实验三：请求页式存储管理

姓名：汪江豪　学号：22121630　实验日期：2024.12.16

---

1. 实验环境：

实验设备：Lenovo Legion R7000P2021H
开发环境：VScode

2. 实验要求：

（1）实现指令地址流->页号的转换；

（2）实现算法：最佳淘汰算法(OPT)、最近最少使用页淘汰算法(LRU)、先进先出淘汰算法(FIFO)。并提供页面置换算法演示:页面引用顺序可随机定义，给出页面置换的全部过程，给出缺页率。

（3）给出 OPT,LRU,FIFO 三种算法在分配不同物理块下的缺页次数和命中率对比。

3. 实现代码：

3.1 页号转换：

```
void vir_addr_to_page()
{
    srand(static_cast<int>(time(0)));
    int n = 10;
    for (int i = 1; i <= n; i++)
    {
        // 生成随机地址
        int t1 = rand() % MAX_ADDR;
        vir_addr[i] = t1;
        // 获取页号
        page[i] = t1 / PAGE_SIZE;
    }
    cout << "虚拟地址流:" << endl;
    for (int i = 1; i <= n; i++)
    {
        printf("vir_addr[%d] = %-10d", i, vir_addr[i]);
        cout << (i % 4 == 0 ? "\n" : "    ");
    }
    cout << endl;
    cout << "页号流:" << endl;
    for (int i = 1; i <= n; i++)
    {
        printf("page[%d] = %-5d", i, page[i]);
        cout << (i % 4 == 0 ? "\n" : " ");
    }
```

```
    cout << endl;
}
```

3.2 实现三种算法：

3.2.1 OPT 算法：

```
// OPT 算法
int OPT(vector<int> &list, int memory_num)
{
    int n = list.size();
    int page_fault = 0; // 缺页数
    vector<int> memory;
    int k = 0;
    memset(res, '_', sizeof(res));
    // 遍历整个请求页面好序列
    for (int i = 0; i < n; i++)
    {
        int &x = list[i];
        bool flag = false;
        for (auto &y : memory)
        {
            if (y == x)
            {
                flag = true;
                res[memory_num][k] = 'T';
                k++;
                break;
            }
        }
        // 如果 x 不在内存中
        if (!flag)
        {
            res[memory_num][k] = 'F';
            page_fault++;
            if (memory.size() < memory_num)
                memory.push_back(x);
            else
            {
                // 找到序列中在 memory 里最远的页号
                int s[n], top = 0;
                bool st[memory_num];
                memset(st, false, sizeof(st));
                for (int j = i + 1; j < n; j++)
                {
                    int x2 = list[j];
                    // 对后续序列中的每项，遍历内存块
```

```
                    for (int u = 0; u < memory.size(); u++)
                    {
                        int &y = memory[u];
                        if (y == x2)
                        {
                            if (!st[u])
                            {
                                st[u] = true;
                                s[top++] = y;
                            }
                        }
                    }
                }
                top--;

                // 找到后，更新内存块中指定块的内容
                bool done = false;
                for (int u = 0; u < memory_num; u++)
                    if (!st[u])
                    {
                        memory[u] = x;
                        done = true;
                        break;
                    }
                if (done == false)
                {
                    for (auto &y : memory)
                        if (y == s[top])
                            y = x;
                }
            }
            // 更新 res 数组
            for (int i = 0; i < memory.size(); i++)
                res[i][k] = memory[i] + '0';
            k++;
        }
    }
    return page_fault;
}
```

3.2.2 FIFO 算法：

```
// FIFO 算法
int FIFO(vector<int> &list, int memory_num)
{
    int n = list.size();
```

```
int page_fault = 0; // 缺页数
deque<int> q;
vector<int> memory;
int k = 0;
memset(res, '_', sizeof(res));
for (auto &x : list)
{
    bool flag = false;
    bool isfull = false;
    char change;
    for (auto &y : memory)
    {
        if (y == x)
        {
            flag = true;
            res[memory_num][k] = 'T';
            k++;
            break;
        }
    }
    // 如果 x 不在内存中
    if (!flag)
    {
        res[memory_num][k] = 'F';
        page_fault++;
        if (memory.size() < memory_num)
        {
            memory.push_back(x);
            q.push_back(x);
        }
        else
        {
            isfull = true;
            change = q.front();
            q.pop_front();
            q.push_back(x);
            for (auto &y : memory)
                if (y == change)
                    y = x;
        }
        // 更新 res 数组,如果 flag 为 true,则不更新
        for (int i = 0; i < memory.size(); i++)
            res[i][k] = memory[i] + '0';
        k++;
```

```
        }
    }
    return page_fault;
}
```

3.2.3 LRU 算法：

```
int LRU(vector<int> &list, int memory_num)
{
    int n = list.size();
    int page_fault = 0; // 缺页数
    vector<int> memory;
    int k = 0;
    memset(res, '_', sizeof(res));
    // 遍历整个请求页面好序列
    for (int i = 0; i < n; i++)
    {
        int &x = list[i];
        bool flag = false;
        for (auto &y : memory)
        {
            if (y == x)
            {
                flag = true;
                res[memory_num][k] = 'T';
                k++;
                break;
            }
        }
        // 如果 x 不在内存中
        if (!flag)
        {
            res[memory_num][k] = 'F';
            page_fault++;
            if (memory.size() < memory_num)
                memory.push_back(x);
            else // memory 已满，向左找，一定能找到
            {
                // 找到序列中在 memory 里最远的页号
                int top;
                bool st[memory_num];
                memset(st, false, sizeof(st));
                for (int j = i - 1; j >= 0; j--)
                {
                    int x2 = list[j];
                    // 对后续序列中的每项，遍历内存块
```

```
                    int cnt = 0;
                    for (int u = 0; u < memory.size(); u++)
                    {
                        int &y = memory[u];
                        if (y == x2)
                        {
                            if (!st[u])
                            {
                                st[u] = true;
                                top = y;
                                cnt++;
                                if (cnt == memory_num)
                                    break;
                            }
                        }
                    }
                }
                // 找到后，更新内存块中指定块的内容
                for (auto &y : memory)
                    if (y == top)
                    {
                        y = x;
                        break;
                    }
            }
            // 更新 res 数组
            for (int i = 0; i < memory.size(); i++)
                res[i][k] = memory[i] + '0';
            k++;
        }
    }
    return page_fault;
}
```

3.3 三种算法比较：

```
void compare(vector<int> &list)
{
    for (int memory_num = 3; memory_num <= 5; memory_num++)
    {
        cout << "================================================" <<
endl;
        cout << "页面大小: 1024B  内存块数: " << memory_num << endl;
        int n = list.size();
        int num_FIFO = FIFO(list, memory_num);
        double rate_FIFO = num_FIFO / (double)n;
```

```
        double hit_FIFO = 1 - rate_FIFO;
        int num_OPT = OPT(list, memory_num);
        double rate_OPT = num_OPT / (double)n;
        double hit_OPT = 1 - rate_OPT;
        int num_LRU = LRU(list, memory_num);
        double rate_LRU = num_LRU / (double)n;
        double hit_LRU = 1 - rate_LRU;
        printf("OPT    缺页数: %-3d    缺页率: %-4.3lf    命中率: %-4.3lf\n",
num_FIFO, rate_FIFO, hit_FIFO);
        printf("FIFO   缺页数: %-3d    缺页率: %-4.3lf    命中率: %-4.3lf\n",
num_OPT, rate_OPT, hit_OPT);
        printf("LRU    缺页数: %-3d    缺页率: %-4.3lf    命中率: %-4.3lf\n",
num_LRU, rate_LRU, hit_LRU);
    }
}
```

4. 结果：

4.1 页号的转换

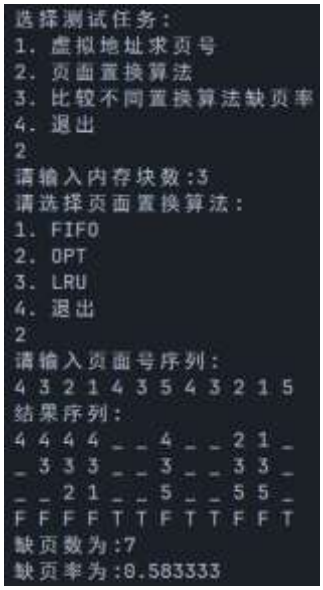
```
选择测试任务：
1. 虚拟地址求页号
2. 页面置换算法
3. 比较不同置换算法缺页率
4. 退出
1
虚拟地址流：
vir_addr[1] = 29219         vir_addr[2] = 5657          vir_addr[3] = 11936         vir_addr[4] = 9283
vir_addr[5] = 24677         vir_addr[6] = 1228          vir_addr[7] = 15708         vir_addr[8] = 20941
page[1] = 28    page[2] = 5     page[3] = 11    page[4] = 9
page[5] = 24    page[6] = 1     page[7] = 15    page[8] = 20
page[9] = 2     page[10] = 8
```

4.2 三种算法测试

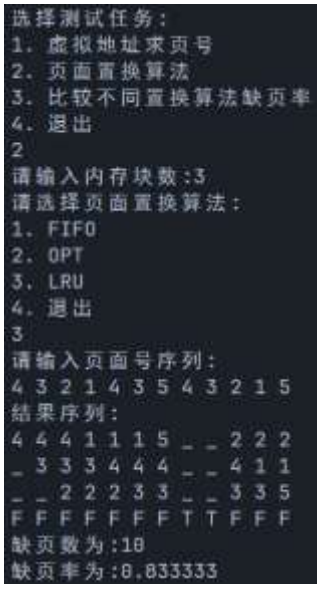
4.2.1 FIFO 算法

```
选择测试任务：
1. 虚拟地址求页号
2. 页面置换算法
3. 比较不同置换算法缺页率
4. 退出
2
请输入内存块数:3
请选择页面置换算法：
1. FIFO
2. OPT
3. LRU
4. 退出
1
请输入页面号序列：
4 3 2 1 4 3 5 4 3 2 1 5
结果序列：
4 4 4 1 1 1 5 _ _ 5 5 _
_ 3 3 3 4 4 4 _ _ 2 2 _
_ _ 2 2 2 3 3 _ _ 3 1 _
F F F F F F F T T F F T
缺页数为:9
缺页率为:0.75
```

4.2.2 OPT 算法

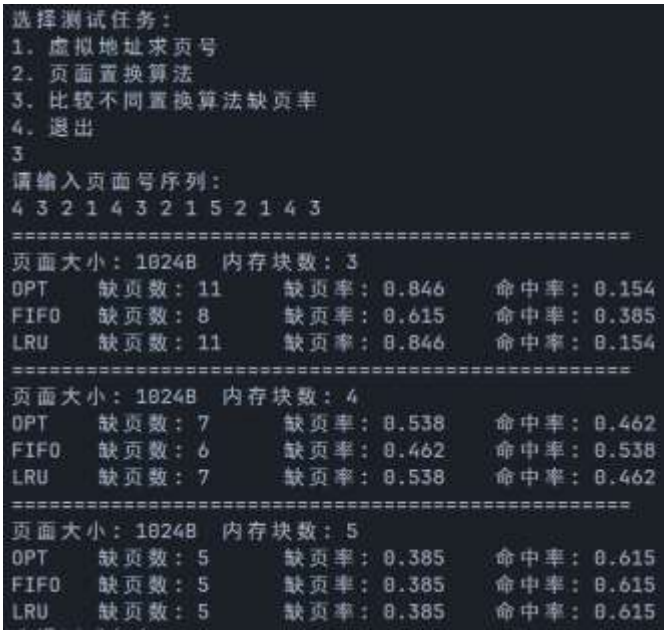


4.2.3 LRU 算法



4.3 三种算法比较：

输入页面序列号后，默认页大小 1024B，分别对内存块数 3，4，5 情况下进行了比较：



体会：

在本次请求页式存储管理实验中，我实现了虚拟地址的页号转换，三种页面置换算法：OPT，FIFO，LRU 算法，并基于不同的内存块数，分别对这三种算法的缺页率和命中率进行了比较。这让我对页面置换算法有了更加深入的理解。

OPT 最优算法，追求理论上的最优解，但在实际中难以实现。

FIFO 先进先出算法简单，但可能不够高效，并且在实际过程中，可能出现 Belady 异常，在增加分配给进程的物理页面的情况下，进程的缺页率反而增加，这种现象在 FIFO 算法中极为常见。

LRU 最近最少使用算法则在实际应用中较为常见，因为它更加贴合程序的实际访问模式，更加符合程序的时间局部性原理。

总之，本次实验不仅加深了我对于 OS 页面置换算法的理解，还提升了我的编程实践能力，让我受益匪浅。