算法设计与分析实验报告

倪玮昊

2020211346

一、实验目的与内容

实验目的

基于 Windows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。

实验内容

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

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）

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。

二、实验过程

思路

先对访问页面初始化,随机生成100个访问位置,再去记录其页号,再循环不同的内存块来计算各种算法的效率

关键函数及代码段的描述

形成访问序列,按题目的要求

```
void formTable(){
    srand((int)time(0));
    Order[0] = 0;
    int opt;
    for (int i = 1; i < 100; i++) {
        opt = rand() % 10;
```

```
        if (opt <= 6 && opt >= 0) {
            Order[i] = Order[i - 1] + 1;
        }//70%按顺序去读取地址
        else if (opt <= 7) {//在前地址的情况
            if (Order[i - 1] == 0) { //处理边界情况
                i--;
                continue;
            }
            Order[i] = rand() % Order[i - 1];
        }
        else {
            if (Order[i - 1] > 98) { //处理边界情况
                i--;
                continue;//重新生成
            }
            Order[i] = (rand() % (99 - Order[i - 1])) + Order[i - 1] + 1;
        }
        if (Order[i] > 99) { //若有超过地址范围的指令，则重新生成
            i--;
        }
    }
}
```

初始化内存块

```
for (int i = 0; i < numOfBlock; i++) {
        block[i].page_id = -1;
        block[i].visit = -1;
    }
```

用不同大小内存块去测试算法效率

```
for (int i = 2; i <= 12; i++) {//内存块大小
        numOfBlock = i;
        pageinfo block[i];//申请内存块大小数组
        printf("对大小为%d的内存块,各算法访问效率为:\n",i);
        init(block);
        optimal(block);
        init(block);
        LRU(block);
        init(block);
        FIFO(block);
        printf("\n");
    }
```

FIFO算法

```
int page_falut = 0; //缺页数
bool is_find; //页面是否已在内存块中
bool is_full; //内存块是否满
for (int i = 0; i < 100; i++) {
    is_find = false;
    is_full = true;
    for (int j = 0; j < numOfBlock; j++) {
```

```
        if (block[j].page_id == pageNum[i]) {
            is_find = true;
        }
    }
    if (is_find == false) {
        page_falut++;
        for (int j = 0; j < numOfBlock; j++) {
            if (block[j].page_id == -1) {
                is_full = false;
                block[j].page_id = pageNum[i];
                break;
            }
        }
        if (is_full) {
            for (int j = 1; j < numOfBlock; j++) {//将最先进的页面替换出去
                block[j - 1].page_id = block[j].page_id;
            }
            block[numOfBlock - 1].page_id = pageNum[i];
        }
    }
}
printf("FIFO算法的缺页次数:%d,缺页率:%f\n", page_falut, (page_falut / 100.0));
```

optimal算法

```
  int page_falut = 0; //缺页数
    bool is_find; //页面是否已在内存块中
    bool is_full; //内存块是否满
    int opt_value;//用于表示页面在之后的页面流里出现的先后顺序
    for (int i = 0; i < 100; i++) {
        is_find = false;
        is_full = true;
        for (int j = 0; j < numOfBlock; j++) {
            if (block[j].page_id == pageNum[i]) {
                is_find = true;
            }
        }
        if (is_find == false) {
            page_falut++;
            for (int j = 0; j < numOfBlock; j++) {
                if (block[j].page_id == -1) {
                    is_full = false;
                    block[j].page_id = pageNum[i];
                    break;
                }
            }
            if (is_full) {
                for (int j = 0; j < numOfBlock; j++) {
                    for (int k = i + 1; k < 100; k++) { //对之后的页面流遍历
                        if (block[j].page_id == pageNum[k]) {
                            block[j].visit = k;
                            break;
                        }
                    }
                }
```

```
                int max = 0;
                int best_page; // 最适合替换出的页面
                for (int j = 0; j < numOfBlock; j++) {
                    if (block[j].visit == -1) { //若该页面在之后都没出现过
                        best_page = j;
                        break;
                    }
                    else {
                        if (block[j].visit > max) {
                            max = block[j].visit;
                            best_page = j;
                        }
                    }
                }
                block[best_page].page_id = pageNum[i];
                for (int j = 0; j < numOfBlock; j++) { //重置block的访问标记
                    block[j].visit = -1;
                }
            }
        }
    }
    printf("optimal算法的缺页次数:%d,缺页率:%f\n", page_falut, (page_falut /
100.0));
```

LRU算法

```
int page_falut = 0; //缺页数
bool is_find; //页面是否已在内存块中
bool is_full; //内存块是否满
for (int i = 0; i < 100; i++) {
    is_find = false;
    is_full = true;
    for (int j = 0; j < numOfBlock; j++) {
        if (block[j].page_id != -1) {
            block[j].visit++; //若该页已在内存块，则每次都要给一个累计增值
            if (block[j].page_id == pageNum[i]) {
                is_find = true;
                block[j].visit = -1; //若命中，直接清除累计
            }
        }
    }
    if (is_find == false) {//没有命中的情况下
        page_falut++;
        for (int j = 0; j < numOfBlock; j++) {
            if (block[j].page_id == -1) {
                is_full = false;
                block[j].page_id = pageNum[i];
                block[j].visit = -1;//未被访问
                break;
            }
        }
        if (is_full) {//已满的情况下
            int longest = -2;//记录最久未使用的页面
            int best_page;
            for (int j = 0; j < numOfBlock; j++) {
```

```
                if (block[j].visit > longest) {
                    best_page = j;
                    longest = block[j].visit;
                }
            }
            block[best_page].page_id = pageNum[i];
            block[best_page].visit = -1;
        }
    }
}
printf("LRU算法的缺页次数:%d,缺页率:%f\n",  page_falut, (page_falut / 100.0));
```

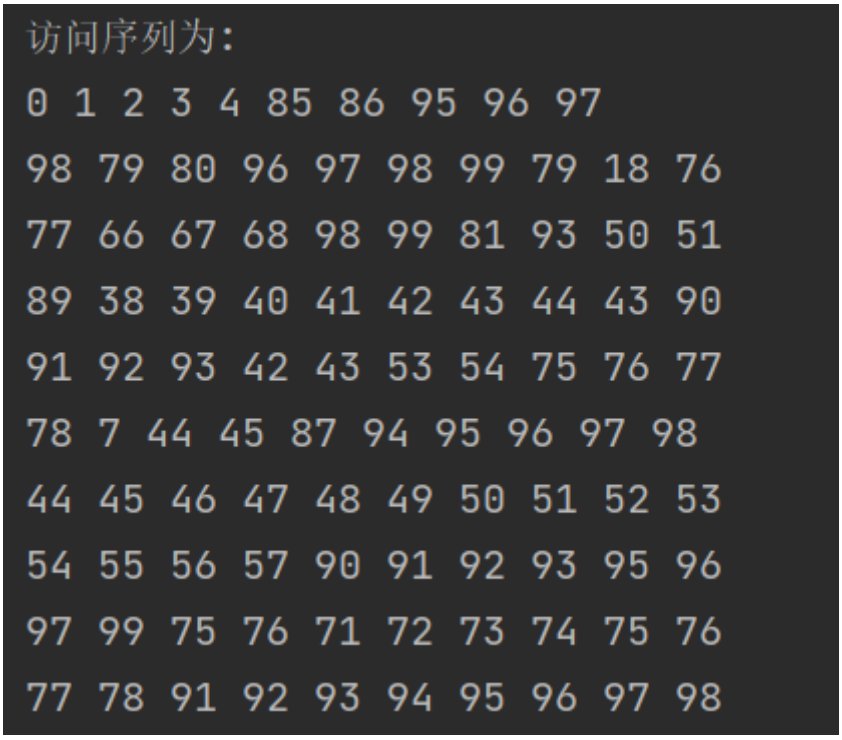
三、实验结果

程序执行环境及运行方式

win11,CLion

程序执行示例

输出:



```
对大小为2的内存块,各算法访问效率为:
optimal算法的缺页次数:20,缺页率:0.200000
LRU算法的缺页次数:26,缺页率:0.260000
FIFO算法的缺页次数:25,缺页率:0.250000

对大小为3的内存块,各算法访问效率为:
optimal算法的缺页次数:15,缺页率:0.150000
LRU算法的缺页次数:20,缺页率:0.200000
FIFO算法的缺页次数:20,缺页率:0.200000

对大小为4的内存块,各算法访问效率为:
optimal算法的缺页次数:12,缺页率:0.120000
LRU算法的缺页次数:18,缺页率:0.180000
FIFO算法的缺页次数:18,缺页率:0.180000

对大小为5的内存块,各算法访问效率为:
optimal算法的缺页次数:10,缺页率:0.100000
LRU算法的缺页次数:15,缺页率:0.150000
FIFO算法的缺页次数:15,缺页率:0.150000

对大小为6的内存块,各算法访问效率为:
optimal算法的缺页次数:9,缺页率:0.090000
LRU算法的缺页次数:12,缺页率:0.120000
FIFO算法的缺页次数:14,缺页率:0.140000

对大小为7的内存块,各算法访问效率为:
optimal算法的缺页次数:9,缺页率:0.090000
LRU算法的缺页次数:10,缺页率:0.100000
FIFO算法的缺页次数:13,缺页率:0.130000
```

```
对大小为8的内存块,各算法访问效率为:
optimal算法的缺页次数:9,缺页率:0.090000
LRU算法的缺页次数:10,缺页率:0.100000
FIFO算法的缺页次数:13,缺页率:0.130000

对大小为9的内存块,各算法访问效率为:
optimal算法的缺页次数:9,缺页率:0.090000
LRU算法的缺页次数:9,缺页率:0.090000
FIFO算法的缺页次数:9,缺页率:0.090000

对大小为10的内存块,各算法访问效率为:
optimal算法的缺页次数:9,缺页率:0.090000
LRU算法的缺页次数:9,缺页率:0.090000
FIFO算法的缺页次数:9,缺页率:0.090000

对大小为11的内存块,各算法访问效率为:
optimal算法的缺页次数:9,缺页率:0.090000
LRU算法的缺页次数:9,缺页率:0.090000
FIFO算法的缺页次数:9,缺页率:0.090000

对大小为12的内存块,各算法访问效率为:
optimal算法的缺页次数:9,缺页率:0.090000
LRU算法的缺页次数:9,缺页率:0.090000
FIFO算法的缺页次数:9,缺页率:0.090000
```

四、 实验总结

三种算法有各自优缺点

1. 最佳置换算法(OPT)

最佳(Optimal, OPT)置换算法所选择的被淘汰页面将是以后永不使用的，或者是在最长时间内不再被访问的页面,这样可以保证获得最低的缺页率。但由于人们目前无法预知进程在内存下的若干页面中哪个是未来最长时间内不再被访问的，因而该算法无法实现。

2. 先进先出(FIFO)页面置换算法

优先淘汰最早进入内存的页面，亦即在内存中驻留时间最久的页面。该算法实现简单，只需把调入内存的页面根据先后次序链接成队列，设置一个指针总指向最早的页面。但该算法与进程实际运行时的规律不适应，因为在进程中，有的页面经常被访问。

3. 最近最久未使用(LRU)置换算法

选择最近最长时间未访问过的页面予以淘汰，它认为过去一段时间内未访问过的页面，在最近的将来可能也不会被访问。该算法为每个页面设置一个访问字段，来记录页面自上次被访问以来所经历的时间，淘汰页面时选择现有页面中值最大的予以淘汰。

在实验中可以看出来:

对于同一个算法来说，分配的内存块数量增加，会使得缺页次数和缺页率明显下

降.

而在同样的情况下，optimal算法无疑是最佳的，体现出了最优的算法性能，有着较低的缺页率。

而LRU和FIFO是效率几乎一致,在少数情况下,LRU更容易体现出低的缺页率,可能是由于较为规律的访问顺序导致的.