信息与软件工程学院上机实验报告

(第 3 次)

一、实验名称

请求页式存储管理中常用页面置换算法模拟

二、实验目的及要求

实验目的

1. 了解内存分页管理策略。
2. 掌握调页策略。
3. 掌握一般常用的调度算法。
4. 学会各种存储分配算法的实现方法。
5. 了解页面大小和内存实际容量对命中率的影响。

实验要求

1. 采用页式分配存储方案，通过分别计算不同算法的命中率来比较算法的优劣，同时也考虑页面大小及内存实际容量对命中率的影响；
2. 实现 OPT（最优置换）、LRU（最近最久未使用）、FIFO（先进先出）算法的模拟；
3. 会使用某种编程语言实现并验证结果。

三、实验环境

- 开发语言：C++
- 实验环境：Linux (Ubuntu 24.04 LTS)
- 编译工具：g++ (C++17)

四、实验设计

1. 实验步骤

设计思想

本实验实现一个“页面置换算法模拟器”，支持三种经典算法：

- FIFO（先进先出）：淘汰最早进入内存的页面，使用队列维护进入顺序；
- LRU（最近最久未使用）：淘汰最近最久未被访问的页面，使用链表 + 哈希位置实现精确 LRU；
- OPT（最优置换）：理论最优，淘汰未来最久才会访问（或不再访问）的页面，作为评估基准。

模拟器支持两种输入方式：

- 直接输入“页号序列”；

- 输入“字节地址序列 + 页面大小”，自动换算为“页号序列”。

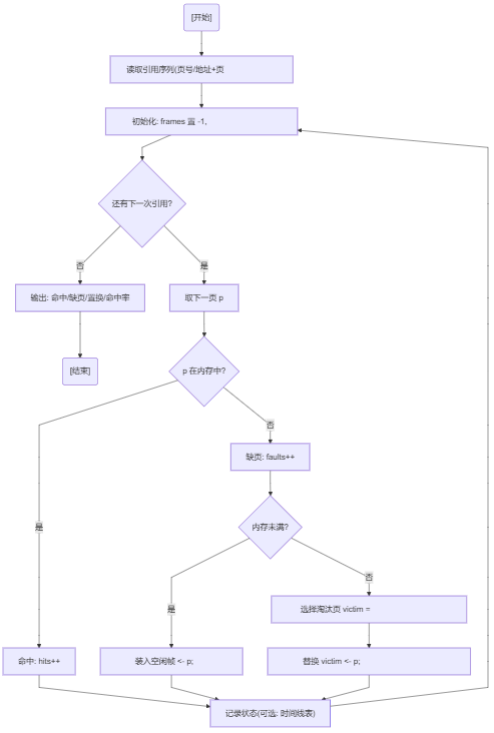
可设置“内存页框数”，输出命中次数、缺页次数、置换次数、命中率，并可选打印“逐步状态表”，便于观察每次访问后的内存帧内容。另提供“参数扫描”，观察页面大小与页框数对命中率的影响。

核心数据结构

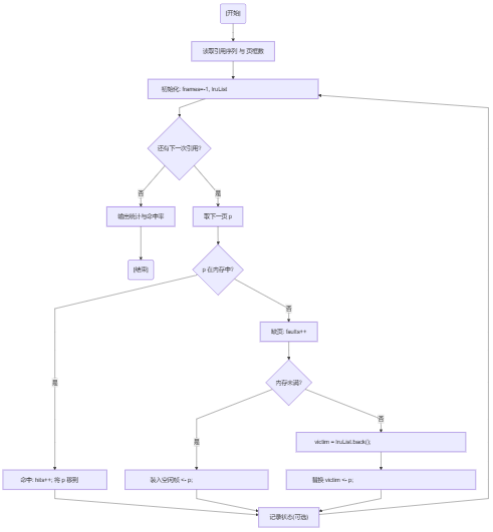
- `Stats`：统计指标
  - `refs`（访问总次数）、`hits`（命中次数）、`faults`（缺页次数）、`replacements`（置换次数）。
- 内存帧表示
  - 使用 `vector<int> frames` 表示当前各帧中的页号（-1 表示空）；
  - 可选的二维表 `frames2D[frame][time]` 用于记录每一步后的帧状态，打印“逐步状态表”。
- 算法辅助结构
  - FIFO：`queue<int> order` + `unordered_set<int> inMem`；
  - LRU：`list<int> lruList` + `unordered_map<int, list<int>::iterator> pos`；
  - OPT：在发生缺页且内存已满时，线性扫描每个候选帧，向后查找其“下一次被访问”的时间，选择最晚者为淘汰页。

算法流程

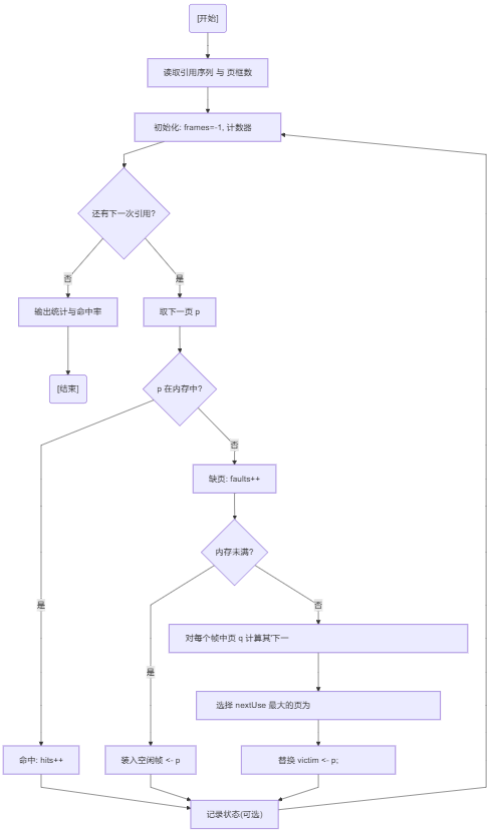
FIFO（先进先出）流程



LRU（最近最久未使用）流程



OPT（最优置换）流程



2. 调试过程与实验结果

测试数据

```
1,0,1,0,2,4,1,0,0,8,7,5,4,3,2,3,4
```

执行结果（页框数 = 3，打印逐步状态表）

```
== FIFO 结果 ==
总访问次数: 17
命中次数   : 5
缺页次数   : 12
置换次数   : 9
命中率     : 0.294118
内存状态表（每列表示一次引用后各帧中的页号，- 表示空）：
+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+
|   1|   0|   1|   0|   2|   4|   1|   0|   0|   8|   7|   5|   4|   3|   2|   3|   4|
+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+
|   1|   1|   1|   1|   1|   4|   4|   4|   4|   8|   8|   8|   4|   4|   4|   4|   4|
|   -|   0|   0|   0|   0|   0|   1|   1|   1|   1|   7|   7|   7|   3|   3|   3|   3|
|   -|   -|   -|   -|   2|   2|   2|   0|   0|   0|   0|   5|   5|   5|   2|   2|   2|
+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+

== LRU 结果 ==
总访问次数: 17
命中次数   : 5
缺页次数   : 12
置换次数   : 9
命中率     : 0.294118
内存状态表（每列表示一次引用后各帧中的页号，- 表示空）：
+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+
|   1|   0|   1|   0|   2|   4|   1|   0|   0|   8|   7|   5|   4|   3|   2|   3|   4|
+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+
|   1|   1|   1|   1|   1|   4|   4|   4|   4|   8|   8|   8|   4|   4|   4|   4|   4|
|   -|   0|   0|   0|   0|   0|   1|   1|   1|   1|   7|   7|   7|   3|   3|   3|   3|
|   -|   -|   -|   -|   2|   2|   2|   0|   0|   0|   0|   5|   5|   5|   2|   2|   2|
+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+

== OPT 结果 ==
总访问次数: 17
命中次数   : 8
缺页次数   : 9
置换次数   : 6
命中率     : 0.470588
内存状态表（每列表示一次引用后各帧中的页号，- 表示空）：
+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+
|   1|   0|   1|   0|   2|   4|   1|   0|   0|   8|   7|   5|   4|   3|   2|   3|   4|
+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+
|   1|   1|   1|   1|   1|   1|   1|   1|   1|   8|   7|   5|   5|   3|   3|   3|   3|
|   -|   0|   0|   0|   0|   0|   0|   0|   0|   0|   0|   0|   0|   2|   2|   2|   2|
|   -|   -|   -|   -|   2|   4|   4|   4|   4|   4|   4|   4|   4|   4|   4|   4|   4|
+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+
```

执行结果（页框数 = 4，使用默认样例）

```
== FIFO 结果 ==
总访问次数: 17
命中次数   : 7
缺页次数   : 10
置换次数   : 6
命中率     : 0.411765
内存状态表（每列表示一次引用后各帧中的页号，- 表示空）：
+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+
|   1|   0|   1|   0|   2|   4|   1|   0|   0|   8|   7|   5|   4|   3|   2|   3|   4|
+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+
|   1|   1|   1|   1|   1|   1|   1|   1|   1|   8|   8|   8|   8|   8|   2|   2|   2|
|   -|   0|   0|   0|   0|   0|   0|   0|   0|   7|   7|   7|   7|   7|   7|   4|
|   -|   -|   -|   -|   2|   2|   2|   2|   2|   2|   5|   5|   5|   5|   5|   5|
|   -|   -|   -|   -|   -|   4|   4|   4|   4|   4|   4|   4|   3|   3|   3|   3|
+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+

== LRU 结果 ==
总访问次数: 17
命中次数   : 7
缺页次数   : 10
置换次数   : 6
命中率     : 0.411765
内存状态表（每列表示一次引用后各帧中的页号，- 表示空）：
+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+
|   1|   0|   1|   0|   2|   4|   1|   0|   0|   8|   7|   5|   4|   3|   2|   3|   4|
+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+
|   1|   1|   1|   1|   1|   1|   1|   1|   1|   1|   1|   5|   5|   5|   5|   5|   5|
|   -|   0|   0|   0|   0|   0|   0|   0|   0|   0|   0|   4|   4|   4|   4|   4|
|   -|   -|   -|   -|   2|   2|   2|   2|   2|   8|   8|   8|   8|   3|   3|   3|   3|
|   -|   -|   -|   -|   -|   4|   4|   4|   4|   4|   7|   7|   7|   7|   2|   2|   2|
+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+

== OPT 结果 ==
总访问次数: 17
命中次数   : 9
缺页次数   : 8
置换次数   : 4
命中率     : 0.529412
内存状态表（每列表示一次引用后各帧中的页号，- 表示空）：
+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+
|   1|   0|   1|   0|   2|   4|   1|   0|   0|   8|   7|   5|   4|   3|   2|   3|   4|
+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+
|   1|   1|   1|   1|   1|   1|   1|   1|   1|   8|   7|   5|   5|   3|   3|   3|   3|
|   -|   0|   0|   0|   0|   0|   0|   0|   0|   0|   0|   0|   0|   0|   0|   0|   0|
|   -|   -|   -|   -|   2|   2|   2|   2|   2|   2|   2|   2|   2|   2|   2|   2|   2|
|   -|   -|   -|   -|   -|   4|   4|   4|   4|   4|   4|   4|   4|   4|   4|   4|   4|
```

3. 结果分析与总结

结果分析

- 在相同序列下，命中率趋势通常为：`OPT ≥ LRU ≥ FIFO`；
- 页框数越多，命中率通常越高（Belady 异常在本样例中未出现）；
- 页面大小越大（对同一地址序列），页号序列更短、更易重复，命中率提升明显，但实际系统需权衡内部碎片与 I/O 放大。

思考题

1. 为什么在实际系统中不直接使用“精确 LRU”，而使用“近似 LRU”？
    - 精确 LRU 需要记录每次访问的全局次序（链表/时间戳/寄存器等），硬件或软件开销大；
    - 实际系统常用访问位（Reference Bit）或 CLOCK 等近似算法，以更低开销取得接近 LRU 的效果。
2. OPT 算法为什么难以实现？
    - 需要知道“未来”的访问情况，运行时不可获知；
    - 仅作为理论最优的评估基准，用于对比其他算法的上限表现。

心得体会

通过本实验，我实现并对比了 FIFO、LRU 与 OPT 三种页面置换算法，直观观察了“页面大小”和“内存页框数”对命中率的影响。实验验证了局部性原理对命中率的显著提升，同时也体会到算法选择与系统开销之间的权衡：

- FIFO 实现与理解最简单，但对局部性把握较弱；
- LRU 在多数实际工作负载上表现良好，但精确实现成本高，通常以近似方案替代；
- OPT 提供理论下限，有助于评估其他算法的潜在改进空间。

在实现过程中，我重点完善了“逐步状态表”和“参数扫描”，帮助从微观与宏观两个层面分析算法行为与命中率趋势。

改进意见

扩展更贴近实际的访问轨迹，并加入冷热混合工作集的简短测试。

---

4. 附录

```
#include <iostream>
#include <vector>
#include <queue>
#include <list>
#include <unordered_map>
#include <unordered_set>
#include <sstream>
#include <string>
#include <algorithm>
#include <cctype>
#include <limits>

using std::cin;
using std::cout;
using std::endl;
using std::list;
using std::queue;
using std::size_t;
using std::string;
using std::unordered_map;
using std::unordered_set;
using std::vector;
```

```cpp
// ------------------------ 辅助数据结构与工具函数 ------------------------

// 统计结果结构体
struct Stats {
    int refs = 0;           // 引用总次数（访问次数）
    int hits = 0;           // 命中次数
    int faults = 0;         // 缺页次数（= 引用次数 - 命中次数）
    int replacements = 0;   // 置换次数（仅当内存满且发生缺页时才会置换）
};

// 打印一条分隔线，用于表格可读性
static void printDivider(int columns) {
    // 每一列形如 | xx ，宽度大概 4-5 个字符
    for (int i = 0; i < columns; ++i) {
        cout << "+----"; // 简化为 4 个横线
    }
    cout << "+" << endl;
}

// 将以空格/逗号分隔的一行整数，解析为 vector<int>
static vector<int> parseIntList(const string &line) {
    vector<int> nums;
    std::stringstream ss;
    // 将逗号替换为空格，便于统一解析
    for (char ch : line) {
        if (ch == ',') ss << ' ';
        else ss << ch;
    }
    int x;
    while (ss >> x) nums.push_back(x);
    return nums;
}

// 将“字节地址序列”按“页面大小”转换成“页号序列”
static vector<int> addressesToPages(const vector<int> &addresses, int pageSize) {
    vector<int> pages;
    pages.reserve(addresses.size());
    for (int addr : addresses) {
        if (addr < 0) addr = 0; // 简单防御：负数地址按 0 处理
        pages.push_back(addr / pageSize);
    }
    return pages;
}

// 打印“逐步状态表”：
// columns: 引用次数
// frames 2D: 每次访问后的内存帧内容，维度 = [frameCount][refs]
// refsList: 原始引用（页号）序列，仅用于首行显示
static void printTimelineTable(const vector<vector<int>> &frames2D,
                               const vector<int> &refsList) {
    int frameCount = (int)frames2D.size();
    int refs = refsList.empty() ? 0 : (int)frames2D[0].size();
    if (frameCount == 0 || refs == 0) return;
```

```
    cout << "内存状态表（每列表示一次引用后各帧中的页号，- 表示空）：" << endl;
    printDivider(refs);

    // 第一行：引用序列
    for (int j = 0; j < refs; ++j) {
        cout << "|";
        // 宽度控制简单化，保证可读
        cout.width(4);
        cout << refsList[j];
    }
    cout << "|" << endl;
    printDivider(refs);

    // 后续行：每一帧在各次访问后的内容
    for (int i = 0; i < frameCount; ++i) {
        for (int j = 0; j < refs; ++j) {
            cout << "|";
            cout.width(4);
            int v = frames2D[i][j];
            if (v < 0) cout << "-"; // -1 用 - 展示
            else cout << v;
        }
        cout << "|" << endl;
    }
    printDivider(refs);
}

// ------------------------------ 算法模拟实现 ------------------------------

// FIFO 先进先出置换算法
static Stats simulateFIFO(const vector<int> &pages, int frameCount,
                          bool recordTimeline,
                          vector<vector<int>> *outFrames2D) {
    Stats st;
    st.refs = (int)pages.size();
    if (frameCount <= 0) return st;

    // 使用队列记录进入顺序；使用集合/字典判断是否在内存中
    queue<int> order;
    unordered_set<int> inMem;

    // 用数组记录所有时刻的内存状态（仅在需要时）
    vector<vector<int>> frames2D;
    if (recordTimeline) {
        frames2D.assign(frameCount, vector<int>(st.refs, -1));
    }

    // 当前内存内容也用一个 vector 维护（便于输出）
    vector<int> frames(frameCount, -1);
    int filled = 0; // 已使用的帧数

    for (int t = 0; t < st.refs; ++t) {
```

```
        int p = pages[t];
        bool hit = (inMem.find(p) != inMem.end());

        if (hit) {
            st.hits++;
        } else {
            st.faults++;
            if (filled < frameCount) {
                // 还有空闲帧，直接装入
                frames[filled] = p;
                order.push(p);
                inMem.insert(p);
                filled++;
                // 没有发生“置换”，因为没有淘汰任何页面
            } else {
                // 需要置换：淘汰最早进入的页面（队列头）
                int victim = order.front();
                order.pop();
                inMem.erase(victim);
                st.replacements++;

                // 在 frames 中找到 victim 的位置并替换
                for (int i = 0; i < frameCount; ++i) {
                    if (frames[i] == victim) {
                        frames[i] = p;
                        break;
                    }
                }
                order.push(p);
                inMem.insert(p);
            }
        }

        if (recordTimeline) {
            // 记录当前时刻的帧内容
            for (int i = 0; i < frameCount; ++i) {
                (*outFrames2D)[i][t] = frames[i];
            }
        }
    }

    return st;
}

// LRU 最近最久未使用置换算法（精确 LRU，借助链表维护最近使用次序）
static Stats simulateLRU(const vector<int> &pages, int frameCount,
                         bool recordTimeline,
                         vector<vector<int>> *outFrames2D) {
    Stats st;
    st.refs = (int)pages.size();
    if (frameCount <= 0) return st;

    // 使用双向链表 + 哈希表：
```

```
//  - 链表头部为最近刚使用的页面；尾部为最久未使用
//  - 哈希表记录每个页面在链表中的位置，便于 O(1) 移动
list<int> lruList;
unordered_map<int, list<int>::iterator> pos;

vector<vector<int>> frames2D;
if (recordTimeline) {
    frames2D.assign(frameCount, vector<int>(st.refs, -1));
}

vector<int> frames(frameCount, -1);
int filled = 0;

auto placeIntoFrame = [&](int page) {
    // 有空位时，将页面装入 frames[filled]
    frames[filled] = page;
    filled++;
};

for (int t = 0; t < st.refs; ++t) {
    int p = pages[t];
    auto it = pos.find(p);
    bool hit = (it != pos.end());
    if (hit) {
        st.hits++;
        // 命中：将该页面移动到链表头（表示最近使用）
        lruList.erase(it->second);
        lruList.push_front(p);
        pos[p] = lruList.begin();
    } else {
        st.faults++;
        if ((int)lruList.size() < frameCount) {
            // 还有空闲帧，直接装入
            lruList.push_front(p);
            pos[p] = lruList.begin();
            placeIntoFrame(p);
        } else {
            // 需要置换：淘汰链表尾部的页面（最近最久未使用）
            int victim = lruList.back();
            lruList.pop_back();
            pos.erase(victim);
            st.replacements++;

            // 在 frames 中找到 victim 的位置并替换为 p
            for (int i = 0; i < frameCount; ++i) {
                if (frames[i] == victim) {
                    frames[i] = p;
                    break;
                }
            }

            // 新页面放入链表头
            lruList.push_front(p);
```

```
                pos[p] = lruList.begin();
            }
        }

        if (recordTimeline) {
            for (int i = 0; i < frameCount; ++i) {
                (*outFrames2D)[i][t] = frames[i];
            }
        }
    }

    return st;
}

// OPT 最优置换算法（理论最优，基于“未来”的使用情况）
static Stats simulateOPT(const vector<int> &pages, int frameCount,
                         bool recordTimeline,
                         vector<vector<int>> *outFrames2D) {
    Stats st;
    st.refs = (int)pages.size();
    if (frameCount <= 0) return st;

    vector<vector<int>> frames2D;
    if (recordTimeline) {
        frames2D.assign(frameCount, vector<int>(st.refs, -1));
    }

    vector<int> frames(frameCount, -1);
    int filled = 0;

    // 判断是否命中：线性查找即可（帧数通常不大，便于理解）
    auto indexOf = [&](int page) -> int {
        for (int i = 0; i < frameCount; ++i) if (frames[i] == page) return i;
        return -1;
    };

    for (int t = 0; t < st.refs; ++t) {
        int p = pages[t];
        int idx = indexOf(p);
        bool hit = (idx != -1);
        if (hit) {
            st.hits++;
        } else {
            st.faults++;
            if (filled < frameCount) {
                frames[filled] = p;
                filled++;
            } else {
                // 选择将来最久才会被访问或不再访问的页面作为淘汰者
                int victimPos = -1;
                int farthestUse = -1; // 越大越远；若未来不再访问，视为“无穷远”

                for (int i = 0; i < frameCount; ++i) {
```

```
                    int pageInFrame = frames[i];
                    int nextUse = -1; // -1 表示未来不再使用
                    for (int k = t + 1; k < st.refs; ++k) {
                        if (pages[k] == pageInFrame) { nextUse = k; break; }
                    }
                    int score = (nextUse == -1) ? (st.refs + 1) : nextUse;
                    if (score > farthestUse) {
                        farthestUse = score;
                        victimPos = i;
                    }
                }
                // 替换 victimPos
                frames[victimPos] = p;
                st.replacements++;
            }
        }

        if (recordTimeline) {
            for (int i = 0; i < frameCount; ++i) {
                (*outFrames2D)[i][t] = frames[i];
            }
        }
    }

    return st;
}

// ------------------------------ 统一运行封装 ------------------------------

enum class Algo { FIFO, LRU, OPT };

static const char* algoName(Algo a) {
    switch (a) {
        case Algo::FIFO: return "FIFO";
        case Algo::LRU:  return "LRU";
        case Algo::OPT:  return "OPT";
    }
    return "";
}

// 运行指定算法，返回统计结果，并可选记录时间线以便打印
static Stats runAlgo(Algo algo, const vector<int> &pages, int frameCount,
                     bool verbose, vector<vector<int>> *frames2D) {
    if (verbose && frames2D) {
        frames2D->assign(frameCount, vector<int>((int)pages.size(), -1));
    }
    switch (algo) {
        case Algo::FIFO:
            return simulateFIFO(pages, frameCount, verbose && frames2D, frames2D);
        case Algo::LRU:
            return simulateLRU(pages, frameCount, verbose && frames2D, frames2D);
        case Algo::OPT:
            return simulateOPT(pages, frameCount, verbose && frames2D, frames2D);
```

```
    }
    return Stats();
}

// 打印某个算法的结果（包含基础统计与可选的状态表）
static void printResult(Algo algo,
                        const Stats &st,
                        const vector<vector<int>> &frames2D,
                        const vector<int> &pages,
                        bool verbose) {
    cout << "\n== " << algoName(algo) << " 结果 ==" << endl;
    cout << "总访问次数: " << st.refs << endl;
    cout << "命中次数  : " << st.hits << endl;
    cout << "缺页次数  : " << st.faults << endl;
    cout << "置换次数  : " << st.replacements << endl;
    double hitRate = (st.refs > 0) ? (double)st.hits / (double)st.refs : 0.0;
    cout.setf(std::ios::fixed);
    cout.precision(6);
    cout << "命中率    : " << hitRate << endl;

    if (verbose) {
        printTimelineTable(frames2D, pages);
    }
}

// ------------------------------ 简易交互菜单 ------------------------------

static void runSingleSimulationWithPageSeq() {
    // 1) 读取页号序列
    cout << "请输入页号序列（以空格或逗号分隔），或回车使用默认样例: \n";
    cout << "默认样例: 1,0,1,0,2,4,1,0,0,8,7,5,4,3,2,3,4\n> ";
    string line;
    // 使用 std::getline 捕获整行输入（兼容空格/逗号）
    std::getline(cin, line);
    vector<int> pages;
    if (line.empty()) {
        pages = {1,0,1,0,2,4,1,0,0,8,7,5,4,3,2,3,4};
    } else {
        pages = parseIntList(line);
    }

    if (pages.empty()) {
        cout << "输入为空，已取消。" << endl;
        return;
    }

    // 2) 读取页框数
    int frames = 0;
    cout << "请输入内存页框数（正整数，建议 3~10）: ";
    cin >> frames;
    if (frames <= 0) {
        cout << "页框数无效，已取消。" << endl;
        // 清理换行
```

```
        cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n');
        return;
    }

    // 3) 是否打印逐步状态
    char v;
    cout << "是否打印逐步状态表？(y/n) ";
    cin >> v;
    bool verbose = (v == 'y' || v == 'Y');
    cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n'); // 吃掉行尾

    // 4) 依次运行三种算法并输出结果
    for (Algo algo : {Algo::FIFO, Algo::LRU, Algo::OPT}) {
        vector<vector<int>> frames2D;
        Stats st = runAlgo(algo, pages, frames, verbose, &frames2D);
        printResult(algo, st, frames2D, pages, verbose);
    }
}

static void runSingleSimulationWithAddressSeq() {
    // 1) 读取地址序列与页面大小
    cout << "请输入字节地址序列（以空格或逗号分隔），或回车使用随机样例：\n> ";
    string line;
    std::getline(cin, line);
    vector<int> addrs;
    if (line.empty()) {
        // 默认生成一个小样例（20 个地址，范围 0~999）
        for (int i = 0; i < 20; ++i) addrs.push_back((i*37 + 123) % 1000);
    } else {
        addrs = parseIntList(line);
    }
    if (addrs.empty()) { cout << "输入为空，已取消。" << endl; return; }

    int pageSize = 0;
    cout << "请输入页面大小（字节，正整数，建议 64/128/256/...）：";
    cin >> pageSize;
    if (pageSize <= 0) {
        cout << "页面大小无效，已取消。" << endl;
        cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n');
        return;
    }

    vector<int> pages = addressesToPages(addrs, pageSize);

    int frames = 0;
    cout << "请输入内存页框数（正整数）：";
    cin >> frames;
    if (frames <= 0) {
        cout << "页框数无效，已取消。" << endl;
        cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n');
        return;
    }
```

```
    char v;
    cout << "是否打印逐步状态表？(y/n) ";
    cin >> v;
    bool verbose = (v == 'y' || v == 'Y');
    cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n');

    for (Algo algo : {Algo::FIFO, Algo::LRU, Algo::OPT}) {
        vector<vector<int>> frames2D;
        Stats st = runAlgo(algo, pages, frames, verbose, &frames2D);
        printResult(algo, st, frames2D, pages, verbose);
    }
}

// 参数扫描：尝试多组“页面大小”和“页框数”，观察命中率影响
static void runParameterSweep() {
    cout << "请输入字节地址序列（以空格或逗号分隔），或回车使用默认随机样例：\n> ";
    string line;
    std::getline(cin, line);
    vector<int> addrs;
    if (line.empty()) {
        // 稍具局部性的一段访问序列（手工构造）：
        // 模拟：先集中访问 0~200，再访问 800~1000，反复几次
        for (int r = 0; r < 3; ++r) {
            for (int i = 0; i < 40; ++i) addrs.push_back((i*13 + 17) % 200);
            for (int i = 0; i < 40; ++i) addrs.push_back(800 + (i*19 + 7) % 180);
        }
    } else {
        addrs = parseIntList(line);
    }
    if (addrs.empty()) { cout << "输入为空，已取消。" << endl; return; }

    cout << "请输入页面大小列表（字节，空格/逗号分隔，回车默认 64 128 256 512）：\n> ";
    string psLine;
    std::getline(cin, psLine);
    vector<int> pageSizes;
    if (psLine.empty()) pageSizes = {64, 128, 256, 512};
    else pageSizes = parseIntList(psLine);
    if (pageSizes.empty()) { cout << "页面大小列表为空，已取消。" << endl; return; }

    cout << "请输入页框数列表（空格/逗号分隔，回车默认 3 4 5 6 8）：\n> ";
    string frLine;
    std::getline(cin, frLine);
    vector<int> frameCounts;
    if (frLine.empty()) frameCounts = {3, 4, 5, 6, 8};
    else frameCounts = parseIntList(frLine);
    if (frameCounts.empty()) { cout << "页框数列表为空，已取消。" << endl; return; }

    cout.setf(std::ios::fixed); cout.precision(4);
    for (int ps : pageSizes) {
        vector<int> pages = addressesToPages(addrs, ps);
        cout << "\n页面大小 = " << ps << " 字节" << endl;
        for (int fc : frameCounts) {
            cout << "  页框数 = " << fc << endl;
```

```
            for (Algo algo : {Algo::FIFO, Algo::LRU, Algo::OPT}) {
                vector<vector<int>> dummy; // 扫描模式不打印详细过程
                Stats st = runAlgo(algo, pages, fc, false, &dummy);
                double hitRate = (st.refs > 0) ? (double)st.hits / st.refs : 0.0;
                cout << "    [" << algoName(algo) << "] 命中率=" << hitRate
                     << ", 缺页=" << st.faults << ", 置换=" << st.replacements << endl;
            }
        }
    }
}

static void printMenu() {
    cout << "\n========== 页面置换算法模拟 ==========" << endl;
    cout << "1) 使用页号序列进行单次模拟" << endl;
    cout << "2) 使用字节地址 + 页面大小进行单次模拟" << endl;
    cout << "3) 进行参数扫描（页面大小/页框数对命中率影响）" << endl;
    cout << "0) 退出" << endl;
    cout << "请选择: ";
}

int main() {
    // 提示信息
    cout << "实验三：请求页式存储管理中常用页面置换算法模拟" << endl;
    cout << "已实现算法：FIFO / LRU / OPT" << endl;
    cout << "说明：可选择输入页号序列，或输入字节地址 + 页面大小。" << endl;

    while (true) {
        printMenu();
        int op = -1;
        if (!(cin >> op)) return 0; // 输入非法则退出
        cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n'); // 丢弃行尾
        if (op == 0) break;
        if (op == 1) runSingleSimulationWithPageSeq();
        else if (op == 2) runSingleSimulationWithAddressSeq();
        else if (op == 3) runParameterSweep();
        else cout << "无效的选项，请重新输入。" << endl;
    }

    cout << "已退出。" << endl;
    return 0;
}
```