虚拟存储与页面置换算法仿真实验指导

一、本节内容导读（知识点）

本节聚焦 **虚拟存储系统中的核心调度问题。**虚拟存储的本质，是在**物理内存有限**的条件下，为进程提供**容量远大于物理内存的地址空间。**这一目标能否实现，关键取决于操作系统如何在运行过程中，**动态决定哪些页面驻留在内存中、哪些页面被换出。**本节将重点理解以下几个核心概念。

---

1. 局部性原理（Locality Principle）

局部性原理是虚拟存储系统得以成立的**理论基础**，它描述了程序访问内存时所呈现出的统计规律，主要包括：

（1）时间局部性

如果某个内存位置在当前时刻被访问，那么在不久的将来它**很可能再次被访问。**

典型例子包括：

- 循环体中的指令和变量；
- 频繁调用的函数；
- 反复使用的全局变量或热点数据。

（2）空间局部性

如果某个内存位置被访问，那么其**相邻地址**在不久的将来被访问的概率也较高。

典型例子包括：

- 顺序执行的指令流；
- 数组、结构体的连续访问；
- 顺序读取文件或缓冲区。

虚拟存储系统正是基于局部性原理的假设设计的： 程序在任意时间内，只会访问其地址空间中的一小部分页面（称为“工作集”），因此无需将全部页面同时装入内存。

---

2. 页面命中与缺页

在分页式虚拟存储系统中，进程访问内存的基本单位是**页面。**

（1）页面命中（Page Hit）

当进程访问某个虚拟地址时：

- 如果该地址所在的页面已经驻留在物理内存中；
- 页表中对应项有效；

则称为**页面命中**，CPU 可以直接完成访问，开销很小。

（2）缺页（Page Fault）

如果访问的页面当前不在内存中：

- 页表项被标记为无效；
- CPU 触发缺页异常；
- 控制权转交给操作系统内核；

内核需要：

1. 找到磁盘上对应的页面；
2. 为其分配物理页框；
3. 将页面调入内存；
4. 更新页表；
5. 恢复进程执行。

缺页的代价极高，其开销通常是一次普通内存访问的**数百到上千倍。** 因此，虚拟存储系统的性能在很大程度上取决于：

> **缺页发生的频率（缺页率）**

页面置换算法的核心目标，就是在有限的物理内存条件下，尽量减少缺页。

---

3. 页面置换算法的设计思想

当发生缺页且内存中已无空闲物理页时，操作系统必须选择一个**已有页面换出**，这一决策由页面置换算法完成。

页面置换算法的设计需要回答一个核心问题：

> **“当前驻留在内存的页面中，哪一个在未来最不可能被再次访问？”**

常见算法的设计思想包括：

（1）利用历史行为预测未来访问

- **FIFO**：认为最早进入内存的页面“最老”，可能最先不再被使用；
- **LRU**：认为最近最少使用的页面，将来再次被访问的概率最低。

这些算法基于一个经验假设：

> 过去的访问行为，在一定程度上可以预测未来。

（2）权衡实现成本与效果

- 精确记录访问历史会带来额外的时间和空间开销；
- 算法越复杂，实现成本越高；
- 实际系统中往往采用“近似算法”而非理论最优。

页面置换算法的本质，是在**预测准确性**与**系统开销**之间寻找平衡。

---

4. 理论最优与工程折中的关系

（1）理论最优算法（OPT）

OPT（Optimal）算法的思想是：

> 每次置换“未来最长时间内不会被访问的页面”

该算法可以保证**缺页次数最少**，因此被用作性能上限的参考。

但 OPT 算法存在根本缺陷：

- 它需要知道**未来的页面访问序列**；
- 在真实系统中无法实现。

因此，OPT 并不是工程方案，而是**理论标杆。**

（2）工程折中算法

实际操作系统采用的算法（如 FIFO、LRU 及其改进形式）：

- 无法做到完全最优；
- 但可以通过局部性原理，在大多数程序中获得接近最优的效果；
- 具有可实现性和可控开销。

工程实践的核心思想是：

> **在可实现的前提下，尽可能接近理论最优**

虚拟存储系统正是通过这种折中设计，在性能、复杂度和稳定性之间取得平衡。

通过理解以上内容，可以形成如下认知链条：

- 局部性原理解释了“为什么虚拟存储可行”；
- 页面命中与缺页描述了虚拟存储的运行状态；
- 页面置换算法决定了缺页发生的频率；
- 理论最优算法提供性能上限；
- 工程折中算法在现实系统中实现可接受的性能。

这正是虚拟存储系统在实际操作系统中能够高效运行的根本原因。

二、实验目标

1. 理解页面访问序列的生成逻辑；
2. 掌握 OPT、FIFO、LRU 三种页面置换算法；
3. 对比不同算法的缺页率；
4. 分析物理块数量对系统性能的影响。

三、实验内容

目标：通过模拟 OPT、FIFO、LRU 三种算法，统计缺页率与命中率，验证虚拟存储技术如何在物理内存有限的情况下，支撑大规模程序的运行。

3.1 核心机制：遵循局部性原理的指令流生成

在真实场景中，程序指令的访问具有“局部性”。实验通过模拟经典的 Workload 模型（50%顺序、25%前跳、25%后跳）来产生 320 条指令对应的页面序列。

```
void generate_address_stream() {
    int pc = rand() % TOTAL_ADDR; // 初始随机位置
    // srand((unsigned)time(NULL));

    // 这里的逻辑遵循经典的 Workload 生成模型：
    // 50% 顺序执行，25% 均匀跳转到前面，25% 均匀跳转到后面

    for (int i = 0; i < TOTAL_ADDR; i++) {
        page_stream[i] = pc / PAGE_SIZE; // 计算页号 = 地址 / 页面大小

        int branch = rand() % 10; // 0~9，用于决定执行流向

        if (branch < 5) {
            // 50% 概率：顺序执行下一条指令
            pc = (pc + 1) % TOTAL_ADDR;
        } else if (branch < 8) {
            // 25% 概率：向前跳转（在 0 到当前地址之间随机跳转）
            pc = rand() % (pc + 1);
        } else {
            // 25% 概率：向后跳转（在当前地址到末尾之间随机跳转）
            pc = (pc + 1 + (rand() % (TOTAL_ADDR - pc)));
            pc %= TOTAL_ADDR; // 确保不越界
        }
    }
}
```

【代码详解】：

- pc / PAGE_SIZE：由于每个页面包含 10 条指令，通过除法将虚拟地址转换为页号，存储在 page_stream 序列中。
- branch < 5：模拟了空间局部性，即程序大部分时间按顺序执行。
- 跳转逻辑：25% 的前跳（模拟循环逻辑）和 25% 的后跳（模拟分支或函数调用）共同构成了一个复杂的页面访问序列，用于测试置换算法的性能。

3.2 理想基准：OPT（最佳置换算法）

原理说明：OPT 算法选择“在未来最长时间内不再被访问”的页面进行置换。这需要预知未来的指令流，因此在现实系统中无法实现，常作为性能评价的上限基准。

```
// 寻找最远才会用到的页进行置换
int max_dist = -1;
int replace_idx = -1;

for (int j = 0; j < FRAME_COUNT; j++) {
    int dist = 99999; // 假设该页不再使用
    for (int k = i + 1; k < TOTAL_ADDR; k++) {
        if (page_stream[k] == memory[j]) {
            dist = k; // 找到下一次使用该页的时间点
            break;
        }
    }
    if (dist > max_dist) {
        max_dist = dist; // 记录最晚使用的页面位置
        replace_idx = j;
    }
}
```

【代码详解】：

- 向后扫描：代码从当前指令位置 i+1 开始向后遍历整个指令流。
- 牺牲决策：max_dist 最大的页面即为“未来最不需要”的页面。将其换出可以保证在当前状态下，下一次缺页发生的时间最晚。

3.3 基础策略：FIFO（先进先出算法）

原理说明：FIFO 算法将内存工作区看作一个队列，总是置换掉最早进入内存的页面。它实现简单，但完全不考虑页面被访问的频率。

```
// 使用循环队列管理物理块
int head = 0; // FIFO 指针，指向最先进入的页

if (empty_idx != -1) {
    memory[empty_idx] = page; // 1. 有空位直接填入
} else {
    int replaced_val = memory[head]; // 2. 牺牲 head 指向的“最老”页面
    memory[head] = page;
    head = (head + 1) % FRAME_COUNT; // 3. 指针循环移动
}
```

【代码详解】：

- head 变量：充当了队列头部的角色。无论页面是否被频繁访问，只要它是最早进入内存的，就会被第一个换出。
- % FRAME_COUNT：利用取模运算实现循环队列逻辑，确保指针始终在 0-3 之间循环，依次置换物理块。

3.4 工业级标准：LRU（最近最久未使用算法）

原理说明：LRU 算法基于“如果一个页面刚被访问过，那么它很快会被再次访问”的假设。它通过记录历史访问记录来模拟 OPT 的效果。

```
// 1. 命中时：更新最近使用时间
if (pos != -1) {
    last_used[pos] = i; // 将当前指令序号 i 作为“时间戳”
    continue;
}

// 2. 置换时：寻找 last_used 最小（即最久未被访问）的页
int min_time = 999999;
int replace_idx = -1;
for (int j = 0; j < FRAME_COUNT; j++) {
    if (last_used[j] < min_time) {
        min_time = last_used[j];
        replace_idx = j;
    }
}
```

【代码详解】：

- last_used[pos] = i：每当页面被访问，就记录下当前的指令序号。序号越大，代表页面越“新鲜”。
- last_used[j] < min_time：在缺页时，遍历四个物理块的时间戳。数值最小的那一个，代表它距离上一次被访问已经过去最久了，因此将其判定为“冷数据”并置换。

四、观察与总结

1. 性能对比：执行程序后，对比控制台输出的“缺页率”。你会发现 OPT < LRU < FIFO。请思考：为什么 LRU 在没有任何预知能力的情况下，性能却能逼近 OPT?
2. 物理块的影响：代码中定义物理块为 4。如果将其改为 8 或 16，缺页率会如何变化?
3. 算法权衡：
    - FIFO 虽然缺页率高，但它的时间复杂度是 $O(1)$，不需要维护时间戳。
    - LRU 缺页率低，但每次访问都要更新时间戳，置换时还要遍历物理块，开销较大。
    - OPT 则是理想的标尺，告诉我们算法优化的天花板在哪里。

五、小结

通过本章从系统观测到代码实践的全方位探索，我们验证了导读中提出的存储管理核心逻辑，实现了对内存高效利用、安全隔离与 I/O 优化的深度理解：

1. 精细化监控与安全感知：通过任务一对 free 及 /proc 的观测，我们不仅掌握了监控工具，更直观看到了页表权限位如何从底层实现安全隔离，防止了应用间的数据篡改。
2. I/O 提速与开发简化：任务三的 mmap 实践证明，内存映射能通过“零拷贝”机制突破 I/O 性能瓶颈。同时，将复杂的磁盘跳转简化为简单的指针移动，印证了虚实映射如何降低开发成本并实现按需加载。
3. 算法仿真与容量突破：任务四通过 OPT、FIFO 和 LRU 算法的对比，完整呈现了虚拟存储对物理内存扩容的支撑逻辑。实验证明，借助局部性原理，系统能在极小的物理内存中运行超大规模程序，正面解决了突破硬件限制的难题。

本章实验让我们深刻理解了操作系统如何通过“虚拟化”这一核心抽象，将有限且复杂的物理内存，转化为高效、安全、灵活的存储体系，为后续研究更高级的内核组件打下了坚实基础。