北京郵電大學

Beijing University of Posts and Telecommunications

实 验 报 告

课 程 名 称	操作系统
题 目	内存管理实验
姓 名	沈原灏
学 号	2021211108
班 级	2021211306
指 导 教 师	李文生

2023 年 11 月

一、概览

1.1 实验目的

基于 openEuler 内核的操作系统，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

1.2.1 生成内存访问串

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。

比如：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 mod 100 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 10（可以自己定义）条指令，将指令地址映射到页面，生成内存访问串。

1.2.2 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

- OpenEuler 20.03(LTS) (由于虚拟机不太好编辑代码，采用了华为云的 ECS 服务器完成)
- MobaXterm：一款远程桌面管理软件，支持 ssh 远程连接服务器
- gcc 7.3.0

二、实验设计

2.1 数据结构

数据结构如下，模拟了内存帧等数据结构，并记录了随机指令序列和对应的页面序列。其中 frameUsage 专门为 LRU 算法而设计。

```
#define FRAME_SIZE 6     // 帧数
#define PAGE_SIZE 10     // 页大小
#define SEQUENCE_COUNT 100 // 指令序列长度

int frames[FRAME_SIZE];            // 模拟内存
int frameUsage[FRAME_SIZE];          // 记录帧未使用次数
int instructionSequence[SEQUENCE_COUNT]; // 存储指令序列
int pageSequence[SEQUENCE_COUNT];      // 存储页面序列
```

2.2 随机指令与页面序列生成

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。

比如：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 mod 100 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 10（可以自己定义）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：

```
// 生成随机指令序列
void generateInstructionSequence()
{
   srand(time(NULL)); // seed
   for (int i = 0; i < SEQUENCE_COUNT; i++)
   {
      int random = rand() % 100;
      if (random < 70)
      {
         // 70% 顺序
         instructionSequence[i] = i;
      }
      else if (random < 80)
      {
         // 10% 前地址
         instructionSequence[i] = (i > 0) ? rand() % i : 0;
      }
      else
      {
         // 20% 后地址
         instructionSequence[i] = i + rand() % (SEQUENCE_COUNT - i);
      }
   }
```

```
}

// 生成页面序号序列
void generatePageSequence()
{
   for (int i = 0; i < SEQUENCE_COUNT; i++)
   {
      pageSequence[i] = instructionSequence[i] / PAGE_SIZE;
   }
}
```

2.3 算法流程概述

算法设计分为主模拟函数 simulatePageReplacement，以及 fifoPageReplacement、lruPageReplacement、optimalPageReplacement 三个替换算法函数。

2.3.1 主模拟函数

simulatePageReplacement 函数是主要用于模拟页面置换算法的函数。它接受一个页面置换算法的函数指针作为参数，然后使用该算法模拟指令序列的执行过程，并输出页面错误数和页面错误率。

```
// 主模拟程序
void simulatePageReplacement(int (*pageReplacementAlgorithm)(int *,
int, int *))
{
   int pageFaults = 0;

   // 初始化
   memset(frames, -1, sizeof(frames));
   memset(frameUsage, 0, sizeof(frameUsage));

   for (int i = 0; i < SEQUENCE_COUNT; i++)
   {
      int pageNumber = pageSequence[i];

      // 检查页面是否已经在帧内
      int pageInMemoryIndex = findPageInMemory(pageNumber);

      if (pageInMemoryIndex == -1)
      {
```

```
            // 页错误
            pageFaults++;

            // 利用算法寻找替换帧
            int pageIndexToReplace = pageReplacementAlgorithm(frames, i,
pageSequence);

            // 替换，并将未使用次数清零
            frames[pageIndexToReplace] = pageNumber;
            frameUsage[pageIndexToReplace] = 0;
        }
        else
        {
            // 将找到的页面，未使用次数清零
            frameUsage[pageInMemoryIndex] = 0;
        }
    }

    // 计算
    printf("Page Faults: %d\n", pageFaults);
    printf("Page Fault Rate: %.2f%%\n", (float)pageFaults /
SEQUENCE_COUNT * 100);
}
```

2.3.1 LRU 算法

采用 frameUsage 数组记录页表中每个条目的使用时间，每次引用一个页或者置换一个页时，将它对应的计数器设为 0，将其他页的计数器自增 1。

算法核心代码如下：

```
// LRU
int lruPageReplacement(int *frames, int currentIndex, int
*pageSequence)
{
    int pageIndexToReplace = 0;

    int max_count = -1;

    for (int i = 0; i < FRAME_SIZE; i++)
    {
        if (max_count < frameUsage[i])
```

```
    {
      max_count = frameUsage[i];
      pageIndexToReplace = i;
    }
  }

  for (int i = 0; i < FRAME_SIZE; i++)
  {
    frameUsage[i]++;
  }

  return pageIndexToReplace;
}
```

2.4 FIFO 算法

实现比较简单。维护一个 index 决定要替换的页面，并且将 index 自增 1，再对 FrameSize 进行模运算防止越界。

算法的核心代码如下：

```
// FIFO
int fifoPageReplacement(int *frames, int currentIndex, int
*pageSequence)
{
  static int fifoIndex = 0;

  int pageIndexToReplace = fifoIndex;
  fifoIndex = (fifoIndex + 1) % FRAME_SIZE;

  return pageIndexToReplace;
}
```

2.5 OPT 算法

实现比较复杂，其间遇到了许多困难。主要思路是模拟未来的指令序列，找到在未来最长时间内不再被使用的页面，并返回该页面的索引。

算法核心代码如下：

```
// 最优页面置换
int optimalPageReplacement(int *frames, int currentIndex, int
*pageSequence)
```

```
{
  int farthestIndex = -1, farthestDistance = -1;

  for (int i = 0; i < FRAME_SIZE; i++)
  {
    int nextPageIndex = currentIndex + 1;
    int j;
    for (j = nextPageIndex; j < SEQUENCE_COUNT; j++)
    {
      if (frames[i] == pageSequence[j])
      {
        if (j > farthestDistance)
        {
          farthestDistance = j;
          farthestIndex = i;
        }
        break;
      }
    }
    if (j == SEQUENCE_COUNT)
      return i;
  }
  return farthestIndex;
}
```

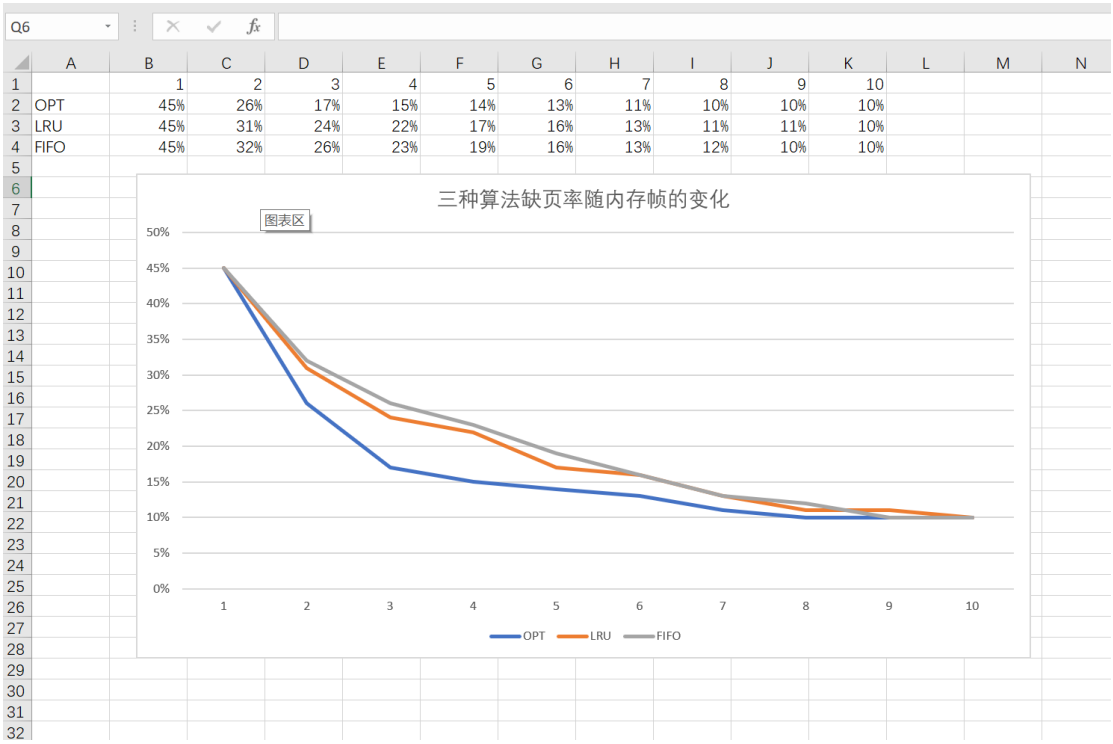
三、测试结果

3.1 使用方法

在 openEuler 环境下，使用 **gcc -o lab5 lab5.c** 命令进行编译，随后使用 **./lab5** 命令即可运行。

3.2 结果分析

修改帧数，用 Excel 记录每次的测试结果，如下：



	A	B	C	D	E	F	G	H	I	J	K
1		1	2	3	4	5	6	7	8	9	10
2	OPT	45%	26%	17%	15%	14%	13%	11%	10%	10%	10%
3	LRU	45%	31%	24%	22%	17%	16%	13%	11%	11%	10%
4	FIFO	45%	32%	26%	23%	19%	16%	13%	12%	10%	10%

可以看出，当可用帧数量过少或者过多时，三种算法的效率趋于一致。可以发现 OPT 算法的缺页错误率明显低于其他两种，FIFO 算法和 LRU 算法的缺页错误率接近，但是大部分情况下 LRU 的缺页错误率更低，LRU 算法没有表现出比 FIFO 算法优更多的性能，推测是由于生成的数据局部性不够强导致。

可以发现随着页表大小的增加，缺页率都呈下降趋势，最优置换算法的下降最快，LRU 次之，FIFO 最慢。由于数据序列的随机性，算法之间的优势不是绝对的，但随着帧数量增大（或减少）到一定程度后，算法间的差异对缺页率产生的影响也逐渐降低。

四、总结

本次实验中我采用 C 语言实现了三种页面置换算法，并且对它们的性能进行比较，让我对于 openEuler-C 编程的特性更加了解，对相关知识点的掌握更加牢固。通过实验，我了解到三种算法性能排序如下：最优置换算法 OPT > 最近最少使用算法 LRU > 先进先出算法 FIFO。

本次实验时长约为 4h，主要时间在 debug 和撰写实验报告，其中 LRU 算法一开始忘记在未缺页的情况下统计未使用次数，导致退化成了 FIFO 算法，不过也借由此让我对 LRU 算法的了解更加深刻。