上海大学

SHANGHAI UNIVERSITY

操作系统（二）实验三报告

组　　号	第 9 组
姓　　名	蔡卓悦
学　　号	18120482
实 验 序 号	实验三
日　　期	2020 年 12 月 25 日

一、 实验目的与要求

1 实验目的

近年来，由于大规模集成电路（LSI）和超大规模集成电路（VLSI）技术的发展，使存储器的容量不断扩大，价格大幅度下降。但从使用角度看，存储器的容量和成本总受到一定的限制。所以，提高存储器的效率始终是操作系统研究的重要课题之一。虚拟存储技术是用来扩大内存容量的一种重要方法。学生应独立地用高级语言编写几个常用的存储分配算法，并设计一个存储管理的模拟程序，对各种算法进行分析比较，评测其性能优劣，从而加深对 这些算法的了解。

2 实验要求

为了比较真实地模拟存储管理，可预先生成一个大致符合实际情况的指令地址流。然后模拟这样一种指令序列 的执行来计算和分析各种算法的访问命中率。

二、实验环境

本实验操作系统为 macOS 操作系统，使用电脑为 MacBook Pro，本实验的 IDE 是苹果官方的 Xcode 软件。

三、实验内容及其设计与实现

1、实验内容

本实验中我采用页式分配存储管理方案，并通过分析计算不同页面淘汰算法情况下的访问命中率来比较各种算法的优劣。

另外也考虑到改变页面大小和实际存储器容量对计算结果的影响，从而可为算则好的算法、合适的页面尺寸和实存容量提供依据。

随机生成一页面序列（在本实验中我们设为 P1，P2，....，Pn）并编程实现操作系统对于页面分配和置换的 OPT、FIFO、LRU 算法，同时计算三种算法的缺页率。（缺页率：假设页面序列长为 n，系统为进程分配 m 个物理块，如果缺页 n0 次，则缺页率为 n0/n×100%）。

对于同一算法，应比较分配的物理块数不同时的缺页率；不同的算法，应比较分配的物理块数相同时的缺页率。

本程序是按下述原则生成指令序列的：

（1） 50%的指令是顺序执行的。

（2） 25%的指令均匀散布在前地址部分。

（3） 25%的指令均匀散布在后地址部分。

示例中选用最佳淘汰算法（OPT）和最近最少使用页面淘汰算法（LRU）计算页面命中率。公式为：

$$命中率 = 1 - \frac{页面失效次数}{页地址流长度}$$

假定虚存容量为 32K，页面尺寸从 1K 至 8K，实存容量从 4 页至 32 页。

2、实验总体思路

整个实验程序框图如下图所示：

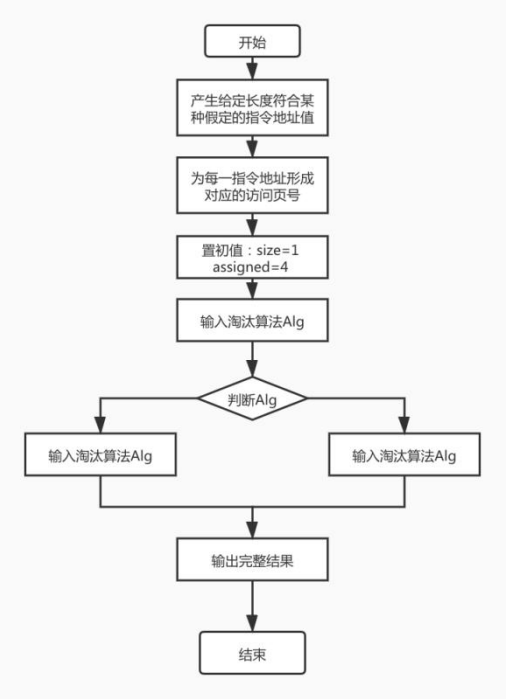

图 1 整体实验程序框图

3、具体设计和实现

(1) 生成指令序列

为各条指令分配满足实验要求的地址，将指令均分为 4 组，第一组随机分配指令地址，第一组指令分配的地址+1 作为第二组指令的地址，满足“50%的指令顺序执行”的条件；第三组指令随机分配前一半的地址，第四组指令随机分配后一半的地址，满足“各有 25%的指令均匀散布在前、后地址”的条件。

```
vector<short> A;
    A={1025,9999,15202,1028,1029,5555,6666,5555,6667,8888,9999,4458};
    srand(time(NULL));
    int N = 256000;
    for(int i = 0;i<N;i++)
    {
        int x;
        if(i <= N*0.25)
            x = rand()%20000;
        else if(i > N*0.25 && i < N*0.75)
            x = A[i-1]+1;
        else
            x = rand()%20000 + 32767-20000;
        A.push_back(x);
        if(debug)
        {
            printf("a[%03d]=",i);
            printf("%-10d",x);
            if((i+1)%4 == 0)
                cout << endl;
        }
    }
```

图 2 生成随机指令代码实现

（2）最佳淘汰算法（OPT）

最佳淘汰算法是一种理想的算法，可用来作为衡量其他算法优劣的依据，在实际系统中是难以实现的，因为它必须先知道指令的全部地址流。由于本实验中已预先生成了全部的指令地址流，故可计算出最佳命中率。

该算法的准则是淘汰已满页表中不再访问或是最迟访问的的页。这就要求将页表中的页逐个与后继指令访问的所有页比较，如后继指令不在访问该页，则把此页淘汰，不然得找出后继指令中最迟访问的页面淘汰。可见最佳淘汰算法要花费较长的运算时间。

最佳淘汰算法的程序框图如下所示：

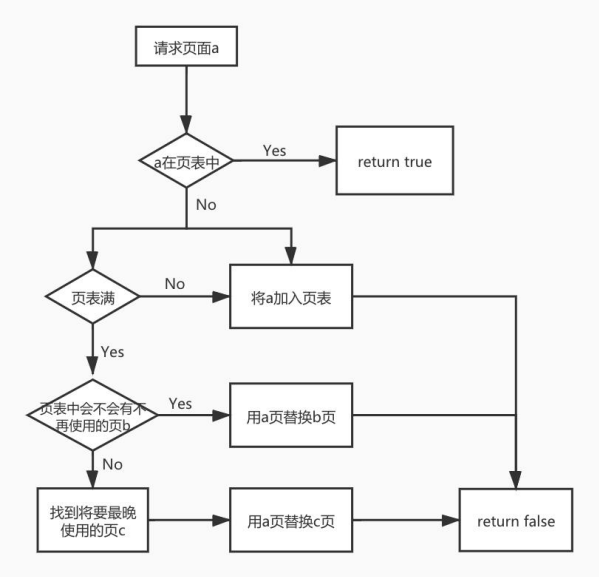

图 3 OPT 算法框图

OPT 算法的置换页面代码为：（完整的 OPT 算法实现代码见附件 1）

```
else//没有空余内存 要置换
                    {
                        if(debug)
                            cout << "置换\n";

                        int maxindex = i;
                        int position = 0;
                        for(int j = 0;j<memory.size();j++)//用j来找出要被换走的页面-OPT算法是找到
pageno中最晚出现的页
                        {
                            int k = find(pageno.begin()+i+1,pageno.end(),memory[j])-
pageno.begin();
                            //k表示找到memory[j]所表示页面的位置
                            if(k == pageno.end()-(pageno.begin()+i+1))
                            {
                                //表示memory[j]表示的页是画物理内存最后一页了
                                maxindex = k;
                                position = j;
                                break;
                            }
```

```
                else
                    //表示找到了memory[j]表示的页
                {
                    if(k > maxindex)//ii比当前的maxindex大 所以更新maxindex
                    {
                        maxindex = k;
                        position = j;
                    } } }
            memory[position] = pageno[i]; //置换完毕
        }
        miss++;//缺页次数+1
```

图 4 OPT 置换算法代码

(3) 先进先出页面置换算法 (FIFO)

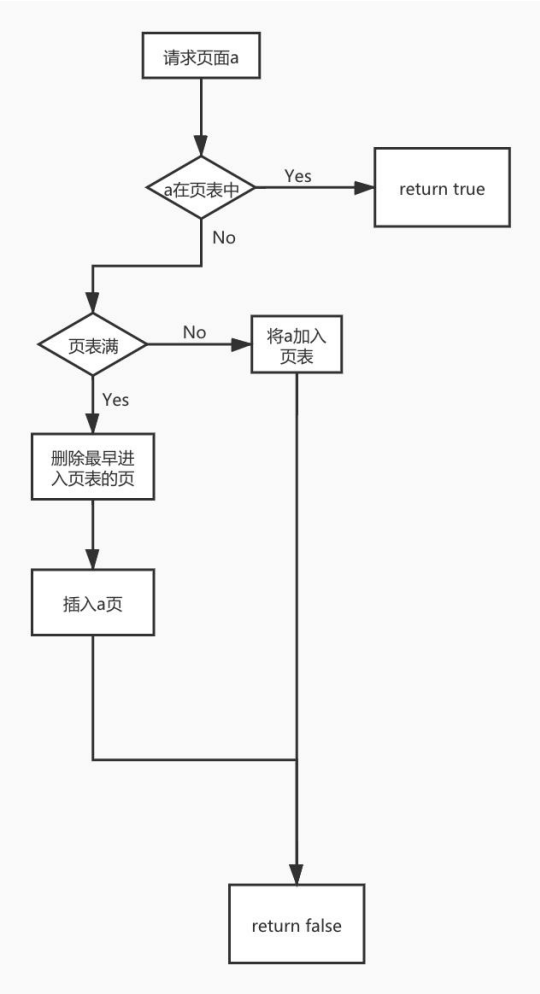

图 5 FIFO 先进先出算法框图

FIFO 算法的置换页面代码为：（完整的 FIFO 算法实现代码见附件 2）

```
else
                {
                    //cout << "缺页\n";
                    if(memory.size() < pageAssigned)
                        //有空闲页
                    {
                        memory.push_back(pageno[i]);
                        if(debug)
                            cout << "装入\n";
                    }
                    else
                    {
                        //置换
                        if(debug)
                            cout << "置换\n";
                        for(int k = 1;k<memory.size();k++)
                        {
                            //所有的页往前移动一个 空出最后一个位置给新页
                            memory[k-1] = memory[k];
                        }
                        memory[memory.size()-1] = pageno[i];
                    }
                    miss++;
                }
```

图 6 FIFO 置换算法代码

（4）最近最少使用页淘汰算法（LRU）

这是一种经常使用的方法，这种算法有各种不同的实施方案，这里采用的是不断调整页表链的方法，即总是淘汰页表链链首的页，而把新访问的页插入链尾。如果当前调用页已在页表内，则把它再次调整到链尾。

这样就能保证最近使用的页，总是处于靠近链尾部分，而不常使用的页就移到链首，逐个被淘汰，在页表较大时，调整页表链的代价也是不小的。

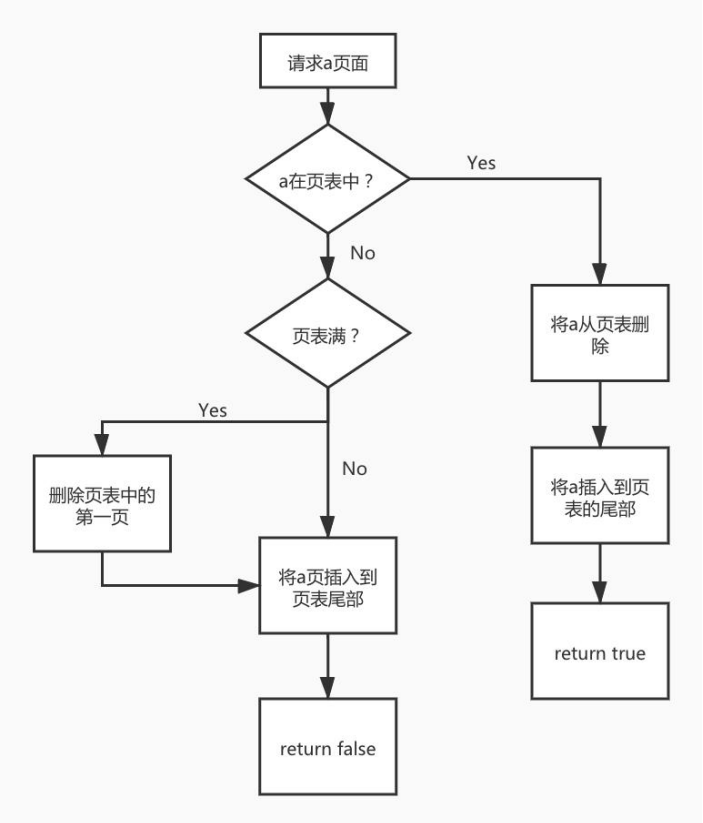

图 7 LRU 最近最少使用算法框图

LRU 算法需要系统的硬件的支持，为了了解一个进程在内存中各个页面各有多少的时间没有被进程访问，以及快速知道哪一页最近未被使用，需要有寄存器和栈两种硬件之一的支持。

（1）寄存器

为了记录进程在内存中各页使用，给内存中的页面配置一个移位寄存器，可以表示为：

$$R = R_{n-1}R_{n-2}...R_2R_1R_0$$

当进程访问某个物理块的时候将内存块对应的寄存器置位为 1.此时，定时信号会每隔一定时间将寄存器右移 1 位。如果我们把 n 位寄存器的数看作整体，那么具有最小数值的寄存器对应的页面就是最久没有使用的页面,在访问内存发生缺页的时候应该最早把它置换出去。

（2）栈

可以利用特殊的栈来表示各个页面的页面号。当进程访问页面的时候，将该页面的页面号从栈中移出，将它押入栈的顶部。因此栈顶总是最新被访问的页面号，而栈底是最近最久未使用的页面页面号。

在本次实验中，最开始我用的是比较偷懒的方式，就是用一个 vector<int>的 clock 数组来记录页面最近一次被访问/使用的时间。每一次操作后 clock++来表示时间的过去。

LRU 算法的实现代码为：（完整的 LRU 算法实现代码见附件 3）

（1）用 clock 数组实现

```
//用vector<int> clock数组实现记录最近一次使用时间

int index=find(memory.begin(),memory.end(),pageno[i])-memory.begin();
//求index的过程就是在物理内存中寻找对应页是否存在的过程
if(index != memory.end()-memory.begin())//没有缺页
{
  clocks[index] = clock;//clock用来记录每个内存所装的页的最近一次的使用时间
}
else
{
  if(memory.size() < pageAssigned)//有空闲页 可以直接装入
  {
    clocks.push_back(clock);//此时的时间装到最后空闲位置
  }
  else//置换页面
  {
    int position = 0;
    int minclocks = clocks[0];
    //找到最久未使用的页面
    for(int k = 1;k<clocks.size();k++)
    {
      if(clocks[k] < minclocks )
      {
        minclocks = clocks[k];
        position = k;
      } }
    memory[position] = pageno[i];
    clocks[position] = clock;
  }
  miss++;
}
clock++; //时钟
}
```

图 8 LRU 的 clock 数组实现代码

但是在验收过程中方老师指出，这种方法不属于书中的两种方法，并且这种方式也很难在计算机中用硬件来实现。

所以我课后重新编写代码实现了书本上的栈的方式，修改后的 LRU 实现代码如下：

```
//用书中 栈的方式实现LRU
int index=find(memory.begin(),memory.end(),pageno[i])-memory.begin();
                //求index的过程就是在物理内存中寻找对应页是否存在的过程

if(index != memory.end()-memory.begin())//命中时
{
  vector<int>::iterator pos=find(clocks_stack.begin(),clocks_stack.end(),pageno[i]);
  //对应页号在栈中的位置pos
  clocks_stack.erase(pos);//删除在栈里的对应页号
  clocks_stack.push_back(pageno[i]);//页号重新pageno[i]压到栈顶（vector<int>尾部）
}
else//未命中 缺页
{
  if(memory.size() < pageAssigned)//缺页 有空闲页 直接装入
{
  memory.push_back(pageno[i]);//把此页装到物理内存最后空闲位置
  clocks_stack.push_back(pageno[i]);//把页号压入栈
}
  else//没有空闲页 置换页面
  {
    int minclocks_stack=clocks_stack.front();//获取页号 栈底是最久未使用的页面
    clocks_stack.erase(clocks_stack.begin());//出栈最久未使用页面的页号
    vector<unsigned char>::iterator pos2=find(memory.begin(),memory.end(),minclocks_stack);
    //在memory中找到这个最久没用页面的位置
    pos2=memory.erase(pos2);//在memory中移出页面
    pos2=memory.insert(pos2,pageno[i]);//新页替换被移出的页的位置
    clocks_stack.push_back(pageno[i]);//新页的页号压入栈
  }
  miss++;
}
```

图 9 LRU 的栈的实现

四、实验结果

在本实验中，OPT、FIFO 和 LRU 算法展示出了不同的缺页率和命中率，具体的结果如下：

```
PAGE NUMBER WITH SIZE 1k FOR EACH ADDRESS IS:Optimal
PAGE NUMBER WITH SIZE 1k EACH ADDRESS IS:
4          5.220654E-01
PAGE NUMBER WITH SIZE 2k EACH ADDRESS IS:
6          3.640884E-01
PAGE NUMBER WITH SIZE 4k EACH ADDRESS IS:
8          2.225052E-01
PAGE NUMBER WITH SIZE 8k EACH ADDRESS IS:
10         1.265995E-01
PAGE NUMBER WITH SIZE 16k EACH ADDRESS IS:
12         8.563270E-02
PAGE NUMBER WITH SIZE 32k EACH ADDRESS IS:
14         5.756761E-02
PAGE NUMBER WITH SIZE 64k EACH ADDRESS IS:
16         3.404528E-02
PAGE NUMBER WITH SIZE 128k EACH ADDRESS IS:
18         1.480790E-02
PAGE NUMBER WITH SIZE 256k EACH ADDRESS IS:
20         1.249941E-04
PAGE NUMBER WITH SIZE 512k EACH ADDRESS IS:
22         1.249941E-04
PAGE NUMBER WITH SIZE 1024k EACH ADDRESS IS:
24         1.249941E-04
PAGE NUMBER WITH SIZE 2048k EACH ADDRESS IS:
26         1.249941E-04
PAGE NUMBER WITH SIZE 4096k EACH ADDRESS IS:
28         1.249941E-04
PAGE NUMBER WITH SIZE 8192k EACH ADDRESS IS:
30         1.249941E-04
PAGE NUMBER WITH SIZE 16384k EACH ADDRESS IS:
32         1.249941E-04
PAGE NUMBER WITH SIZE 32768k EACH ADDRESS IS:
34         1.249941E-04
```

图 10 OPT 算法运行结果

```
PAGE NUMBER WITH SIZE 1k FOR EACH ADDRESS IS:LF
PAGE NUMBER WITH SIZE 1k EACH ADDRESS IS:
4          7.875256E-01
PAGE NUMBER WITH SIZE 2k EACH ADDRESS IS:
6          6.814212E-01
PAGE NUMBER WITH SIZE 4k EACH ADDRESS IS:
8          5.545834E-01
PAGE NUMBER WITH SIZE 8k EACH ADDRESS IS:
10         3.044857E-01
PAGE NUMBER WITH SIZE 16k EACH ADDRESS IS:
12         1.979243E-01
PAGE NUMBER WITH SIZE 32k EACH ADDRESS IS:
14         1.475946E-01
PAGE NUMBER WITH SIZE 64k EACH ADDRESS IS:
16         9.810478E-02
PAGE NUMBER WITH SIZE 128k EACH ADDRESS IS:
18         4.813056E-02
PAGE NUMBER WITH SIZE 256k EACH ADDRESS IS:
20         2.109276E-04
PAGE NUMBER WITH SIZE 512k EACH ADDRESS IS:
22         1.601487E-04
PAGE NUMBER WITH SIZE 1024k EACH ADDRESS IS:
24         1.289002E-04
PAGE NUMBER WITH SIZE 2048k EACH ADDRESS IS:
26         1.249941E-04
PAGE NUMBER WITH SIZE 4096k EACH ADDRESS IS:
28         1.249941E-04
PAGE NUMBER WITH SIZE 8192k EACH ADDRESS IS:
30         1.249941E-04
PAGE NUMBER WITH SIZE 16384k EACH ADDRESS IS:
32         1.249941E-04
PAGE NUMBER WITH SIZE 32768k EACH ADDRESS IS:
34         1.249941E-04
```

图 11 LRU 算法运行结果

```
PAGE NUMBER WITH SIZE 1k FOR EACH ADDRESS IS:FIFO
PAGE NUMBER WITH SIZE 1k EACH ADDRESS IS:
4          8.166297E-01
PAGE NUMBER WITH SIZE 2k EACH ADDRESS IS:
6          7.104003E-01
PAGE NUMBER WITH SIZE 4k EACH ADDRESS IS:
8          5.471775E-01
PAGE NUMBER WITH SIZE 8k EACH ADDRESS IS:
10         2.682101E-01
PAGE NUMBER WITH SIZE 16k EACH ADDRESS IS:
12         1.987329E-01
PAGE NUMBER WITH SIZE 32k EACH ADDRESS IS:
14         1.485009E-01
PAGE NUMBER WITH SIZE 64k EACH ADDRESS IS:
16         9.810478E-02
PAGE NUMBER WITH SIZE 128k EACH ADDRESS IS:
18         4.933362E-02
PAGE NUMBER WITH SIZE 256k EACH ADDRESS IS:
20         2.773308E-04
PAGE NUMBER WITH SIZE 512k EACH ADDRESS IS:
22         2.187397E-04
PAGE NUMBER WITH SIZE 1024k EACH ADDRESS IS:
24         1.523366E-04
PAGE NUMBER WITH SIZE 2048k EACH ADDRESS IS:
26         1.406184E-04
PAGE NUMBER WITH SIZE 4096k EACH ADDRESS IS:
28         1.289002E-04
PAGE NUMBER WITH SIZE 8192k EACH ADDRESS IS:
30         1.249941E-04
PAGE NUMBER WITH SIZE 16384k EACH ADDRESS IS:
32         1.249941E-04
PAGE NUMBER WITH SIZE 32768k EACH ADDRESS IS:
34         1.249941E-04
```

图 12 FIFO 算法运行结果

通过分析上面的三张运行结果图片我们可以发现，不管是三种算法中的哪一种，缺页率都会随着块数的增加而下降，并且缺页率下降的速度会越来越慢。

同时我发现，OPT 算法的缺页率在任何条件下（物理块数是多少）都小于 LRU 算法和 FIFO 算法的缺页率。由此观察我们可以得出结论，OPT 算法在本实验中是表现和性能最好的算法。从课上学习和课后研讨我们都知道，尽管 OPT 算法的表现非常好，这种算法从本质上来说是一种理想化的算法，在现实生活中很难实现。

五、收获与体会

本次实验三是老师重点验收的第一个实验，也是本学期操作系统 2 课程中我们最重要的实验之一。

在该实验中，我利用 c++语言编写了代码，我结合操作系统关于分页存储和页表查询的知识，用代码实现并比较了三种不同的页面淘汰算法的特性。同时我们还根据指导书的要求，利用随机生成的方式模拟真正情况下的操作系统中可能出现的情况，使得实验更加准确并且更真实地反应了三种算法的优劣和特点。

这个实验不仅加深了我们对于操作系统中分页存储、页面分配和置换等概念的理解，还巩固了我的 c++语言编程技巧和提升了我对于 c++的 STL 库的掌握，让我们收获非常大。