和空间局部性。时间局部性意味着一个地址一旦被访问，很可能在短时间内再次被访问。空间局部性则表示地址附近的页面很可能被连续访问。FIFO 算法忽略了局部性原则，而 LRU 和 OPT 都在不同程度上尝试捕捉这种规律来优化页面替换决策。

实验通过模拟页面访问序列和页面置换过程，分析三种算法在特定场景下的缺页率表现，为虚拟存储系统的优化提供参考依据。

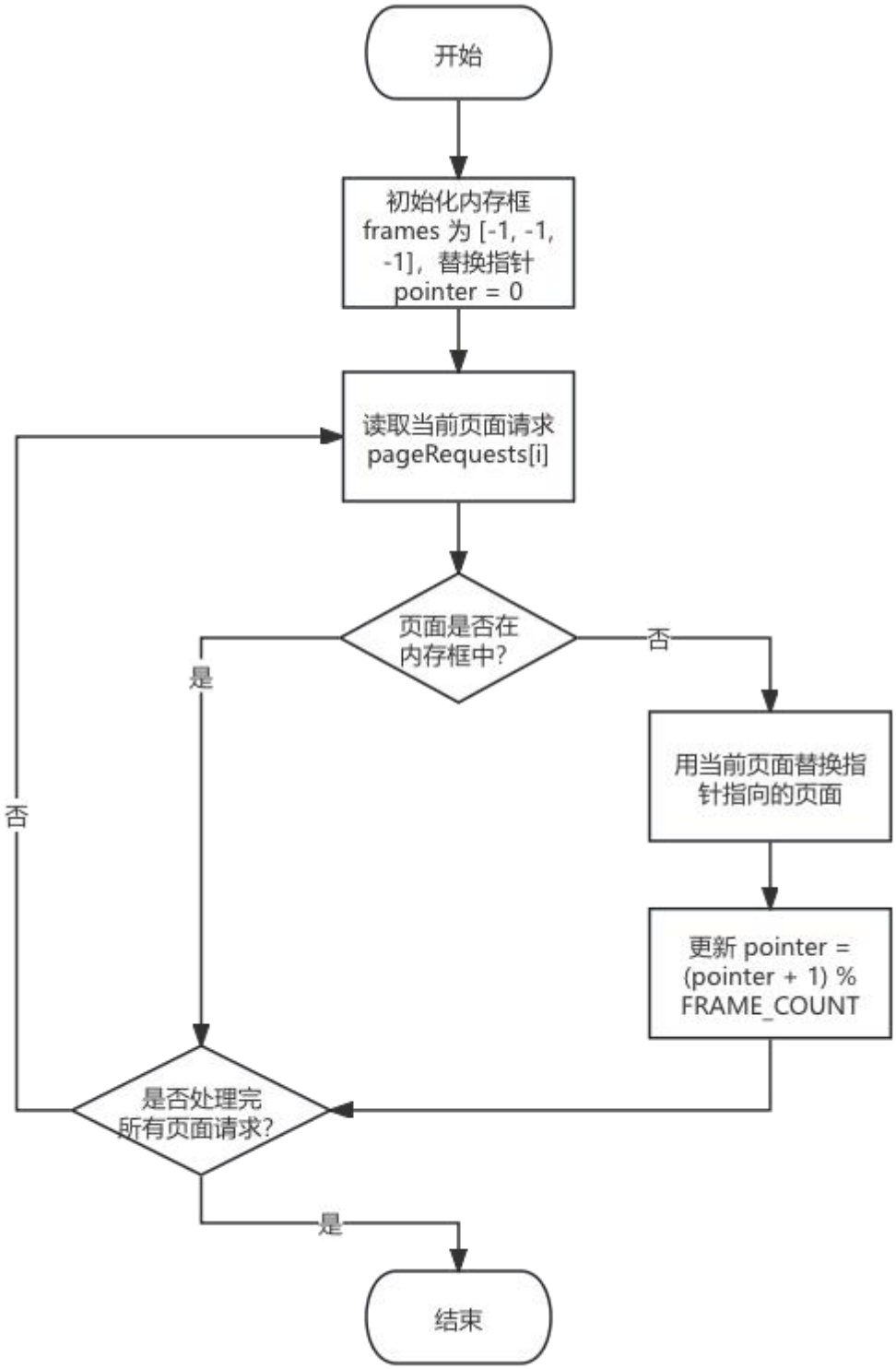
【实验步骤】

1.程序流程图

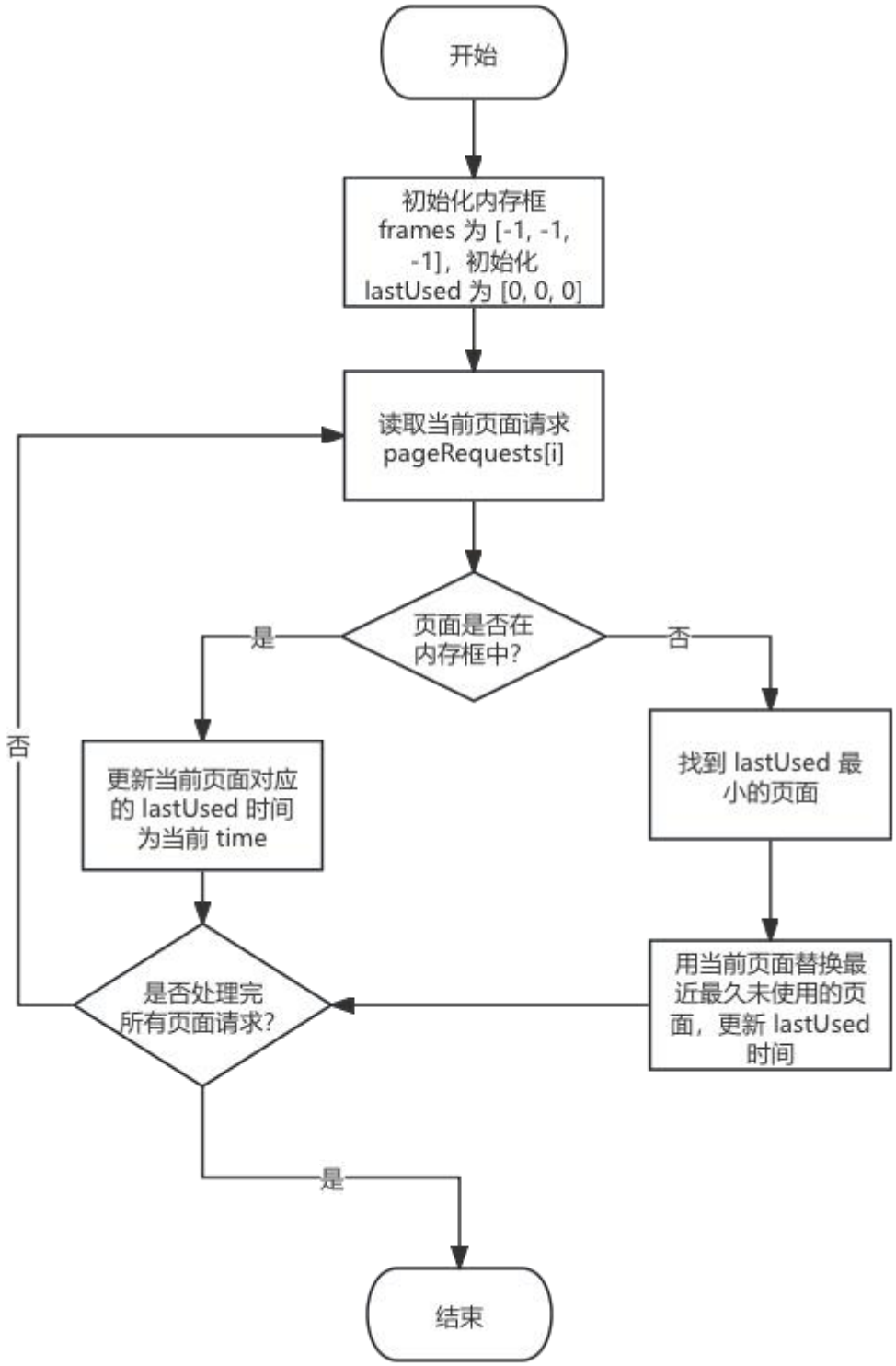
最佳置换（OPT）页面置换算法



先进先出（FIFO）页面置换算法



最近最久未使用（LRU）页面置换算法



2.核心代码

```
#include <stdio.h>
#include <stdbool.h>

// 页面请求数量应与数组中实际元素数相符
```

```
#define PAGE_REQUEST_COUNT 12 // 定义页面请求数量，需与实际页面数组匹配
int pageRequests[PAGE_REQUEST_COUNT] = {1, 4, 3, 1, 2, 5, 1, 4, 2, 1, 4, 5};

// 主存块数
#define FRAME_COUNT 3 // 定义主存块数量（帧数）

// 打印当前内存框中的页面状态
void printFrames(int frames[], int frameCount) {
for (int i = 0; i < frameCount; i++) {
if (frames[i] != -1) // 如果页面存在
printf("%d ", frames[i]);
else
printf("- "); // 空帧以 "-" 表示
}
printf("\n");
}

// 检查页面是否在内存框中
bool isPageInFrames(int frames[], int frameCount, int page)
{
for (int i = 0; i < frameCount; i++) {
if (frames[i] == page) // 遍历当前帧，查找页面
return true; // 页面已在内存中，返回 true
}
return false; // 页面不在内存中，返回 false
}

// 找到需要替换的页面索引（FIFO 算法核心）
int findFIFOIndex(int *pointer, int frameCount) {
int index = *pointer; // 替换位置由指针决定
*pointer = (*pointer + 1) % frameCount; // 更新替换指针，循环指向下一个帧
return index; // 返回需要替换的页面索引
}

// 找到需要替换的页面索引（OPT 算法核心）
```

```
int findOPTIndex(int frames[], int frameCount, int
currentPageIndex) {
int farthest = -1, replaceIndex = -1; // 初始化最远未来访问位
置和替换索引
for (int i = 0; i < frameCount; i++) { // 遍历当前帧
int j;
for (j = currentPageIndex + 1; j < PAGE_REQUEST_COUNT; j++)
{
if (frames[i] == pageRequests[j]) {
if (j > farthest) { // 找到最远未来访问的页面
farthest = j;
replaceIndex = i;
}
break; // 页面已找到，停止内部循环
}
}
if (j == PAGE_REQUEST_COUNT) { // 如果页面未来不再访问
return i; // 直接返回该页面索引
}
}
return replaceIndex; // 返回未来最晚访问的页面索引
}

// 找到需要替换的页面索引（LRU 算法核心）
int findLRUIndex(int lastUsed[], int frameCount) {
int min = lastUsed[0], index = 0; // 初始化最近最久未使用页面
的时间和索引
for (int i = 1; i < frameCount; i++) {
if (lastUsed[i] < min) { // 查找时间最小的页面
min = lastUsed[i];
index = i;
}
}
return index; // 返回最近最久未使用页面的索引
}

// 先进先出（FIFO）页面置换算法
void fifo() {
printf("\n 先进先出（FIFO）置换算法:\n");
```

```
int frames[FRAME_COUNT]; // 定义内存框
for (int i = 0; i < FRAME_COUNT; i++) frames[i] = -1; // 初始化内存框为空
int pointer = 0, pageFaults = 0; // 替换指针和缺页计数器初始化

for (int i = 0; i < PAGE_REQUEST_COUNT; i++) {
if (!isPageInFrames(frames, FRAME_COUNT, pageRequests[i]))
{
// 如果页面不在内存中，发生缺页
int replaceIndex = findFIFOIndex(&pointer, FRAME_COUNT); // 找到替换位置
frames[replaceIndex] = pageRequests[i]; // 替换页面
pageFaults++; // 缺页次数加一
}
printf("访问页面 %d: ", pageRequests[i]);
printFrames(frames, FRAME_COUNT); // 打印当前内存状态
}
printf("总缺页次数: %d\n", pageFaults);
}

// 最佳置换（OPT）页面置换算法
void opt() {
printf("\n最佳置换（OPT）置换算法:\n");
int frames[FRAME_COUNT]; // 定义内存框
for (int i = 0; i < FRAME_COUNT; i++) frames[i] = -1; // 初始化内存框为空
int pageFaults = 0; // 缺页计数器初始化

for (int i = 0; i < PAGE_REQUEST_COUNT; i++) {
if (!isPageInFrames(frames, FRAME_COUNT, pageRequests[i]))
{
// 如果页面不在内存中，发生缺页
int replaceIndex = findOPTIndex(frames, FRAME_COUNT, i); // 找到替换位置
frames[replaceIndex] = pageRequests[i]; // 替换页面
pageFaults++; // 缺页次数加一
}
printf("访问页面 %d: ", pageRequests[i]);
printFrames(frames, FRAME_COUNT); // 打印当前内存状态
```

```
}
printf("总缺页次数: %d\n", pageFaults);
}

// 最近最久未使用（LRU）页面置换算法
void lru() {
printf("\n 最近最久未使用（LRU）置换算法:\n");
int frames[FRAME_COUNT]; // 定义内存框
for (int i = 0; i < FRAME_COUNT; i++) frames[i] = -1; // 初始化内存框为空
int lastUsed[FRAME_COUNT] = {0, 0, 0}; // 初始化最近使用时间
int pageFaults = 0, time = 0; // 缺页计数器和时间步初始化

for (int i = 0; i < PAGE_REQUEST_COUNT; i++) {
time++; // 时间步进
if (!isPageInFrames(frames, FRAME_COUNT, pageRequests[i]))
{
// 页面不在内存中 -> 缺页
int replaceIndex = findLRUIndex(lastUsed, FRAME_COUNT); // 找出 LRU 页面的索引
frames[replaceIndex] = pageRequests[i]; // 用当前页面替换 LRU 页面
lastUsed[replaceIndex] = time; // 更新新装入页面的最近使用时间
pageFaults++; // 缺页次数加一
} else {
// 页面已在内存中 -> 命中
// 需要更新该页面在 lastUsed 中的时间
for (int j = 0; j < FRAME_COUNT; j++) {
if (frames[j] == pageRequests[i]) {
lastUsed[j] = time; // 更新该页面的最近使用时间
break;
}
}
}
printf("访问页面 %d: ", pageRequests[i]);
printFrames(frames, FRAME_COUNT); // 打印内存状态
}
printf("总缺页次数: %d\n", pageFaults);
}
```

```
int main() {
opt(); // 运行 OPT 算法
fifo(); // 运行 FIFO 算法
lru(); // 运行 LRU 算法
return 0;
}
```

3.运行结果

最佳置换（OPT）置换算法：
访问页面 1: 1 - -
访问页面 4: 1 4 -
访问页面 3: 1 4 3
访问页面 1: 1 4 3
访问页面 2: 1 4 2
访问页面 5: 1 4 5
访问页面 1: 1 4 5
访问页面 4: 1 4 5
访问页面 2: 1 4 2
访问页面 1: 1 4 2
访问页面 4: 1 4 2
访问页面 5: 5 4 2
总缺页次数：7

先进先出（FIFO）置换算法：
访问页面 1: 1 - -
访问页面 4: 1 4 -
访问页面 3: 1 4 3
访问页面 1: 1 4 3
访问页面 2: 2 4 3
访问页面 5: 2 5 3
访问页面 1: 2 5 1
访问页面 4: 4 5 1
访问页面 2: 4 2 1
访问页面 1: 4 2 1
访问页面 4: 4 2 1
访问页面 5: 4 2 5
总缺页次数：9

最近最久未使用（LRU）置换算法：
访问页面 1: 1 - -
访问页面 4: 1 4 -
访问页面 3: 1 4 3
访问页面 1: 1 4 3
访问页面 2: 1 2 3
访问页面 5: 1 2 5
访问页面 1: 1 2 5
访问页面 4: 1 4 5
访问页面 2: 1 4 2
访问页面 1: 1 4 2
访问页面 4: 1 4 2
访问页面 5: 1 4 5
总缺页次数：8

最佳置换（OPT）置换算法:
访问页面 1: 1 - -
访问页面 4: 1 4 -
访问页面 3: 1 4 3
访问页面 1: 1 4 3
访问页面 2: 1 4 2
访问页面 5: 1 4 5
访问页面 1: 1 4 5
访问页面 4: 1 4 5
访问页面 2: 1 4 2
访问页面 1: 1 4 2
访问页面 4: 1 4 2
访问页面 5: 5 4 2
总缺页次数: 7

先进先出（FIFO）置换算法:
访问页面 1: 1 - -
访问页面 4: 1 4 -
访问页面 3: 1 4 3
访问页面 1: 1 4 3
访问页面 2: 2 4 3
访问页面 5: 2 5 3
访问页面 1: 2 5 1
访问页面 4: 4 5 1
访问页面 2: 4 2 1
访问页面 1: 4 2 1
访问页面 4: 4 2 1
访问页面 5: 4 2 5
总缺页次数: 9

最近最久未使用（LRU）置换算法:
访问页面 1: 1 - -
访问页面 4: 1 4 -
访问页面 3: 1 4 3
访问页面 1: 1 4 3
访问页面 2: 1 2 3
访问页面 5: 1 2 5
访问页面 1: 1 2 5
访问页面 4: 1 4 5
访问页面 2: 1 4 2
访问页面 1: 1 4 2
访问页面 4: 1 4 2
访问页面 5: 1 4 5
总缺页次数: 8

4.结果分析

以下是实验中定义的初始数据：

页面访问序列（共 12 次访问，索引从 0 开始）：

0:1, 1:4, 2:3, 3:1, 4:2, 5:5, 6:1, 7:4, 8:2, 9:1, 10:4, 11:5

主存块数：3

一、OPT（最佳置换）算法手算过程

规则：淘汰未来最长时间内不再使用的页面。如果多个页面将在未来被访问，选择下次使用最晚的那个页面淘汰；如果页面未来不再使用，则优先淘汰该页面。

- 初始：内存 [- , - , -]

请求序号	请求页	内存状态变更	缺页？	决策过程
0:1		[1,—,—]	缺页	fault=1
1:4		[1,4,—]	缺页	fault=2
2:3		[1,4,3]	缺页	fault=3
3:1	已在内存	[1,4,3]	命中	
4:2	不在内存	[1,4,2]	缺页	比较 [1,4,3]未来使用：未来请求序列:5:5,6:1,7:4,8:2,9:1,10:4,11:5 1 下次用在 64 下次用在 73 未来不用淘汰 3 fault=4
5:5	不在内存	[1,4,5]	缺页	当前[1,4,2],未来:6:1,7:4,8:2,9:1,10:4,11:5 1 下次 6,4 下次 7,2 下次 8，最远是 2 淘汰 2 ，fault ＝5

6:1	已在内存	[1,4,5]	命中	
7:4	已在内存	[1,4,5]	命中	
8:2	不在内存	[1,4,2]	缺页	当前 [1,4,5],未来:9:1,10:4,11:51 下次 9,4 下次 10,5 下次 11，最远是 5 淘汰 5 ，fault ＝6
9:1	已在内存	[1,4,2]	命中	
10:4	已在内存	[1,4,2]	命中	
11:5	不在内存	[5,4,2]	缺页	当前 [1,4,2],未来无请求 1、4、2 均不再使用随便淘汰任意一个（代码中通常按次序检查，第一个无未来使用的为 1 ）淘汰 1 ，fault=7

OPT 缺页次数：7 次

二、FIFO（先进先出）算法手算过程

规则：总是淘汰最先进入内存且尚未被淘汰的页面。

- 初始：内存空为 [- , - , -] ，替换指针 pointer=0

请求序号	请求页	内存状态变更	缺页?	决策过程
0:1		[1，－，－]	缺页	载入 1，fault=1，pointer 指向下一个=1
1:4		[1,4,－]	缺页	载入 4，fault=2，pointer=2
2:3		[1,4,3]	缺页	载入 3，fault=3，pointer ＝0
3:1	已在内存	[1,4,3]	命中	不换，pointer ＝0
4:2	不在内存	[2,4,3]	缺页	FIFO 淘汰 frames[0]=1，换入 2，fault ＝4，pointer=1
5:5	不在内存	[2,5,3]	缺页	淘汰 frames [1]＝4，换入 5 ，fault ＝5，pointer ＝2
6:1	不在内存	[2,5,1]	缺页	淘汰 frames [2]＝3，换入 1，fault=6，pointer=0
7:4	不在内	[4,5,1]	缺页	淘汰 frames[0]=2，换入 4，fault=7，

	存			pointer=1
8:2	不在内存	[4,2,1]	缺页	淘汰 frames[1]=5，换入 2，fault=8，pointer=2
9:1	已在内存	[4,2,1]	命中	不换，pointer=2
10:4	已在内存	[4,2,1]	命中	不换，pointer=2
11:5	不在内存	[4,2,5]	缺页	淘汰 frames[2]=1，换入 5 ，fault=9，pointer=0

FIFO 缺页次数：9 次

三、LRU（最近最久未使用）算法手算过程

规则：淘汰最近最久未使用的页面。记录上次使用时间，每次要换页时选上次使用时间最久远的那页淘汰。

时间	请求页面	内存状态变化	缺页?	说明
1	1	[1，－，－]	缺页	载入 1，lastUsed[0]=1
2	4	[1,4,－]	缺页	载入 4，lastUsed[1]=2
3	3	[1,4,3]	缺页	载入 3，lastUsed[2]=3
4	1	[1,4,3]	命中	1 已存在，更新 1 的 lastUsed 为 4
5	2	[1,2,3]	缺页	不在内存，淘汰 LRU 页面：当前 lastUsed={4(1), 2(4), 3(3)} 最小是 2 ，对应页面 4，于是淘汰 4，载入 2，lastUsed[1]=5
6	5	[1,2,5]	缺页	不在内存，淘汰 LRU 页面：此时 lastUsed={4(1), 5(2), 3(3) → 已更新为 2＝5 了，3＝3} ，最小是 3 对应页面 3 ，替换为 5 ，lastUsed[2]=6
7	1	[1,2,5]	命中	页面 1 存在，更新 lastUsed[0]=7
8	4	[1,4,5]	缺页	不在内存，淘汰 LRU 页面：lastUsed={7(1),5(2),6(5)}中最小为 5 对应页面 2，替换为 4，lastUsed[1]=8

9	2	[1,4,2]	缺页	不在内存，淘汰 LRU 页面： lastUsed ={7(1), 8(4), 6(5)} 最小为 6 对应页面 5 ，替换为 2 ， lastUsed[2]=9
10	1	[1,4,2]	命中	页面 1 存在，更新 lastUsed[0]=10
11	4	[1,4,2]	命中	页面 4 存在，更新 lastUsed[1]=11
12	5	[1,4,5]	缺页	不在内存，淘汰 LRU 页面： lastUsed ={10(1), 11(4), 9(2)} 最小为 9 对应页面 2，替换为 5 ， lastUsed[2]=12

LRU 缺页次数：8 次

最终结果对比：

• **OPT**：缺页次数 =7 次 （最优）

• **FIFO**：缺页次数 =9 次

• **LRU**：缺页次数 =8 次

从计算可见，OPT 算法在该访问序列下缺页最少，这是符合 OPT 算法理论最优性的。LRU 次之，FIFO 缺页最多。

要测试页面置换算法在不同主存块数量（物理块数）下的表现，可以通过调整 FRAME_COUNT 的值，分别设置为 2、3、4，运行程序并观察每种算法的输出结果。以下是具体方法和需要注意的结果对比。

FRAME_COUNT=2

```
最佳置换（OPT）置换算法:
访问页面 1: 1 -
访问页面 4: 1 4
访问页面 3: 1 3
访问页面 1: 1 3
访问页面 2: 1 2
访问页面 5: 1 5
```

访问页面 1: 1 5
访问页面 4: 1 4
访问页面 2: 1 2
访问页面 1: 1 2
访问页面 4: 4 2
访问页面 5: 5 2
总缺页次数: 9

先进先出（FIFO）置换算法:
访问页面 1: 1 -
访问页面 4: 1 4
访问页面 3: 3 4
访问页面 1: 3 1
访问页面 2: 2 1
访问页面 5: 2 5
访问页面 1: 1 5
访问页面 4: 1 4
访问页面 2: 2 4
访问页面 1: 2 1
访问页面 4: 4 1
访问页面 5: 4 5
总缺页次数: 12

最近最久未使用（LRU）置换算法:
访问页面 1: 1 -
访问页面 4: 1 4
访问页面 3: 3 4
访问页面 1: 3 1
访问页面 2: 2 1
访问页面 5: 2 5
访问页面 1: 1 5
访问页面 4: 1 4
访问页面 2: 2 4
访问页面 1: 2 1
访问页面 4: 4 1
访问页面 5: 4 5
总缺页次数: 12

FRAME_COUNT=4

最佳置换（OPT）置换算法:
访问页面 1: 1 - - -
访问页面 4: 1 4 - -
访问页面 3: 1 4 3 -

```
访问页面 1: 1 4 3 -
访问页面 2: 1 4 2 -
访问页面 5: 1 4 2 5
访问页面 1: 1 4 2 5
访问页面 4: 1 4 2 5
访问页面 2: 1 4 2 5
访问页面 1: 1 4 2 5
访问页面 4: 1 4 2 5
访问页面 5: 1 4 2 5
总缺页次数: 5

先进先出（FIFO）置换算法:
访问页面 1: 1 - - -
访问页面 4: 1 4 - -
访问页面 3: 1 4 3 -
访问页面 1: 1 4 3 -
访问页面 2: 1 4 3 2
访问页面 5: 5 4 3 2
访问页面 1: 5 1 3 2
访问页面 4: 5 1 4 2
访问页面 2: 5 1 4 2
访问页面 1: 5 1 4 2
访问页面 4: 5 1 4 2
访问页面 5: 5 1 4 2
总缺页次数: 7

最近最久未使用（LRU）置换算法:
访问页面 1: 1 - - -
访问页面 4: 1 4 - -
访问页面 3: 1 4 3 -
访问页面 1: 1 4 3 -
访问页面 2: 1 4 3 2
访问页面 5: 1 5 3 2
访问页面 1: 1 5 3 2
访问页面 4: 1 5 4 2
访问页面 2: 1 5 4 2
访问页面 1: 1 5 4 2
访问页面 4: 1 5 4 2
访问页面 5: 1 5 4 2
总缺页次数: 6
```

在不同的物理块数量（FRAME_COUNT=2、3、4）下，页面置换算法的表现差异显著，主要体现在缺页次数和内存状态变化趋势上。当

FRAME_COUNT=2 时，内存空间较为紧张，三个算法的缺页次数均较高。最佳置换算法（OPT）的缺页次数为 9，因为它总是选择未来最久不再使用的页面进行淘汰，能够最大程度降低缺页次数；最近最久未使用算法（LRU）和先进先出算法（FIFO）则都需要替换更多的页面，缺页次数分别为 12，体现出这两种算法在内存资源不足时的性能局限。随着 FRAME_COUNT 增加到 3，三个算法的缺页次数均有所下降，其中 OPT 的缺页次数降至 7，LRU 减少到 8，而 FIFO 为 9。这种变化表明，内存块数的增加使得更多的页面可以驻留在内存中，减少了页面被频繁替换的情况，同时 LRU 的时间局部性捕捉能力开始显现，表现出比 FIFO 更优的性能。而当 FRAME_COUNT 达到 4 时，内存资源更加充裕，OPT 的缺页次数进一步减少至 5，几乎接近理想状态，LRU 的缺页次数也降至 6，表现接近 OPT，说明它在实际场景中能充分利用时间局部性，优化页面替换；FIFO 的缺页次数则降至 7，但仍高于其他两种算法，体现出其对局部性规律的不敏感性。总体来看，物理块数的增加有效降低了缺页次数，而算法的性能差异也逐渐显现。OPT 始终是理论上的最优算法，随着内存块数增加，它的性能提升最显著；LRU 在实际中接近最优，尤其在内存较充裕时表现突出；FIFO 则因替换策略简单，无法捕捉局部性，在任何场景下的性能都较为一般，但其实现简便，适合对性能要求较低的应用场景。这些结果清晰地反映了物理块数和页面置换策略对性能的影响规律。

【实验总结】

本次实验通过实现三种经典页面置换算法（FIFO、OPT、LRU）并对比其在给定页面访问序列中的表现，深入理解了页面置换的机制和影响因素。实验重点在于分析各算法的适用场景和性能差异，并解决编程实现过程中遇到的问题。

页面置换算法是虚拟存储系统的核心，负责在内存不足时决定哪一个页面需要被淘汰，以加载新的页面。每种算法都有其独特的决策规则：

1. **OPT 算法**：理论上最优的页面置换算法，通过淘汰未来最久不再被访问的页面，最小化了缺页次数。但 OPT 算法需要完整预知未来的页面访问序列，因此仅作为理论基准存在，实际系统中无法实现。

2. **FIFO 算法**：基于先进先出规则，淘汰最早进入内存的页面。虽然实现简单，但容易出现“Belady 异常”，即内存块数量增加时，缺页次数反而增加。这是因为 FIFO 算法忽略了程序的时间局部性，可能频繁淘汰刚刚需要的页面。

3. **LRU 算法**：基于时间局部性原理，选择最近最久未使用的页面淘汰。它通过记录每个页面最近的访问时间来实现，在大多数情况下能够很好地捕捉程序的局部性。然而，LRU 算法需要额外的数据结构（如时间戳或链表）支持，复杂度高于 FIFO，且在循环访问模式下性能可能不及 OPT。

通过实验，我观察到 OPT 的缺页次数始终是最低的，这符合其理论最优性；LRU 紧随其后，能够有效降低缺页率；而 FIFO 则因为无法感知局部性，在某些情况下表现较差。

实验过程中出现的错误及修改过程

1.OPT 算法问题

OPT 算法需要预测未来的页面访问顺序。在实现过程中，由于未正确处理页面未来不再访问的情况，导致部分页面未被正确淘汰。例如，未考虑页面在整个访问序列中完全未再次出现的场景，程序错误地选择了其他页面。通过添加以下逻辑：

```
if (j == PAGE_REQUEST_COUNT) {
    return i; // 未来不再访问的页面直接淘汰
}
```

解决了该问题，使 OPT 算法能够正确淘汰未来不再访问的页面。

2． FIFO 算法问题

在最初实现 FIFO 算法时，指针循环更新的逻辑出现错误，导致替换指针未正确指向需要淘汰的页面。具体表现为某些情况下淘汰了错误的页面，甚至未按顺序循环。问题的根源在于未对指针进行正确的模运算。通过修正替换逻辑：

```
*pointer = (*pointer + 1) % frameCount;
```

确保指针能够循环更新，问题得以解决。

3．LRU 算法问题

LRU 算法在实现时，最初未能正确更新 lastUsed 数组，导致无法准确记录页面的最近访问时间。例如，页面命中时未更新对应时间，导致后续替换逻辑错误。通过在页面命中逻辑中补充时间更新：

```
if (frames[j] == pageRequests[i]) {
    lastUsed[j] = time; // 更新页面最近使用时间
    break;
}
```

确保每次访问页面时都准确记录其最近使用时间，问题得以解决。

此外，在调试过程中发现，lastUsed 数组未初始化为合理的值，可能导致程序读取无效数据。为此，在初始化时将其全部设置为 0：

```
int lastUsed[FRAME_COUNT] = {0, 0, 0};
```

代码中 PAGE_REQUEST_COUNT 定义为 13，这会导致程序访问 pageRequests 数组越界，从而读到随机值（例如 0），导致出现不存在的页面号。

解决方法：将 PAGE_REQUEST_COUNT 修改为正确的页面请求数量，即 12。

```
#define PAGE_REQUEST_COUNT 12
int pageRequests[PAGE_REQUEST_COUNT] = {1, 4, 3, 1, 2, 5,
```

1，4，2，1，4，5}；

错误的结果：

```
最佳置换（OPT）置换算法：
访问页面 1: 1 - -
访问页面 4: 1 4 -
访问页面 3: 1 4 3
访问页面 1: 1 4 3
访问页面 2: 1 4 2
访问页面 5: 1 4 5
访问页面 1: 1 4 5
访问页面 4: 1 4 5
访问页面 2: 1 4 2
访问页面 1: 1 4 2
访问页面 4: 1 4 2
访问页面 5: 5 4 2
访问页面 0: 0 4 2
总缺页次数：8

先进先出（FIFO）置换算法：
访问页面 1: 1 - -
访问页面 4: 1 4 -
访问页面 3: 1 4 3
访问页面 1: 1 4 3
访问页面 2: 2 4 3
访问页面 5: 2 5 3
访问页面 1: 2 5 1
访问页面 4: 4 5 1
访问页面 2: 4 2 1
访问页面 1: 4 2 1
访问页面 4: 4 2 1
访问页面 5: 4 2 5
访问页面 0: 0 2 5
总缺页次数：10

最近最久未使用（LRU）置换算法：
访问页面 1: 1 - -
访问页面 4: 1 4 -
访问页面 3: 1 4 3
访问页面 1: 1 4 3
访问页面 2: 1 2 3
访问页面 5: 1 2 5
访问页面 1: 1 2 5
访问页面 4: 1 4 5
访问页面 2: 1 4 2
访问页面 1: 1 4 2
访问页面 4: 1 4 2
访问页面 5: 1 4 5
访问页面 0: 0 4 5
总缺页次数：9
```

实验过程中，虽然三种算法的实现逻辑相对简单，但细节问题较多。例如，指针循环、时间记录、未来访问序列的判断等环节都可能出现偏差，导致算法行为与预期不符。通过逐步排查和修正代码，我不仅验证了三种算法的正确性，还加深了对其内在原理的理解。

本次实验展示了不同页面置换算法的优劣对比：OPT 虽然最优，但仅限于理论环境；LRU 在实际应用中较为高效，但实现复杂度较高；FIFO 适合实现简单的场景，但在局部性强的访问序列中可能表现不佳。