****

《操作系统》课程设计报告书

**题目：基于Linux的页面替换算法设计**

**学 院 计算机科学与工程学院**

**专 业 计算机科学与技术2班**

**学生姓名 黄鸿展**

**学生学号 202230441138**

**指导教师 杨楷翔**

**课程编号**

**课程学分**

**起始日期 2024年12月20日**

|  |  |
| --- | --- |
| 教  师  评  语 | 教师签名：  日期： |
| 成  绩  评  定 |  |
| 备  注 |  |

**基于Linux的页面替换算法设计**

**一、选题背景**

**课题任务：**基于Linux操作系统环境，设计页面替换算法，实现OPT、FIFO、LRU、Clock、改进Clock算法。理解各页面替换算法的原理及实现方法。

**课题要求：**

（1）编程实现OPT算法、FIFO算法、LRU算法、简单时钟算法和改进时钟算法;

（2）编程实现页面访问序列的随机化机制，包括设置每个页面的读写访问方式，以满足改进时钟算法的要求。

（3）在进程执行和访问每一页的过程中，每一次对一页的访问都应显示输出当时进程页表的内容(包括页号、物理块号、状态位、读写访问方法等字段)和当前页访问操作(如该页已经在内存中或触发缺页中断);

（4）所有算法都应该基于相同的条件，包括:

①系统采用“固定分配，局部替换”策略;

②进程逻辑地址空间的大小相同;

③分配给进程的物理块数目相同。

④相同页面访问序列(整数序列，整数区间[0,N));

（5）进行多次测试，统计分析和比较算法的性能(如缺页率，缺页次数)

**实验环境：**阿里云ECS服务器(2核2GiB)

**操作系统：**Alibaba Cloud Linux 3.2104 LTS 64位

**编程语言：**C++（17标准）

**二、方案论证(设计理念)**

本设计的核心在于实现和比较不同的页面替换算法：OPT（最佳替换算法）、FIFO（先进先出算法）、LRU（最近最少算法）、Clock（简单时钟算法）和改进Clock算法。这些算法在操作系统用于管理内存中的页面替换，以优化内存使用效率和减少缺页率。

**1.OPT算法：**最佳替换算法（Optimal Page Replacement Algorithm）会在发生缺页时选择将来不会被使用或者最迟被使用的页面进行替换。

该算法实现的流程如下：

**①初始化帧数组**：创建一个大小等于物理块数量的数组，用于跟踪当前哪些页面被加载到物理内存中。

**②遍历页面访问序列**：逐一处理页面访问序列中的每个页面访问请求。

**③检查页面是否已在内存中**：检查当前请求的页面是否已存在物理内存中。。

**④页面置换**：

如果页面不在内存中（缺页），则需要进行页面置换。

寻找将来最长时间不会被访问的页面，这涉及到查看未来的页面访问序列，并找到那个在未来最迟被访问的页面。

将找到的页面替换为当前需要加载的页面。

**⑤重复页面检查和置换**：继续处理页面访问序列中的下一个页面，直到所有页面都被处理。

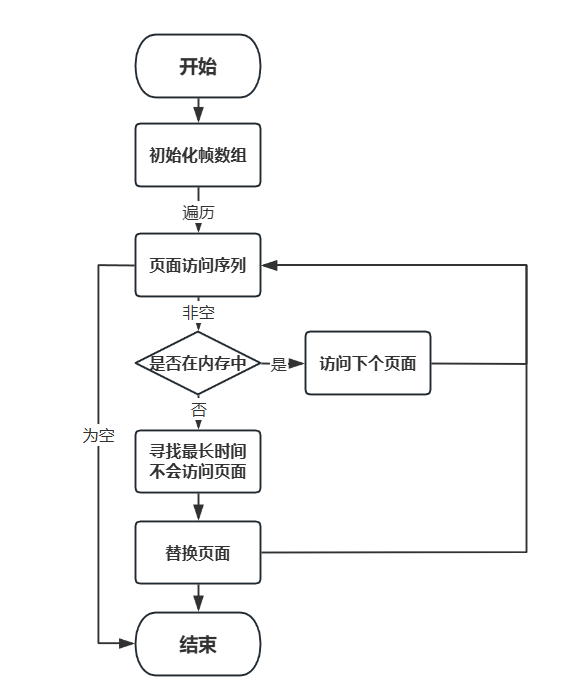


图1 OPT算法流程图

**2.FIFO算法：**先进先出算法（First-In, First-Out Page Replacement Algorithm）是最简单的页面替换算法，它基于页面到达内存的顺序来替换页面。最早进入内存的页面将是第一个被替换的页面。

该算法实现流程如下：

**①初始化帧数组和队列**：创建一个帧数组来存储当前的页面，以及一个队列来记录页面进入内存的顺序。

**②遍历页面访问序列**：处理每一个页面访问请求。

**③检查页面是否已在内存中**：检查当前请求的页面是否已存在物理内存中。

**④页面置换**：

如果页面不在内存中，且内存未满，将页面添加到内存，更新队列。

如果内存已满，将队列头部页面（最早进入页面）换为当前页面，并更新队列。

**⑤重复页面检查和置换**：继续处理下一个页面访问请求。

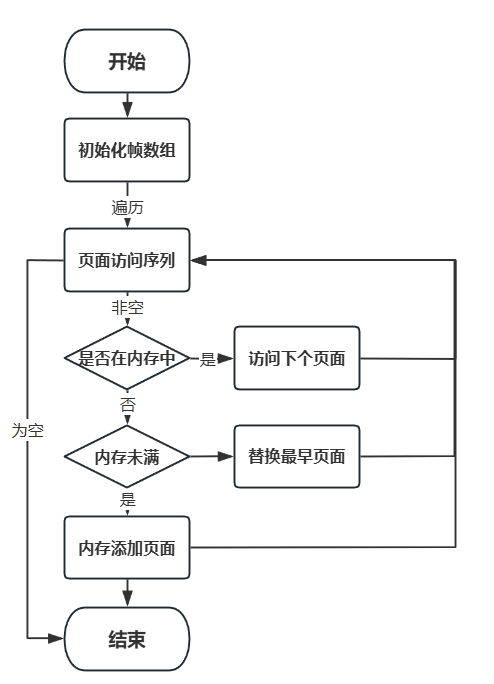


图2 FIFO算法流程图

**③LRU算法：**最近最少使用算法（LeastRecently Used Page Replacement Algorithm）会替换最长时间未被访问的页面。LRU算法通过维护一个页面访问历史列表来实现。

该算法的实现流程如下：

**①初始化帧数组和LRU列表**：创建一个帧数组来存储当前的页面，以及一个列表来记录页面的访问顺序，以便识别最近最少使用的页面。

**②遍历页面访问序列**：逐一处理页面访问序列中的每个页面访问请求。

**③检查页面是否已在内存中**：检查当前请求的页面是否已存在物理内存中。

**④页面置换**：

如果页面不在内存中，且内存未满，将页面添加到内存中，并更新LRU列表。

如果内存已满，根据LRU列表找到最久未被访问的页面，并将其替换为当前页面，同时更新LRU列表。

**⑤重复页面检查和置换**：继续处理下一个页面访问请求。

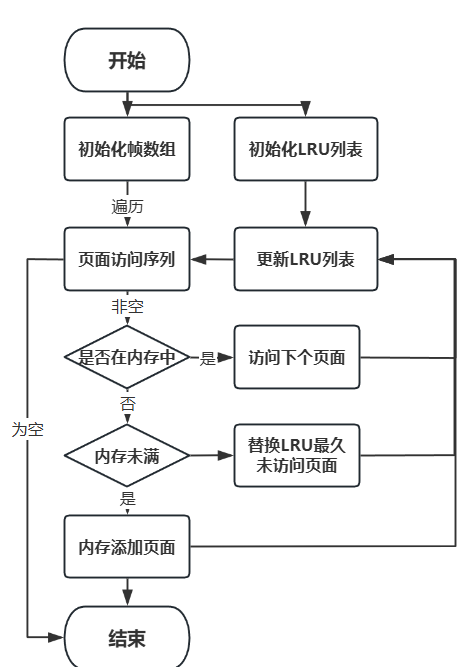


图3 LRU算法流程图

**④Clock算法：**时钟算法（Clock Page Replacement Algorithm）是对LRU算法的一种近似实现，它通过一个循环列表来模拟页面的使用情况，并通过一个指针（时钟手）来选择下一个替换的页面。

该算法实现流程如下：

**①初始化帧数组和时钟指针**：创建一个帧数组来存储当前的页面，并初始化一个指针来模拟时钟算法的循环行为。

**②遍历页面访问序列**：处理每一个页面访问请求。

**③检查页面是否已在内存中**：检查当前请求的页面是否已存在物理内存中。**④页面置换**：

如果页面不在内存中，且内存未满，将页面添加到内存中。

如果内存已满，移动时钟指针，找到下一个标记为未引用的页面进行替换，如果指针绕一圈回到起点仍未找到，则替换指针当前指向的页面，并继续移动指针。

**⑤重复页面检查和置换**：继续处理下一个页面访问请求。

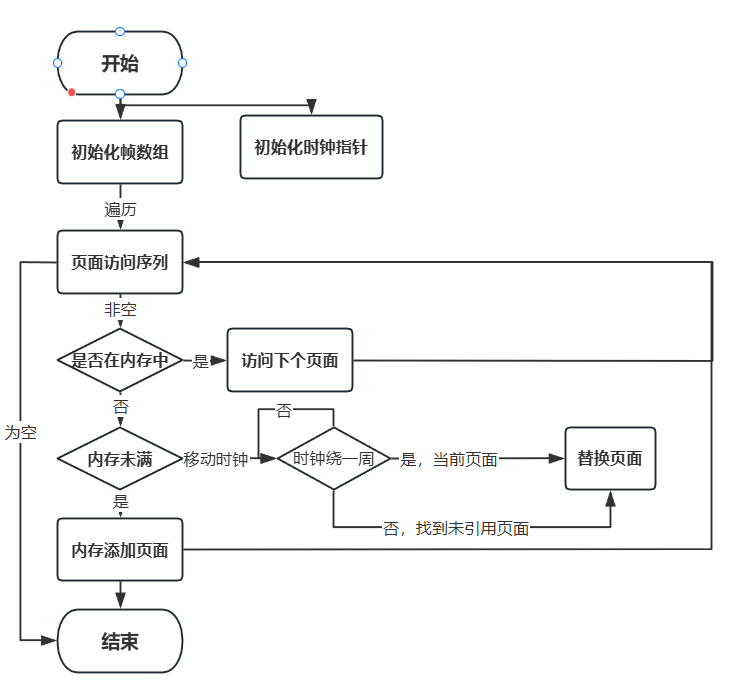


图4 Clock算法流程图

**⑤改进Clock算法**：改进的时钟算法在Clock算法的基础上增加了对修改位的考虑，优先替换那些未被修改的页面，如果所有页面都被修改，则选择最老页面替换。

该算法实现流程如下：

**①初始化帧数组和时钟指针**：创建帧数组和初始化时钟指针。

**②遍历页面访问序列**：逐一处理页面访问序列中的每个页面访问请求。

**③检查页面是否已在内存中**：对于每个页面访问，检查该页面是否已经存在于物理内存中。

**④页面置换**：

如果页面不在内存中，且内存未满，将页面添加到内存中。

如果内存已满，移动时钟指针，优先替换未被修改且未被引用的页面，如果所有页面都被修改过，则按照Clock算法的方式进行替换。

**⑤重复页面检查和置换**：继续处理下一个页面访问请求。

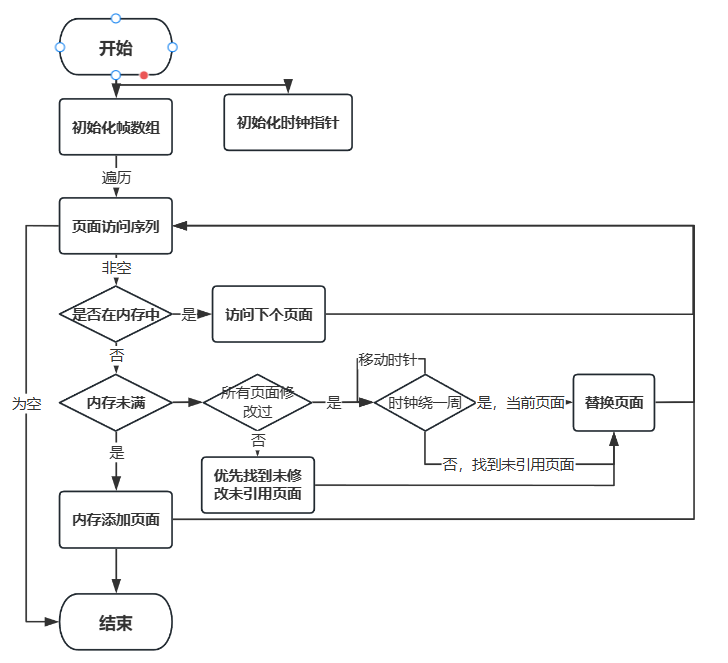


图5 改进Clock算法流程图

**三、过程论述**

根据上述提到的算法流程，用代码实现如下：

首先，拟定程序的主要步骤：

①初始化页表

首先，我们初始化页表，为每个页面分配一个唯一的页号，并设置物理块号、有效位、引用位和修改位。这里我们定义页面10页，物理块数量为3.

|  |
| --- |
| *// 页表项结构体* struct PageTableEntry {     int pageNumber;   *// 页号*     int frameNumber;  *// 物理块号*     bool valid;       *// 有效位（是否在内存中）*     bool referenced;  *// 引用位（是否被访问过）*     bool modified;    *// 修改位（是否被修改过）* };  vector<pair<int, char>> pageAccessSequence; *// 页面访问序列（页号, 访问方式）* vector<PageTableEntry> pageTable(N);        *// 页表* vector<int> frames(M, -1);                  *// 物理块，初始为空* *// 初始化页表* void initializePageTable() {     for (int i = 0; i < N; i++) {         pageTable[i].pageNumber = i;         pageTable[i].frameNumber = -1;         pageTable[i].valid = false;         pageTable[i].referenced = false;         pageTable[i].modified = false;     } }  const int N = 10; // 页面数量  const int M = 3;  // 物理块数量 |

②生成页面访问序列

随机生成页面访问序列，模拟进程对页面的访问请求。

|  |
| --- |
| *// 生成页面访问序列* void generatePageAccessSequence() {     srand(time(0)); *// 设置随机种子*     for (int i = 0; i < 20; i++) {         int page = rand() % N; *// 随机生成页面号*         char accessType = (rand() % 2 == 0) ? 'R' : 'W'; *// 随机生成访问方式（读/写）*         pageAccessSequence.emplace\_back(page, accessType);     } } |

③实现页面替换算法

对于每种算法，我们实现了以下关键步骤：

OPT算法：遍历页面访问序列，对于每次访问，检查页面是否已在物理块中。如果不在，则选择未来最迟被访问的页面进行替换。

|  |
| --- |
| *// OPT算法（最佳替换算法）* int optAlgorithm() {     cout << "OPT算法：" << endl;     int pageFaults = 0; *// 缺页次数*     for (int i = 0; i < pageAccessSequence.size(); i++) {         int page = pageAccessSequence[i].first; *// 当前访问的页面*         char accessType = pageAccessSequence[i].second; *// 访问方式*         cout << "访问页面 " << page << " (" << accessType << ")" << endl;         printPageTable();         printFrames();         if (findPageInFrames(page) != -1) {             cout << "页面 " << page << " 已在内存中。" << endl;             pageTable[page].referenced = true; *// 更新引用位*             if (accessType == 'W') {                 pageTable[page].modified = true; *// 如果是写操作，更新修改位*             }         } else {             pageFaults++;             if (frames[M - 1] == -1) { *// 如果还有空闲物理块*                 for (int j = 0; j < M; j++) {                     if (frames[j] == -1) {                         frames[j] = page;                         pageTable[page].frameNumber = j;                         pageTable[page].valid = true;                         pageTable[page].referenced = true;                         if (accessType == 'W') {                             pageTable[page].modified = true;                         }                         break;                     }                 }             } else { *// 没有空闲物理块，需要替换*                 vector<int> futureAccess(M, pageAccessSequence.size()); *// 记录每个页面未来被访问的时间*                 for (int j = 0; j < M; j++) {                     for (int k = i + 1; k < pageAccessSequence.size(); k++) {                         if (pageAccessSequence[k].first == frames[j]) {                             futureAccess[j] = k;                             break;                         }                     }                 }                 int replaceIndex = 0;                 for (int j = 1; j < M; j++) {                     if (futureAccess[j] > futureAccess[replaceIndex]) {                         replaceIndex = j;                     }                 }                 int replacedPage = frames[replaceIndex];                 frames[replaceIndex] = page;                 pageTable[replacedPage].frameNumber = -1;                 pageTable[replacedPage].valid = false;                 pageTable[page].frameNumber = replaceIndex;                 pageTable[page].valid = true;                 pageTable[page].referenced = true;                 if (accessType == 'W') {                     pageTable[page].modified = true;                 }             }             cout << "缺页：页面 " << page << " 被加载到物理块 " << pageTable[page].frameNumber << endl;         }     }     cout << "总缺页次数: " << pageFaults << endl;     return pageFaults; } |

FIFO算法：使用队列来管理物理块的顺序，当缺页时，替换队列最早进入的页面。

|  |
| --- |
| *// FIFO算法（先进先出算法）* int fifoAlgorithm() {     cout << "FIFO算法：" << endl;     int pageFaults = 0;     queue<int> fifoQueue; *// FIFO队列*     for (int i = 0; i < pageAccessSequence.size(); i++) {         int page = pageAccessSequence[i].first;         char accessType = pageAccessSequence[i].second;         cout << "访问页面 " << page << " (" << accessType << ")" << endl;         printPageTable();         printFrames();          if (findPageInFrames(page) != -1) {             cout << "页面 " << page << " 已在内存中。" << endl;             pageTable[page].referenced = true;             if (accessType == 'W') {                 pageTable[page].modified = true;             }         } else {             pageFaults++;             if (frames[M - 1] == -1) { *// 如果还有空闲物理块*                 for (int j = 0; j < M; j++) {                     if (frames[j] == -1) {                         frames[j] = page;                         pageTable[page].frameNumber = j;                         pageTable[page].valid = true;                         pageTable[page].referenced = true;                         if (accessType == 'W') {                             pageTable[page].modified = true;                         }                         fifoQueue.push(j); *// 将物理块索引加入FIFO队列*                         break;                     }                 }             } else { *// 没有空闲物理块，需要替换*                 if (fifoQueue.empty()) {                     cerr << "错误：FIFO队列为空，无法进行页面替换。" << endl;                     exit(1);                 }                 int replaceIndex = fifoQueue.front();                 fifoQueue.pop();                 int replacedPage = frames[replaceIndex];                 frames[replaceIndex] = page;                 pageTable[replacedPage].frameNumber = -1;                 pageTable[replacedPage].valid = false;                 pageTable[page].frameNumber = replaceIndex;                 pageTable[page].valid = true;                 pageTable[page].referenced = true;                 if (accessType == 'W') {                     pageTable[page].modified = true;                 }                 fifoQueue.push(replaceIndex); *// 将替换后的物理块索引重新加入FIFO队列*             }             cout << "缺页：页面 " << page << " 被加载到物理块 " << pageTable[page].frameNumber << endl;         }     }     cout << "总缺页次数: " << pageFaults << endl;     return pageFaults; } |

LRU算法：维护最近最少使用的页面列表，当缺页时，替换列表最后被访问的页面。

|  |
| --- |
| *// LRU算法（最近最少使用算法）* int lruAlgorithm() {     cout << "LRU算法：" << endl;     int pageFaults = 0;     vector<int> lruList; *// LRU列表*     for (int i = 0; i < pageAccessSequence.size(); i++) {         int page = pageAccessSequence[i].first;         char accessType = pageAccessSequence[i].second;         cout << "访问页面 " << page << " (" << accessType << ")" << endl;         printPageTable();         printFrames();          if (findPageInFrames(page) != -1) {             cout << "页面 " << page << " 已在内存中。" << endl;             pageTable[page].referenced = true;             if (accessType == 'W') {                 pageTable[page].modified = true;             }             *// 更新LRU列表，将访问的页面移到最前面*             lruList.erase(remove(lruList.begin(), lruList.end(), page), lruList.end());             lruList.insert(lruList.begin(), page);         } else {             pageFaults++;             if (frames[M - 1] == -1) { *// 如果还有空闲物理块*                 for (int j = 0; j < M; j++) {                     if (frames[j] == -1) {                         frames[j] = page;                         pageTable[page].frameNumber = j;                         pageTable[page].valid = true;                         pageTable[page].referenced = true;                         if (accessType == 'W') {                             pageTable[page].modified = true;                         }                         lruList.insert(lruList.begin(), page); *// 将新页面插入LRU列表*                         break;                     }                 }             } else { *// 没有空闲物理块，需要替换*                 int replacePage = lruList.back(); *// 选择LRU列表中最后一个页面（最近最少使用的页面）*                 lruList.pop\_back(); *// 从LRU列表中移除*                 int replaceIndex = pageTable[replacePage].frameNumber;                 frames[replaceIndex] = page;                 pageTable[replacePage].frameNumber = -1;                 pageTable[replacePage].valid = false;                 pageTable[page].frameNumber = replaceIndex;                 pageTable[page].valid = true;                 pageTable[page].referenced = true;                 if (accessType == 'W') {                     pageTable[page].modified = true;                 }                 lruList.insert(lruList.begin(), page); *// 将新页面插入LRU列表*             }             cout << "缺页：页面 " << page << " 被加载到物理块 " << pageTable[page].frameNumber << endl;         }     }     cout << "总缺页次数: " << pageFaults << endl;     return pageFaults; } |

Clock算法：使用一个循环列表模拟时钟算法，通过时钟指针选择下一替换页面。

|  |
| --- |
| *// Clock算法（简单时钟算法）* int clockAlgorithm() {     cout << "Clock算法：" << endl;     int pageFaults = 0;     int clockHand = 0; *// 时钟指针*     for (int i = 0; i < pageAccessSequence.size(); i++) {         int page = pageAccessSequence[i].first;         char accessType = pageAccessSequence[i].second;         cout << "访问页面 " << page << " (" << accessType << ")" << endl;         printPageTable();         printFrames();          if (findPageInFrames(page) != -1) {             cout << "页面 " << page << " 已在内存中。" << endl;             pageTable[page].referenced = true; *// 更新引用位*             if (accessType == 'W') {                 pageTable[page].modified = true; *// 如果是写操作，更新修改位*             }         } else {             pageFaults++;             while (true) {                 if (!pageTable[frames[clockHand]].referenced) { *// 找到未被引用的页面*                     int replacedPage = frames[clockHand];                     frames[clockHand] = page;                     pageTable[replacedPage].frameNumber = -1;                     pageTable[replacedPage].valid = false;                     pageTable[page].frameNumber = clockHand;                     pageTable[page].valid = true;                     pageTable[page].referenced = true;                     if (accessType == 'W') {                         pageTable[page].modified = true;                     }                     clockHand = (clockHand + 1) % M; *// 更新时钟指针*                     break;                 } else {                     pageTable[frames[clockHand]].referenced = false; *// 清除引用位*                     clockHand = (clockHand + 1) % M;                 }             }             cout << "缺页：页面 " << page << " 被加载到物理块 " << pageTable[page].frameNumber << endl;         }     }     cout << "总缺页次数: " << pageFaults << endl;     return pageFaults; } |

改进Clock算法：在Clock算法的基础上，增加了对页面修改位的检查，优先替换未被修改的页面。

|  |
| --- |
| *// 改进Clock算法* int improvedClockAlgorithm() {     cout << "改进Clock算法：" << endl;     int pageFaults = 0;     int clockHand = 0;     for (int i = 0; i < pageAccessSequence.size(); i++) {         int page = pageAccessSequence[i].first;         char accessType = pageAccessSequence[i].second;         cout << "访问页面 " << page << " (" << accessType << ")" << endl;         printPageTable();         printFrames();          if (findPageInFrames(page) != -1) {             cout << "页面 " << page << " 已在内存中。" << endl;             pageTable[page].referenced = true; *// 更新引用位*             if (accessType == 'W') {                 pageTable[page].modified = true; *// 如果是写操作，更新修改位*             }         } else {             pageFaults++;             bool replaced = false;             while (!replaced) {                 if (!pageTable[frames[clockHand]].referenced && !pageTable[frames[clockHand]].modified) { *// 找到未引用且未修改的页面*                     int replacedPage = frames[clockHand];                     frames[clockHand] = page;                     pageTable[replacedPage].frameNumber = -1;                     pageTable[replacedPage].valid = false;                     pageTable[page].frameNumber = clockHand;                     pageTable[page].valid = true;                     pageTable[page].referenced = true;                     if (accessType == 'W') {                         pageTable[page].modified = true;                     }                     clockHand = (clockHand + 1) % M;                     replaced = true;                 } else if (!pageTable[frames[clockHand]].referenced && pageTable[frames[clockHand]].modified) { *// 找到未引用但已修改的页面*                     pageTable[frames[clockHand]].modified = false; *// 清除修改位*                     clockHand = (clockHand + 1) % M;                 } else if (pageTable[frames[clockHand]].referenced) { *// 找到已引用的页面*                     pageTable[frames[clockHand]].referenced = false; *// 清除引用位*                     clockHand = (clockHand + 1) % M;                 }             }             cout << "缺页：页面 " << page << " 被加载到物理块 " << pageTable[page].frameNumber << endl;         }     }     cout << "总缺页次数: " << pageFaults << endl;     return pageFaults; } |

④打印页表和物理块状态

在每次页面访问后，打印当前的页表和物理块状态，以便于跟踪算法的行为。

|  |
| --- |
| *// 打印页表* void printPageTable() {     cout << "页表内容：" << endl;     cout << "页号\t物理块号\t有效位\t引用位\t修改位" << endl;     for (int i = 0; i < N; i++) {         cout << pageTable[i].pageNumber << "\t\t" << pageTable[i].frameNumber << "\t\t"              << pageTable[i].valid << "\t" << pageTable[i].referenced << "\t\t"              << pageTable[i].modified << endl;     } } *// 打印物理块状态* void printFrames() {     cout << "物理块状态: ";     for (int i = 0; i < M; i++) {         if (frames[i] != -1) {             cout << frames[i] << " ";         } else {             cout << "- ";         }     }     cout << endl; } |

主程序如下：

|  |
| --- |
| int main() {     initializePageTable(); *// 初始化页表*     generatePageAccessSequence(); *// 生成页面访问序列*     int fifoFaults = fifoAlgorithm(); *// 运行FIFO算法*     cout << endl;     initializePageTable();     int lruFaults = lruAlgorithm(); *// 运行LRU算法*     cout << endl;     initializePageTable();     int optFaults = optAlgorithm(); *// 运行OPT算法*     cout << endl;     initializePageTable();     int clockFaults = clockAlgorithm(); *// 运行Clock算法*     cout << endl;     initializePageTable();     int improvedClockFaults = improvedClockAlgorithm(); *// 运行改进Clock算法*     *// 比较各算法的缺页率*     cout << "各算法的缺页率比较：" << endl;     cout << "FIFO算法缺页率: " << (double)fifoFaults / pageAccessSequence.size() <<"缺页次数："<< fifoFaults << endl;     cout << "LRU算法缺页率: " << (double)lruFaults / pageAccessSequence.size() <<"缺页次数"<< lruFaults << endl;     cout << "OPT算法缺页率: " << (double)optFaults / pageAccessSequence.size() <<"缺页次数"<< optFaults << endl;     cout << "Clock算法缺页率: " << (double)clockFaults / pageAccessSequence.size() <<"缺页次数"<<clockFaults<< endl;     cout << "改进Clock算法缺页率: " << (double)improvedClockFaults / pageAccessSequence.size() <<"缺页次数"<< improvedClockFaults << endl;     return 0; } |

**四、结果分析**

**实验环境搭建：**首先在阿里云购置云服务器，选用Linux系统：

登录阿里云，进行学生认证：

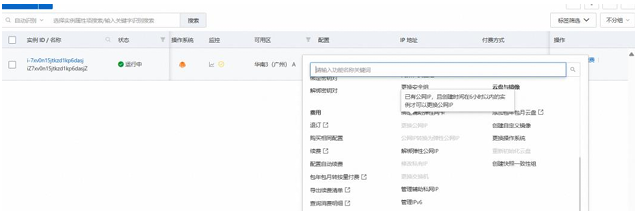
[https://myaccount.console.aliyun.com/basic-info?open=student](https://myaccount.console.aliyun.com/basic-info?open=student%20)

完成学生认证后可以在此处领取代金券购买服务器：

<https://university.aliyun.com/>

操作系统选用的是Alibaba Cloud Linux 3.2104 LTS 64位。

购买以后还需购买弹性公网IP



在Linux系统中设置好账户密码，准备C++开发环境搭建：

在命令行中输入并运行，安装g++编译工具：

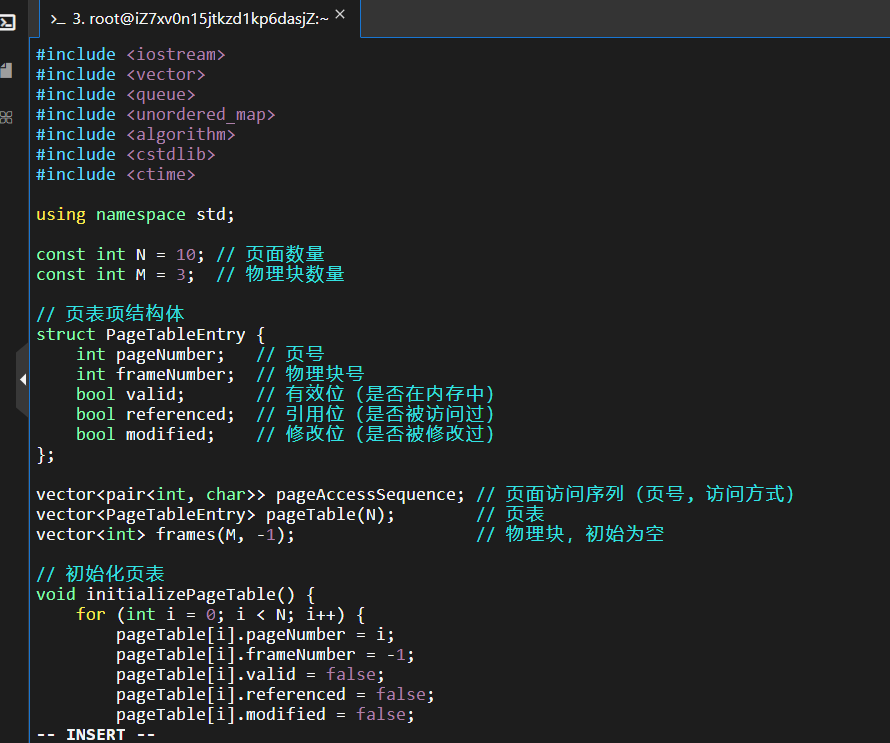
**yum install gcc-c++ libstdc++-devel**

**程序的创建与运行:**

准备好g++后，可以直接在目录中创建文件：

**vim Hello.cpp**

按i进入插入模式，将写好的代码复制到文件中，按Esc，输入 :wq 保存并退出；



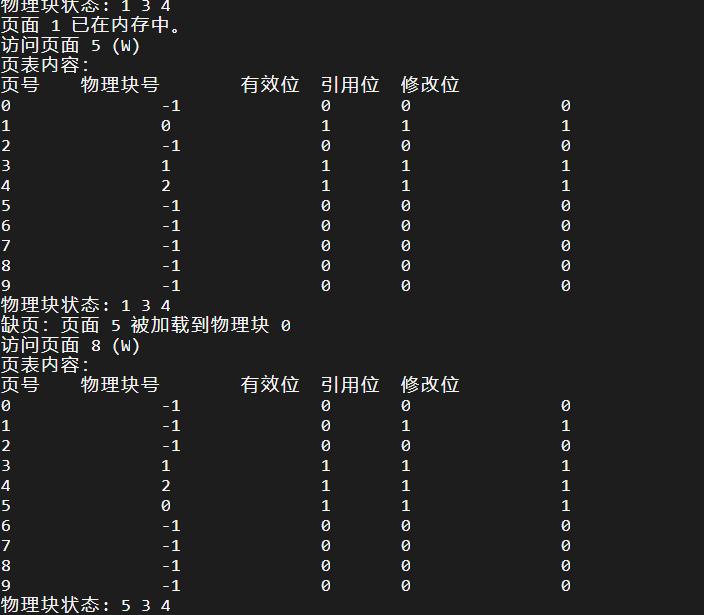
输入以下命令开始编译，生成可执行文件：

**g++ -o Hello Hello.cpp**

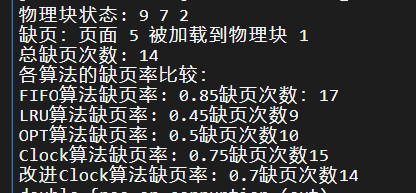
**使用ls即可看到生成的可执行文件：**

****

**输入 ./Hello ,(Hello为程序名)，即可得到程序运行输出：**

****

**中间记录了每个算法在访问页面以后的页面状态，包含物理块号，有效位，引用位，修改位等信息：其中物理块号为-1表示页面未加载入内存中，有效位表示页面是否在内存中（在表示1，否则为0），引用位表示页面是否被访问（被访问为1，否则为0），修改位表示页面是否被修改过（修改过为1，否则为0）；最后物理块状态显示了读入内存的页面，若为空则用-显示。**



最后，程序比较不同算法在同样页面访问情况下性能，用缺页率、缺页次数表示。

**结果分析：**

根据最终的结果，我们可以对每种页面替换算法的性能进行详细分析。分析的依据是每种算法的缺页次数和缺页率，这些数据直接反映了算法在处理页面访问序列时的效率。

FIFO算法（先进先出算法）

缺页次数：17次 缺页率：0.85

FIFO算法的缺页次数和缺页率相对较高，这表明该算法在处理页面访问请求时频繁地发生缺页。FIFO算法简单且易于实现，但由于其不考虑页面未来使用情况，导致性能不是最优的。特别是在页面访问序列中存在大量连续访问不同页面的情况时，FIFO算法的性能会显著下降。

2. LRU算法（最近最少使用算法）

缺页次数：9次 缺页率：0.45

LRU算法的缺页次数和缺页率相对较低，表明其在处理页面访问请求时较为高效。LRU算法通过跟踪页面的访问历史，优先替换最长时间未被访问的页面，因此在多数情况下能够较好地预测未来的访问模式，从而减少缺页的发生。

3. OPT算法（最佳替换算法）

缺页次数：10次 缺页率：0.5

OPT算法的缺页次数和缺页率介于FIFO和LRU之间。OPT算法理论上能够达到最低的缺页率，因为它总是选择将来最迟被访问的页面进行替换。然而，在实际中，OPT算法需要知道未来的访问序列，这在实际操作中是不可能的，因此其实际应用受限。

4. Clock算法（简单时钟算法）

缺页次数：15次 缺页率：0.75

Clock算法的缺页次数和缺页率较高，这可能是因为它简单地循环遍历物理块，而没有有效地考虑页面的未来使用情况。Clock算法通过循环列表模拟页面的使用情况，虽然比FIFO算法复杂，但仍然不如LRU算法那样能够准确预测页面的未来访问。

5. 改进Clock算法

缺页次数：14次 缺页率：0.7

改进的Clock算法在Clock算法的基础上增加了对页面修改位的考虑，优先替换未被修改的页面。这种改进使得算法在某些情况下能够更好地预测页面的未来使用情况，从而降低缺页次数和缺页率。

从结果来看，LRU算法在这些算法中表现最佳，具有最低的缺页次数和缺页率，这与其能够较好地预测页面未来访问的特性有关。OPT算法虽然理论上最优，但在实际应用中受限于对未来访问序列的预知。FIFO和Clock算法由于缺乏对页面未来使用情况的考虑，导致较高的缺页率。改进的Clock算法通过考虑页面的修改状态，相对于基本的Clock算法有所改进，但仍不如LRU算法。

**五、课程设计总结**

各个页面替换算法有以下特点：

OPT算法提供了理论最优解，但实际中难以实现。

FIFO算法简单易实现，但可能不是性能最优。

LRU算法性能较好，但实现复杂。

Clock算法是LRU算法的简化版本，易于实现，但不准确。

改进Clock算法在Clock算法上增加对页面修改状态的考虑，提高了算法的性能。

综上所述，LRU算法在这些算法中表现最佳，缺页率最低，但实现复杂度也最高。FIFO算法虽然实现简单，但性能最差。OPT算法理论上最优，但实际应用受限。Clock算法和改进Clock算法作为LRU算法的近似，提供了一个性能和实现复杂度之间的平衡。在实际的操作系统设计中，选择页面替换算法时需要考虑算法的预测能力和实现的复杂性。LRU算法因其较好的性能和相对简单的实现而广泛使用，而OPT算法则更多地作为理论参考。

在实验过程中，我遇到了不少挑战，包括对某些算法原理的理解不深入、代码实现中的错误、以及程序运行时的异常。例如，LRU算法在实现时出现了段错误（Segmentation fault），这通常是由于数组越界或非法内存访问引起的。通过仔细检查代码和调试，我找到了问题所在，并修正了算法实现中的逻辑错误。总的来说，这次实验不仅加深了我对操作系统内存管理的理解，也提升了我的编程技能和问题解决能力。通过在Linux环境下完成实验，我更加熟悉了Linux系统，这对于我未来在计算机科学领域的学习和研究将大有裨益。

**参考文献**

［1］CSDN.OPT、FIFO、LRU算法的实现. <https://www.cnblogs.com/wasi-991017/p/13072328.html>

［2］CSDN.操作系统——页面置换FIFO、LRU、OPT、CLOCK、改进版CLOCK、LFU等算法. <https://blog.csdn.net/qq_51561585/article/details/113114448>

［3］阿里云平台. 在服务器配置c++开发环境. <https://developer.aliyun.com/article/984289>