北京郵電大學

实验报告：实验 4 内存管理实验

学院：计算机学院（国家示范性软件学院）

专业：计算机科学与技术

班级：2022211305

学号：2022211683

姓名：张晨阳

2024 年 12 月 9 号

目录

1.实验概述

1.1. 实验目的

基于 openEuler 操作系统，通过模拟实现按需调页式存储管理的几种基本页面置换算法，了解虚拟存储技术的特点，掌握虚拟存储按需调页式存储管理中几种基本页面置换算法的基本思想和实现过程，并比较它们的效率。

1.2. 实验内容及要求

首先用 srand()和 rand()函数定义和产生指令地址序列，然后将指令地址序列变换成相应的页地址流。

设计算法，计算访问缺页率并对算法的性能加以比较。

（1）最优置换算法（Optimal）

（2）最近最少使用（Least Recently Used）

（3）先进先出法（Fisrt In First Out）

其中，缺页率= 页面失效次数/ 页地址流长度

要求：

分析在同样的内存访问串上执行，分配的物理内存块数量和缺页率之间的关系; 并在同样情况下，对不同置换算法的缺页率比较。

1.3. 实验环境

- Visual Studio Code 1.94.2
- c11
- gcc 8.1.0

2.程序设计说明

2.1. 生成内存访问串

通过随机数产生一个内存地址，共 100 个地址，地址按下述原则生成：

1) 70%的指令是顺序执行的
2) 10%的指令是均匀分布在前地址部分
3) 20%的指令是均匀分布在后地址部分

具体的实施方法是：

a) 从地址 0 开始;

b) 若当前指令地址为 m，按上面的概率确定要执行的下一条指令地址，分别为顺序、在前和在后：

- 顺序执行： 地址为 $m+1$ 的指令;
- 在前地址：$[0，m-1]$中依前面说明的概率随机选取地址;
- 在后地址：$[m+1，99]$中依前面说明的概率随机选取地址;

c) 重复（b）直至生成 100 个指令地址。

假设每个页面可以存放 4 条指令，将指令地址映射到页面，生成内存访问串。

代码实现如下：

```
#define PAGE_SIZE 4       // 每页存储 PAGE_SIZE 条指令
#define ACCESS_COUNT 100  // 总共生成 100 个指令地址
#define MEMORY_SIZE 15    // 设置物理内存大小（页面数量）

// 生成内存访问串
void generate_page_access_sequence(int* page_access) {
    srand(time(NULL));  // 使用当前时间作为随机种子
    int m = 0;          // 初始地址

    for (int i = 0; i < ACCESS_COUNT; i++) {
        page_access[i] = m / PAGE_SIZE;  // 将内存地址映射到页面

        int rand_choice = rand() % 10;  // 随机生成下一条指令的访问模式
        if (rand_choice < 7) {
```

```
            m++;  // 顺序执行
        } else if (rand_choice < 8) {
            m = rand() % (m > 0 ? m : 1);
        } else {
            if (m < ACCESS_COUNT - 1) {
                m = rand() % (ACCESS_COUNT - m - 1) + m + 1;
            } else {
                m++;
            }
        }
        m %= ACCESS_COUNT;  // 确保地址在 0 到 ACCESS_COUNT-1 范围内
    }
}
```

针对边界情况的特殊处理：

- 当$m = 0$时出现了10%的概率，则令下一个地址为 0
- 当$m = 99$时出现了20%的概率，则令下一个地址为 0

2.2. 页面置换算法设计

2.2.1.FIFO 页面置换算法

FIFO（First In, First Out）算法是最简单的页面置换算法。其核心思想是：在物理内存满时，最先进入内存的页面最先被置换出去。

详细流程：

1. 初始化内存为空（-1 表示没有页面）。
2. 遍历每次页面访问，检查页面是否在内存中：
   - 如果页面命中（即已经在内存中），则继续处理下一个页面。
   - 如果页面未命中（即不在内存中），则发生缺页：
     - 将新页面加载到内存中的 ptr 位置。
     - ptr 会顺序循环，指向下一个位置，模拟先进先出的操作。
3. 计算缺页次数并返回缺页率。

具体代码实现如下：

```
int page_faults = 0;
int ptr = 0;                                    // 指向下一个空位置
memset(memory, -1, memory_size * sizeof(int));  // 初始化内存为空

// 遍历所有页面访问
for (int i = 0; i < ACCESS_COUNT; ++i) {
    bool hit = false;
    // 查找当前页面是否在内存中
    for (int j = 0; j < memory_size; ++j)
        if (memory[j] == pages[i]) {
            hit = true;
            break;
        }

    if (!hit) {
        page_faults++;
        memory[ptr] = pages[i];         // 将页面载入内存
        ptr = (ptr + 1) % memory_size;  // 轮询指针，循环利用内存帧
    }
}
return page_faults;
```

2.2.2.LRU 页面置换算法

LRU（Least Recently Used）算法的基本思想是：每次置换时，选择最近最少使用的页面进行替换。LRU 根据页面的访问历史来决定哪些页面在未来最不可能被访问，从而优先将它们替换出去。

详细流程：

1. 初始化内存为空，并且为每个页面分配一个访问计数器。
2. 遍历每次页面访问：
   - 如果页面命中，更新该页面的访问计数为 0，并将其他页面的访问计数加 1。
   - 如果页面未命中，发生缺页：
     - 选择访问计数最大的页面，将其替换。
     - 重置新页面的访问计数，并更新所有其他页面的访问计数。
3. 计算缺页次数并返回缺页率。

具体代码实现如下：

```
1. int lru_page_replacement(int* memory, int memory_size, int* pages) {
2.     int page_faults = 0;
3.     memset(memory, -1, memory_size * sizeof(int));
4.     int* access_count = (int*)malloc(memory_size * sizeof(int));
5.     memset(access_count, 0, memory_size * sizeof(int));
6.
7.     // 遍历所有页面访问
8.     for (int i = 0; i < ACCESS_COUNT; ++i) {
9.         int max_index = 0, max_count = 0;
10.        bool hit = false;
11.
12.        // 查找页面是否已经在内存中
13.        for (int j = 0; j < memory_size; ++j) {
14.            if (memory[j] == pages[i]) {
15.                hit = true;
16.                max_index = j;
17.                break;
18.            }
19.            if (memory[j] == -1) {  // 找到空帧，直接放入
20.                max_index = j;
21.                break;
22.            }
```

```
            // 找到访问次数最多的页面，进行置换
            if (access_count[j] > max_count) {
                max_count = access_count[j];
                max_index = j;
            }
        }

        // 更新每个页面的访问计数
        for (int j = 0; j < memory_size; ++j)
            access_count[j]++;

        access_count[max_index] = 0;  // 重置当前页面的访问计数

        // 如果页面未命中，发生缺页
        if (!hit) {
            page_faults++;
            memory[max_index] = pages[i];  // 置换页面
        }
    }

    free(access_count);
    return page_faults;
}
```

2.2.3.Optimal 页面置换算法

Optimal 算法是最理想的页面置换策略，其基本思想是：每次置换时，选择在未来最久不会被访问的页面进行替换。理想的情况下，最优算法能够达到最小的缺页率。

详细流程：

1. 初始化 next_access 数组为 ACCESS_COUNT，表示没有后续访问。
2. 从后向前遍历页面访问序列，计算每个页面的下一次访问时间，并更新 next_access 数组。
3. 遍历每次页面访问：
   - 如果页面已经在内存中，则继续处理下一个页面。
   - 如果页面未命中，发生缺页：
     - 找到未来访问时间最远的页面，进行替换。
4. 计算缺页次数并返回缺页率。

具体代码实现如下：

```
int optimal_page_replacement(int* memory, int memory_size, int* pages) {
    int page_faults = 0;
    memset(memory, -1, memory_size * sizeof(int));
    int** next_access = (int**)malloc(ACCESS_COUNT * sizeof(int*));
    for (int i = 0; i < ACCESS_COUNT; ++i)
        next_access[i] = (int*)malloc(memory_size * sizeof(int));

    // 初始化 next_access 数组为 ACCESS_COUNT，表示没有后续访问
    for (int i = 0; i < ACCESS_COUNT; ++i)
        for (int j = 0; j < memory_size; ++j)
            next_access[i][j] = ACCESS_COUNT;

    // 计算每个页面的未来访问位置
    for (int i = ACCESS_COUNT - 2; i >= 0; --i) {
        for (int j = 0; j < memory_size; ++j) {
            if (pages[i + 1] == j)
                next_access[i][j] = i + 1;
            else
                next_access[i][j] = next_access[i + 1][j];
        }
    }

```

```
    // 遍历所有页面访问
    for (int i = 0; i < ACCESS_COUNT; ++i) {
        int max_index = 0, max_count = 0;
        bool hit = false;

        // 查找页面是否在内存中
        for (int j = 0; j < memory_size; ++j) {
            if (memory[j] == pages[i]) {
                hit = true;
                break;
            }
            if (memory[j] == -1) {  // 找到空帧，直接放入
                max_index = j;
                break;
            }
            // 找到下次访问距离最远的页面，进行置换
            if (next_access[i][memory[j]] > max_count) {
                max_count = next_access[i][memory[j]];
                max_index = j;
            }
        }

        // 如果页面未命中，发生缺页
        if (!hit) {
            page_faults++;
            memory[max_index] = pages[i];  // 置换页面
        }
    }

    // 释放内存
    for (int i = 0; i < ACCESS_COUNT; ++i)
        free(next_access[i]);
    free(next_access);

    return page_faults;
}
```

3.结果分析

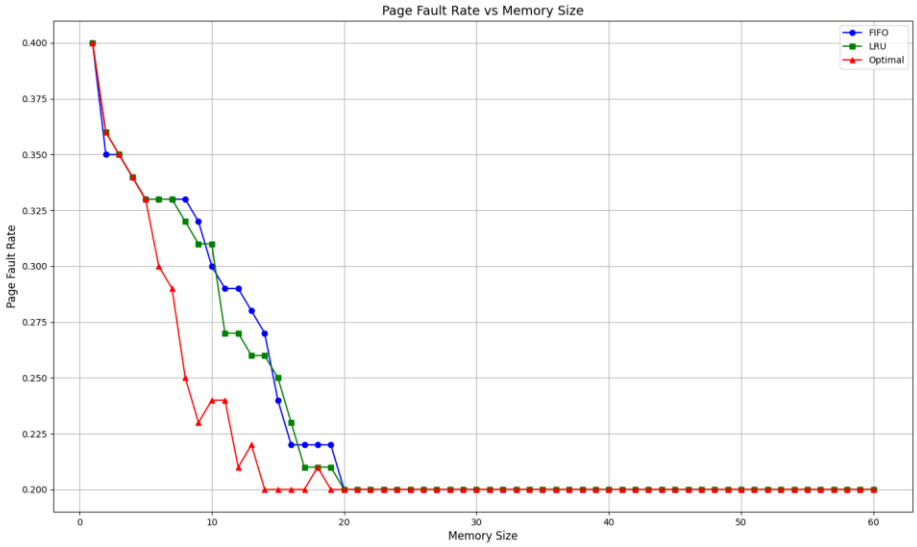
3.1. 物理内存块数量与缺页率

给出以下设定：

每个页面可以存放 4 条指令，生成 100 条执行指令，序列如下：

```
0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 2, 2, 1, 1, 1, 1, 2, 2, 20, 21, 21, 7, 7, 8, 8,
22, 23, 23, 23, 23, 24, 24, 24, 7, 18, 13, 13, 13, 14, 14, 14, 14, 3, 3, 3, 14, 14,
14, 15, 3, 3, 3, 4, 4, 4, 4, 5, 21, 23, 24, 24, 0, 0, 19, 19, 19, 20, 24, 24, 8, 8,
8, 8, 11, 18, 18, 21, 21, 21, 21, 22, 22, 22, 17, 17, 23, 23, 24, 24, 0, 0, 0, 0,
0, 0, 0, 0, 1, 0
```

对于三种算法，不同的物理内存块数量对应的缺页率如下：



由图可知：

1) 随着物理内存块数量的增加，总体的缺页率都呈下降趋势；
2) Optiaml 算法总体上优于另外两种算法；
3) Optimal 算法的缺页率随物理内存块的增加，下降的最快；
4) 当物理内存块大于总页数时，缺页率趋于稳定，即各页面首次未命中的数量。

3.2. 不同置换算法的缺页率比较

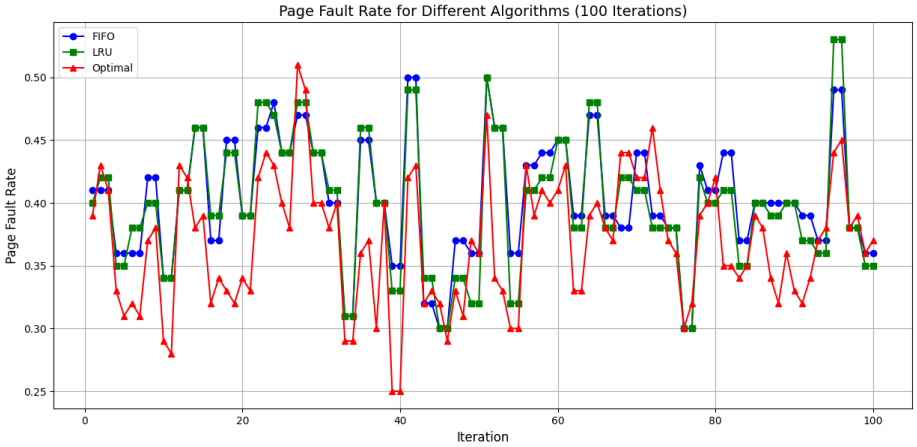
给出以下设定：

(1) 每个页面可以存放 3 条指令；

(2) 随机生成 100 条执行指令；

(3) 物理内存块数量为 8

三种算法均执行 100 次，缺页率结果如下：



由图可知：

1) Optimal 算法的缺页率基本都明显低于 FIFO 和 LRU；
2) LRU 和 FIFO 大部分情况下的缺页率都非常接近；
3) 一些特殊情况可以归结为生成数据的随机性；
4) 与 FIFO 相比，LRU 没有表现出该有的性能优势，这里认为是数据生成的不够合适。

4.心得总结

在这次页面置换算法实验中，我深入探索了不同的页面置换算法，并通过编写程序和生成大量的测试数据，对其缺页率进行了系统的分析和比较。

在实验中，我首先通过 C 语言编写了三种页面置换算法的实现，并生成了页面访问序列。这些访问序列模拟了真实系统中的内存访问情况。

为了深入分析算法的表现，我设计了程序分别在不同的内存大小下进行实验，并计算每种算法的缺页率。

内存大小的变化：通过调整内存大小，我观察到随着内存的增大，缺页率通常呈现下降趋势。这是因为内存可以容纳更多的页面，从而减少了替换的频率。

不同算法的效率比较：通过多次实验，我发现最优算法的缺页率最低，但由于其依赖于未来信息，因此不适用于实际应用。理论上，LRU 算法则在大多数情况下表现优于 FIFO，尤其是在内存较小的情况下。FIFO 算法虽然简单易实现，但在访问模式不规律时常常会出现较高的缺页率。

为了直观地展示实验结果，我使用 Python 和 matplotlib 库对每种算法的缺页率进行了批量计算和可视化展示。通过绘制缺页率折线图，我能够清晰地看到不同内存大小下各算法的表现差异，这使得实验数据的分析更加直观。

通过这次实验，我对页面置换算法有了更深刻的理解。

本次实验不仅增强了我对操作系统内存管理机制的理解，也提升了我的编程能力，尤其是如何将算法与实际应用相结合的能力。