BUPT

内存控制

实验报告

姓名：陈朴炎　　学号：2021211138

2023-12-3

目录

1 实验内容

1.1 实验目的

基于 openEuler 或其他 Linux 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

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 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 性能比较

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

1.3 实验要求

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。

2 环境说明

本次实验在 Windows11 的 Linux 子系统 Ubuntu22.04.2 LTS（GNU/Linux 5.10.16.3-microsoft-standard-WSL2 x86_64）下完成。

3 程序设计

3.1 全局变量定义

```
#define TOTAL_INSTRUCTIONS 100
#define PAGE_SIZE 10
#define PAGE_COUNT 10
int pageTableSize = PAGE_TABLE_SIZE;
```

在本次实验中，我定义了四个全局变量/宏，其中，变量名和含义如下：

(1) TOTAL_INSTRUCTIONS：代表总共的指令个数，这里我设置成 100。

(2) PAGE_SIZE：代表一个页面能够存放多少个指令，这里我设置为 10。

(3) PAGE_COUNT：代表一共有几个页面，这里我设置为 10。

(4) pageTableSize：代表页表一共有几个表项，这里我设置成 4 个。

3.2 函数定义

3.2.1 generateAddressPages(int instructions[ ], int pages[ ], int seed)

这个函数在实验要求的基础上又增加了一点点，由于我的主函数是需要不断迭代的，所以有一个 seed 种子参数，用来辅助随机生成序列。

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 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.    void generateAddressPages(int instructions[], int pages[]) {
2.     srand(time(NULL));
3.     int currentAddress = 0;
4.     for (int i = 0; i < TOTAL_INSTRUCTIONS; i++) {
5.         double probability = ((double)rand() / RAND_MAX) * 100;
6.         if (probability < 70) {
7.             // 70%的概率选择顺序执行
8.             currentAddress++;
9.         } else if (probability < 80) {
10.            // 10%的概率选择在前地址部分均匀分布
11.            currentAddress = rand() % (i == 0 ? 1 : i);
12.        } else {
13.            // 20%的概率选择在后地址部分均匀分布
14.            currentAddress = (rand() % (TOTAL_INSTRUCTIONS - i + 1))
   + currentAddress + 1;
15.        }
16.        instructions[i] = currentAddress;
17.    }
18.    for(int i = 0; i < TOTAL_INSTRUCTIONS; i++){
19.        pages[i] = instructions[i] / PAGE_SIZE;
20.    }
21. }
```

3.2.2 double fifo(int pages[ ])

该函数用 FIFO 页面置换算法维护一个有限大小的页表来模拟内存管理。遍历指令地址流时，如果访问的页面不在页表中，表示发生了缺页，此时将新页面添加到页表中。当页表达到最大容量时，采用先进先出的策略淘汰最早进入页表的页面。通过统计缺页次数并计算缺页率。

```
1.    // 先进先出页面置换
```

```
double fifo(int pages[]){
    int pageTable[PAGE_TABLE_SIZE];
    memset(pageTable, -1, sizeof(pageTable));
    int index = 0;
    int countPageFault = 0;
    for(int i = 0; i < TOTAL_INSTRUCTIONS; i++){
        // 模拟访问页表--缺页
        if(findPage(pageTable, pages[i]) == -1){
            // 没找着--缺页
            countPageFault++;
            pageTable[index] = pages[i];
            index = (index + 1) % PAGE_TABLE_SIZE;
        }
    }
    double faultRate = 100*(double)countPageFault/(double)TOTAL_INSTRUCTIONS;
    printf("FIFO 页面置换的缺页数为: %d, 缺页率是:%.2lf%%\n",countPageFault, faultRate);
    return faultRate;
}
```

3.2.3 double lru(int pages[ ])

该代码段实现了最近最少使用（LRU）页面置换算法。通过维护一个有限大小的页表和一个记录每个页最近访问时间的 LRU 时间数组，在遍历指令地址流的过程中，检查页表是否命中。若未命中，则选择 LRU 时间最长的页进行淘汰，并将新页面插入页表。同时，更新 LRU 时间数组，标记刚刚访问的页面时间为 0。最后，计算缺页次数并根据总指令数计算出缺页率，以评估 LRU 算法在模拟环境下的性能。

```
// LRU 页面置换
double lru(int pages[]){
    int lruTime[PAGE_TABLE_SIZE], pageTable[PAGE_TABLE_SIZE];
    memset(pageTable, -1, sizeof(pageTable));
    int lruPageFaultCount = 0;
    for(int i = 0; i < PAGE_TABLE_SIZE; i++){
        lruTime[i] = 100;
```

```
    }
    for(int i = 0 ; i < TOTAL_INSTRUCTIONS; i++){
        if(findPage(pageTable, pages[i]) == -1){
            // 没找着
            int victimIndex = findLruMax(lruTime);
            pageTable[victimIndex] = pages[i];
            lruTime[victimIndex] = 0;
            lruPageFaultCount++;
        }else{
            // 找着了
            int index = findPage(pageTable, pages[i]);
            lruTime[index] = 0;
        }
        flushLruTime(lruTime);
    }
    double faultRate = 100*(double)lruPageFaultCount/(double)TOTAL_IN
STRUCTIONS;
    printf("LRU 页面置换算法的缺页次数是：%d, 缺页率
是:%.2lf%%\n", lruPageFaultCount, faultRate);
}
```

3.2.4 double opt(int pages[ ])

该函数实现了最优（OPT）页面置换算法。通过维护一个有限大小的页表，代码在每次产生缺页时使用 findOPT 函数找到最佳的淘汰页，并将新的页面插入页表。最后，它计算了缺页次数，并根据总指令数计算出了缺页率。这样可以评估 OPT 算法在模拟环境中的性能表现。

```
// OPT 置换，输出缺页信息
double opt(int pages[]){
    int pageTable[PAGE_TABLE_SIZE];
    int optCount = 0;
    memset(pageTable, -1, sizeof(pageTable));
    for(int i = 0; i < TOTAL_INSTRUCTIONS; i++){
        if(findPage(pageTable, pages[i]) == -1){
            int victimIndex = findOPT(pageTable, pages, i);
            pageTable[victimIndex] = pages[i];
            optCount ++;
        }
```

```
    }
    double faultRate = 100*(double)optCount/TOTAL_INSTRUCTIONS;
    printf("OPT 页面置换算法的缺页次数是：%d，缺页率是:%.2lf%%\n", optCount, faultRate);
    return faultRate;
}
```

3.2.5 int findPage(int pageTable[ ], int pageNum)

该函数通过遍历页表来查找有没有 pageNum 的表项，如果有，则返回索引值，如果没有，则返回-1。

```
// 在页表查看是否缺页
int findPage(int pageTable[], int pageNum){
    for(int i = 0 ; i < PAGE_TABLE_SIZE; i++){
        if(pageTable[i] == pageNum){
            return i;
        }
    }
    return -1;
}
```

3.2.6 int findOPT(int pageTable[ ], int pages[ ], int currentIndex)

该函数 findOPT 用于在 OPT（最优）页面置换算法中找到最佳淘汰页的下标。它接收当前页表、指令地址序列以及当前指令地址在序列中的下标作为参数。函数的执行步骤如下：

（1）对于每个页表中的页面，计算该页面在未来指令序列中下一次被访问的距离。这是通过从当前指令地址的下一位置开始搜索找到相应的页面来实现的。

（2）将计算得到的距离存储在名为 distance 的数组中。

（3）在 distance 数组中找到最大的距离，并记录下标。这表示在未来最长时间内不会被访问到的页面，即最佳淘汰页。

（4）返回最佳淘汰页的下标。

该函数的目的是为了在 OPT 页面置换算法中选择最优的淘汰页，以最小化未来缺页的可能性。

```
// 找到 OPT 要置换的那个元素下标，并返回
int findOPT(int pageTable[], int pages[], int currentIndex){
    // 在页表中，找到每个元素下一次要访问的距离
    int distance[PAGE_TABLE_SIZE];
    for(int i = 0 ; i < PAGE_TABLE_SIZE; i++){
        int j;
        for(j = currentIndex+1; j < TOTAL_INSTRUCTIONS; j++){
            // 找到了
            if(pageTable[i] == pages[j])
                break;
        }
        distance[i] = j - currentIndex;
    }
    // 比较距离，选择最大的
    int maxDistance = 0, retIndex = 0;
    for(int i = 0 ; i < PAGE_TABLE_SIZE; i++){
        if(maxDistance < distance[i]){
            maxDistance = distance[i];
            retIndex = i;
        }
    }
    return retIndex;
}
```

3.2.7 flushLruTime(int lruTime[ ])

该函数 `flushLruTime` 用于在 LRU（最近最少使用）页面置换算法中更新各个页面的访问时间。LRU 算法通过记录每个页面最近一次被访问的时间来判断哪个页面最久未被使用。

函数接收一个名为 `lruTime` 的数组作为参数，该数组存储了各个页面的最近一次访问时间。

对于数组中的每个元素（对应一个页面），将其值加一，表示时间的流逝。

该函数的目的是模拟时间的流逝，确保在每次页面被访问时，相关页面的访问时间被更新。通过不断刷新页面的访问时间，LRU 算法能够始终保持对最近使用过的页面的跟踪。

```
    // 刷新 lru 的时间
void flushLruTime(int lruTime[]){
    for(int i = 0;i < PAGE_TABLE_SIZE; i++){
        lruTime[i]++;
    }
}
```

3.2.8 findLruMax(int lruTime[ ])

该函数 findLruMax 用于在 LRU（最近最少使用）页面置换算法中找到具有最大访问时间的页面。LRU 算法根据页面的最近使用情况进行页面替换，因此需要找到最长时间未被访问的页面。

函数接收一个名为 lruTime 的数组作为参数，该数组存储了各个页面的最近一次访问时间。函数的执行步骤如下：

（1）初始化 max 变量为-1，retIndex 变量为 0。它们用于记录最大访问时间和对应的页面索引。

（2）遍历数组中的每个元素（对应一个页面的访问时间），比较其访问时间是否大于 max。

（3）如果某个页面的访问时间大于 max，则更新 max 和 retIndex 为当前页面的访问时间和索引。

（4）返回具有最大访问时间的页面索引 retIndex。

该函数的目的是在 LRU 算法中找到当前时间段内最长时间未被访问的页面，以便在页面置换时选择该页面进行替换。

```
// 找到 lru 时间最大的那个表项
int findLruMax(int lruTime[]){
    int max = -1;
    int retIndex = 0;
    for(int i = 0;i < PAGE_TABLE_SIZE; i++){
        if(lruTime[i] > max){
            max = lruTime[i];
            retIndex = i;
        }
    }
    return retIndex;
}
```

3.2.9 主函数

用户可以输入期望进行的实验次数，程序将在每次实验中生成新的指令地址序列和页号序列，然后通过调用先进先出（FIFO）、最近最少使用（LRU）和最优（OPT）页面置换算法来计算每种算法的缺页率。最终，程序输出了经过多次实验后各算法的平均缺页率，为本次实验提供了对页面置换算法性能的综合评估。

在平均测试之后，还会测试 Belady 现象，方法是每次通过相同的页表流，先测试页表大小为 pageTableSize 的情况，再测试页表大小为 pageTableSize+1 的情况。

```
int main() {
    int instructions[TOTAL_INSTRUCTIONS], pages[TOTAL_INSTRUCTIONS];

    double fifoSum = 0, lruSum = 0, optSum = 0;
    int total;
    printf("请输入想要实验的次数: ");
    scanf("%d", &total);
    printf("请输入想要测试的页表大小: ");
    scanf("%d", &pageTableSize);
```

```
    for(int i = 0; i < total; i++){
        printf("第 %d 次：\n", i+1);
        generateAddressPages(instructions, pages, i);
        double fifoRate = fifo(pages);
        double lruRate = lru(pages);
        double optRate = opt(pages);
        fifoSum += fifoRate;
        lruSum += lruRate;
        optSum += optRate;
    }
    double avgFifoRate = fifoSum / total ;
    double avgLruRate = lruSum / total;
    double avgOptRate = optSum / total;

    printf("经过 %d 实验\nFIFO 的缺页率平均值为:\t%.2lf\nLRU 的缺页率平均值为:\t%.2lf\nOPT 的缺页率平均值
为:\t%.2lf\n",total, avgFifoRate, avgLruRate, avgOptRate);
    printf("请输入想要测试页表的大小，我们将测试该页表以及比它大 1 的页表大小: ");
    scanf("%d", &pageTableSize);
    for(int i = 0 ;i < total ; i++){
        printf("============第 %d 次==========\n", i+1);
        generateAddressPages(instructions, pages, i);
        printf("页表大小为：%d\n", pageTableSize);
        fifo(pages);
        lru(pages);
        opt(pages);
        pageTableSize++;
        printf("页表大小为：%d\n", pageTableSize);
        fifo(pages);
        lru(pages);
        opt(pages);
        pageTableSize--;
    }

    return 0;
}
```

4 测试报告

4.1 比较三种性能页面置换算法的性能

为了能够比较直观且客观的比较三种页面算法的性能，我采取了多次实验，每次实验都测试多次三种算法的缺页率，再取它们的平均来比较。

4.1.1 测试用例 1

第一次实验采用方法是：做 10 次循环实验，打印出每次的缺页率，并取平均值，结果如下：

```
请输入想要实验的次数：10
请输入想要测试的页表大小：3
第 1 次：
FIFO页面置换的缺页数为：32，缺页率是:32.00%
LRU 页面置换算法的缺页次数是：33，缺页率是:33.00%
OPT 页面置换算法的缺页次数是：24，缺页率是:24.00%
第 2 次：
FIFO页面置换的缺页数为：28，缺页率是:28.00%
LRU 页面置换算法的缺页次数是：27，缺页率是:27.00%
OPT 页面置换算法的缺页次数是：20，缺页率是:20.00%
第 3 次：
FIFO页面置换的缺页数为：28，缺页率是:28.00%
LRU 页面置换算法的缺页次数是：27，缺页率是:27.00%
OPT 页面置换算法的缺页次数是：22，缺页率是:22.00%
第 4 次：
FIFO页面置换的缺页数为：27，缺页率是:27.00%
LRU 页面置换算法的缺页次数是：27，缺页率是:27.00%
OPT 页面置换算法的缺页次数是：21，缺页率是:21.00%
第 5 次：
FIFO页面置换的缺页数为：33，缺页率是:33.00%
LRU 页面置换算法的缺页次数是：33，缺页率是:33.00%
OPT 页面置换算法的缺页次数是：27，缺页率是:27.00%
```

图 4-1 测试 1 1-5 执行结果图

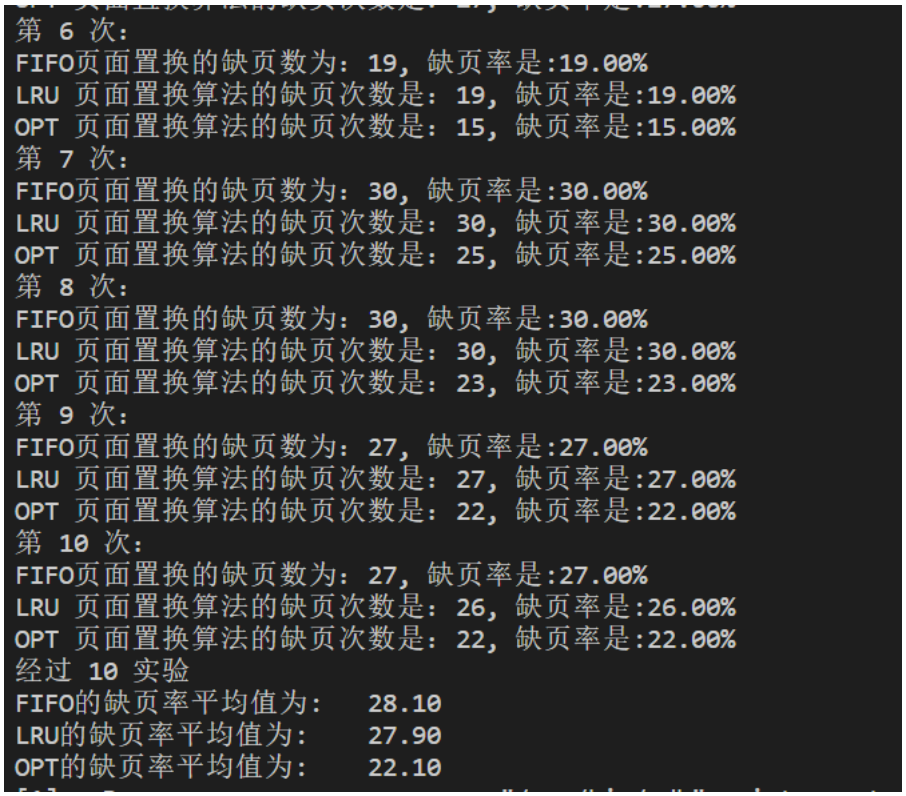

图 4-2 测试 1 6-10 执行结果图

分析:

(1) FIFO 页面置换算法:

缺页次数和缺页率: 缺页次数在每次实验中都在 27 到 33 之间波动, 平均缺页率为 28.10%。

(2) LRU 页面置换算法:

缺页次数和缺页率: 缺页次数在每次实验中都在 26 到 33 之间波动, 平均缺页率为 27.90%。LRU 考虑了页面的使用历史, 通常在某些场景下比 FIFO 效果好。

(3) OPT 页面置换算法:

缺页次数和缺页率: 缺页次数在每次实验中都在 15 到 27 之间波动, 平均缺页率为 22.10%。OPT 算法是理论上的最优算法, 但实际上很难实现, 因为

需要预知未来的页面访问情况。在实验中，它的效果相对较好，但实际应用中难以应用。

（4）平均缺页率比较：

在这次实验中，OPT 的平均缺页率最低，说明在这些场景下 OPT 的效果较好。LRU 和 FIFO 的平均缺页率相差不大，LRU 相对稍微好一些。

4.1.2 测试用例 2

第二次实验的方法是：采用 10 次循环实验，页表长度为 4，每次都打印出结果。相应的执行结果如下：

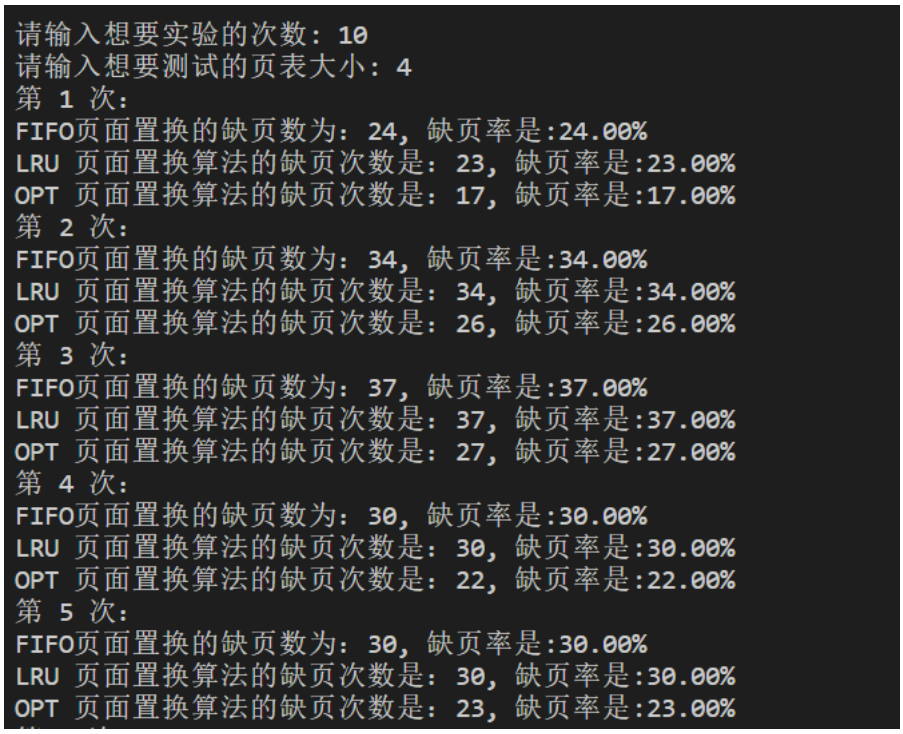

图 4-3 测试 2 1-5 执行结果图

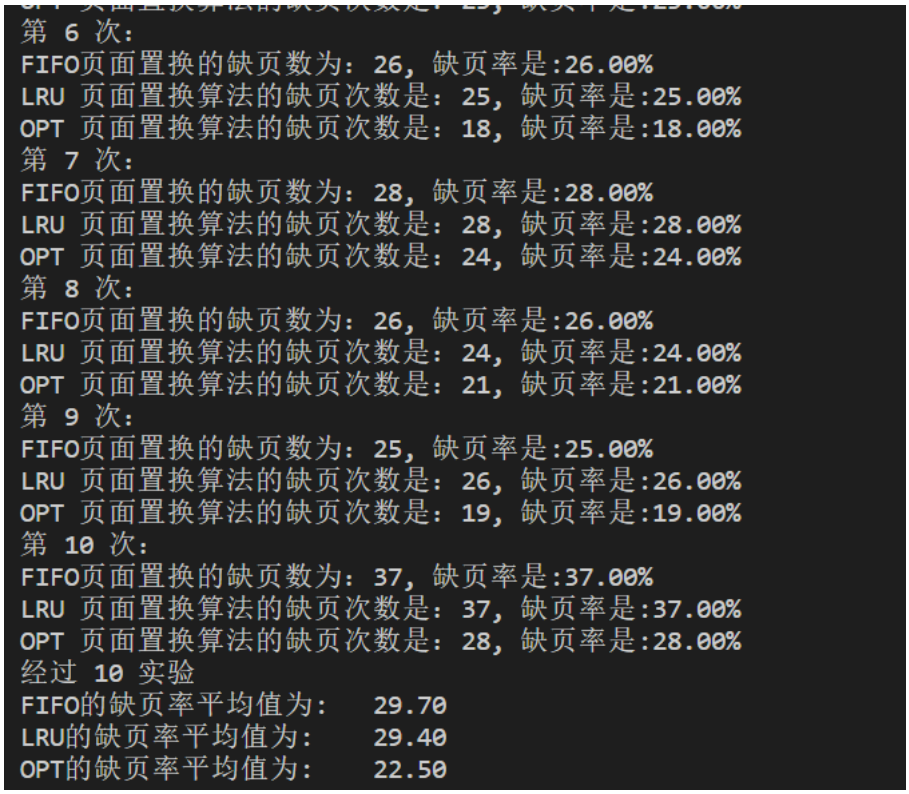

图 4-4 测试 2 6-10 执行结果图

这次实验，除了比较它们的平均缺页率，我还关注了每次循环的缺页率。

（1）FIFO 页面置换算法：

缺页次数和缺页率：每次实验的缺页率在 24% 到 37% 之间波动，平均缺页率为 29.70%。基本每次缺页率都和 LRU 算法持平或更高，但是第九次实验 FIFO 的缺页率比 LRU 的缺页率低，说明 LRU 在某些时候并不是性能最差的算法。

（2）LRU 页面置换算法：

缺页次数和缺页率：每次实验的缺页率在 23% 到 37% 之间波动，平均缺页率为 29.40%。LRU 算法的缺页率稍微比 FIFO 算法的缺页率好一些，但是在第九次实验中，LRU 算法缺页率比 FIFO 算法高，说明在一些场景 LRU 的算法性能并不一定比 FIFO 算法性能优。

（3）OPT 页面置换算法：

缺页次数和缺页率：每次实验的缺页率在 17% 到 28% 之间波动，平均缺页率为 22.50%。OPT 在每次实验中表现相对稳定，并且整体来说，它的平均缺页率较低。OPT 的优势在于理论上能够达到最佳的页面置换效果。

（4）平均缺页率比较：

与上一次实验相比，这次实验的每次实验的缺页率波动幅度相对较大。OPT 仍然是平均缺页率最低的算法，而 FIO 和 LRU 的缺页率仍然相对较高。

4.2 测试三种算法的 Belady 异常

在主函数中添加下面代码，测试 Belady 异常：

```
1.  printf("经过 %d 实验\nFIFO 的缺页率平均值为:\t%.2lf\nLRU 的缺页率平均值为:\t%.2lf\nOPT 的缺页率平均值为:\t%.2lf\n",total, avgFifoRate, avgLruRate, avgOptRate);
2.  printf("请输入想要测试页表的大小，我们将测试该页表以及比它大 1 的页表大小： ");
3.  scanf("%d", &pageTableSize);
4.  for(int i = 0 ;i < total ; i++){
5.      printf("============第 %d 次==========\n", i+1);
6.      generateAddressPages(instructions, pages, i);
7.      printf("页表大小为：%d\n", pageTableSize);
8.      fifo(pages);
9.      lru(pages);
10.     opt(pages);
11.     pageTableSize++;
12.     printf("页表大小为：%d\n", pageTableSize);
13.     fifo(pages);
14.     lru(pages);
15.     opt(pages);
16.     pageTableSize--;
17. }
```

4.2.1 测试用例 1

该次测试采用循环 10 次，每次测试的页表大小分别为 3 和 4。

```
请输入想要测试页表的大小，我们将测试该页表以及比它大1的页表大小：3
============第 1 次==========
页表大小为：3
FIFO页面置换的缺页数为：33，缺页率是:33.00%
LRU 页面置换算法的缺页次数是：33，缺页率是:33.00%
OPT 页面置换算法的缺页次数是：27，缺页率是:27.00%
页表大小为：4
FIFO页面置换的缺页数为：33，缺页率是:33.00%
LRU 页面置换算法的缺页次数是：33，缺页率是:33.00%
OPT 页面置换算法的缺页次数是：25，缺页率是:25.00%
============第 2 次==========
页表大小为：3
FIFO页面置换的缺页数为：33，缺页率是:33.00%
LRU 页面置换算法的缺页次数是：31，缺页率是:31.00%
OPT 页面置换算法的缺页次数是：23，缺页率是:23.00%
页表大小为：4
FIFO页面置换的缺页数为：30，缺页率是:30.00%
LRU 页面置换算法的缺页次数是：29，缺页率是:29.00%
OPT 页面置换算法的缺页次数是：20，缺页率是:20.00%
============第 3 次==========
页表大小为：3
FIFO页面置换的缺页数为：28，缺页率是:28.00%
LRU 页面置换算法的缺页次数是：28，缺页率是:28.00%
OPT 页面置换算法的缺页次数是：21，缺页率是:21.00%
页表大小为：4
FIFO页面置换的缺页数为：25，缺页率是:25.00%
LRU 页面置换算法的缺页次数是：24，缺页率是:24.00%
OPT 页面置换算法的缺页次数是：20，缺页率是:20.00%
```

图 4-5 Belady 测试用例 1 1-3 执行结果图

```
============第 4 次==========
页表大小为：3
FIFO页面置换的缺页数为：32，缺页率是:32.00%
LRU 页面置换算法的缺页次数是：32，缺页率是:32.00%
OPT 页面置换算法的缺页次数是：26，缺页率是:26.00%
页表大小为：4
FIFO页面置换的缺页数为：28，缺页率是:28.00%
LRU 页面置换算法的缺页次数是：29，缺页率是:29.00%
OPT 页面置换算法的缺页次数是：22，缺页率是:22.00%
============第 5 次==========
页表大小为：3
FIFO页面置换的缺页数为：32，缺页率是:32.00%
LRU 页面置换算法的缺页次数是：31，缺页率是:31.00%
OPT 页面置换算法的缺页次数是：26，缺页率是:26.00%
页表大小为：4
FIFO页面置换的缺页数为：30，缺页率是:30.00%
LRU 页面置换算法的缺页次数是：30，缺页率是:30.00%
OPT 页面置换算法的缺页次数是：25，缺页率是:25.00%
============第 6 次==========
页表大小为：3
FIFO页面置换的缺页数为：25，缺页率是:25.00%
LRU 页面置换算法的缺页次数是：24，缺页率是:24.00%
OPT 页面置换算法的缺页次数是：22，缺页率是:22.00%
页表大小为：4
FIFO页面置换的缺页数为：24，缺页率是:24.00%
LRU 页面置换算法的缺页次数是：23，缺页率是:23.00%
OPT 页面置换算法的缺页次数是：20，缺页率是:20.00%
```

图 4-6 Belady 测试用例 1 4-6 执行结果图

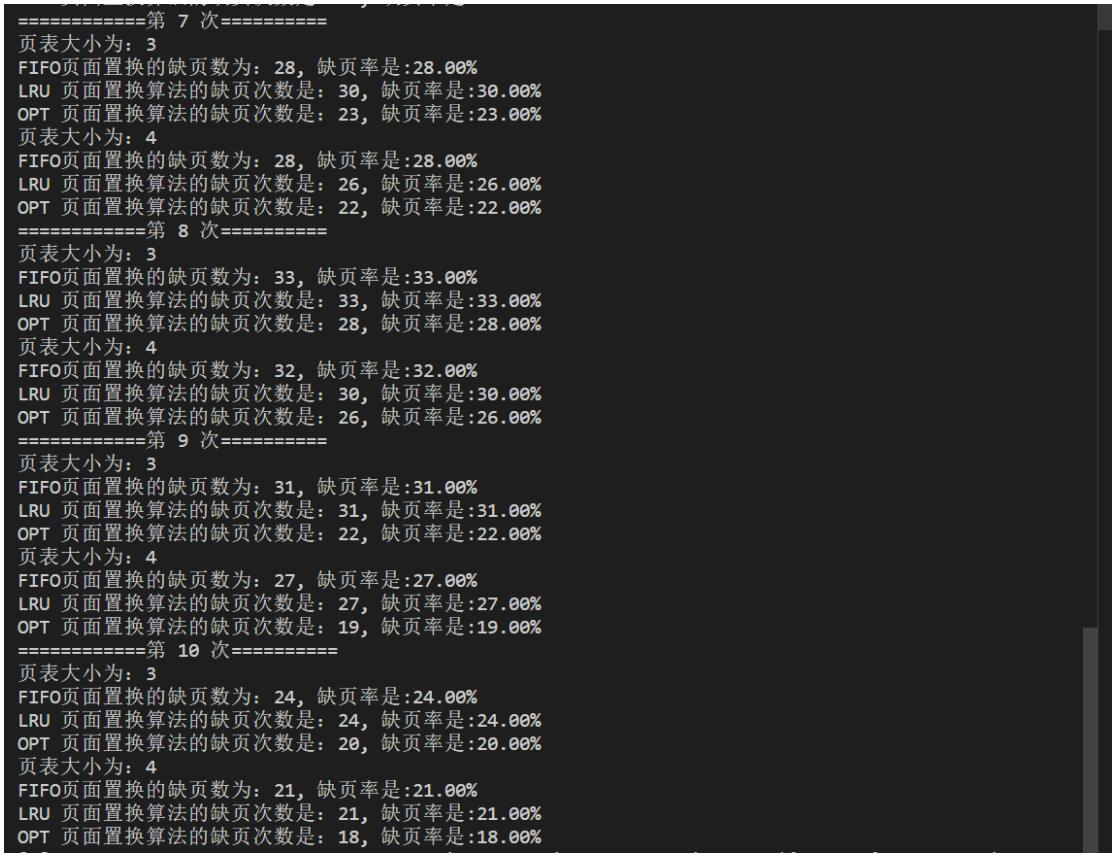

图 4-7 Belady 测试用例 1 7-10 执行结果图

结果及分析：本次测试并没有发现 Belady 异常。

4.2.2 测试用例 2

第二次测试，我采用 8 次循环，每次取页表大小 5 和 6 来测试。执行结果如下所示：

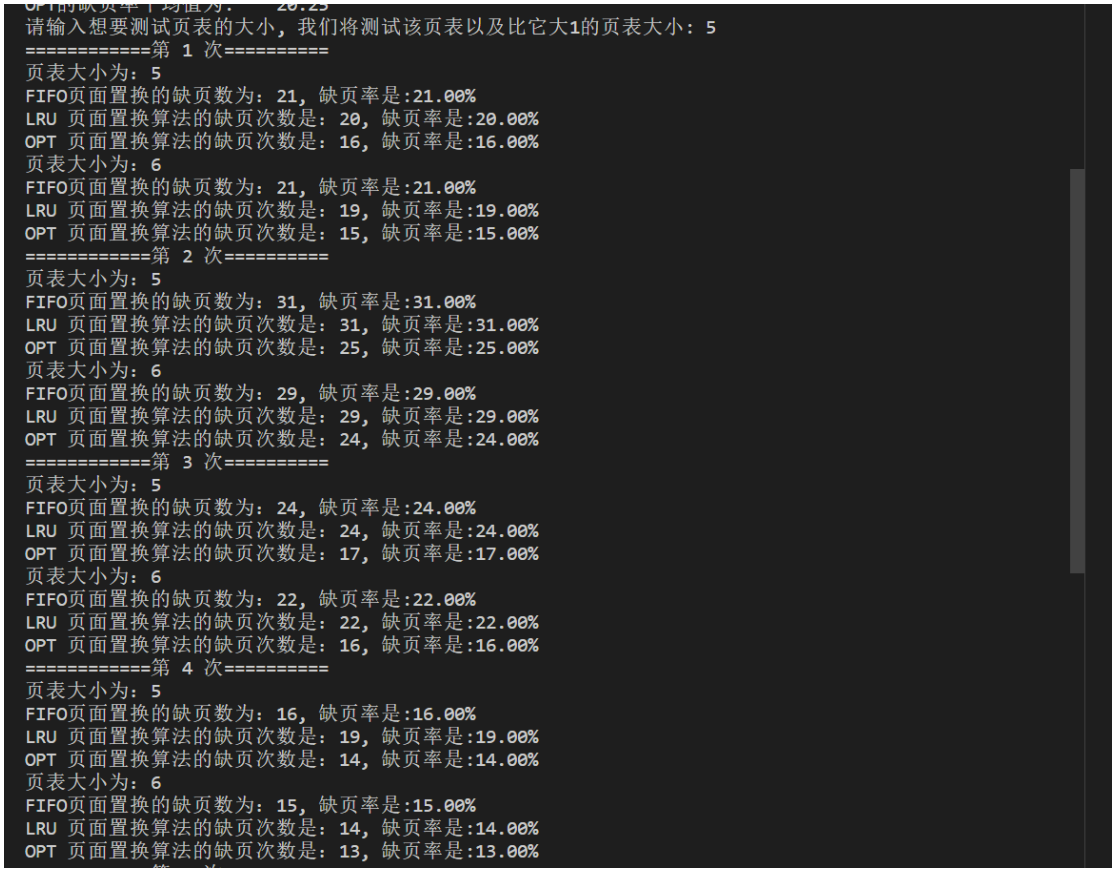

图 4-8 Belady 测试 2 1-4 执行结果图

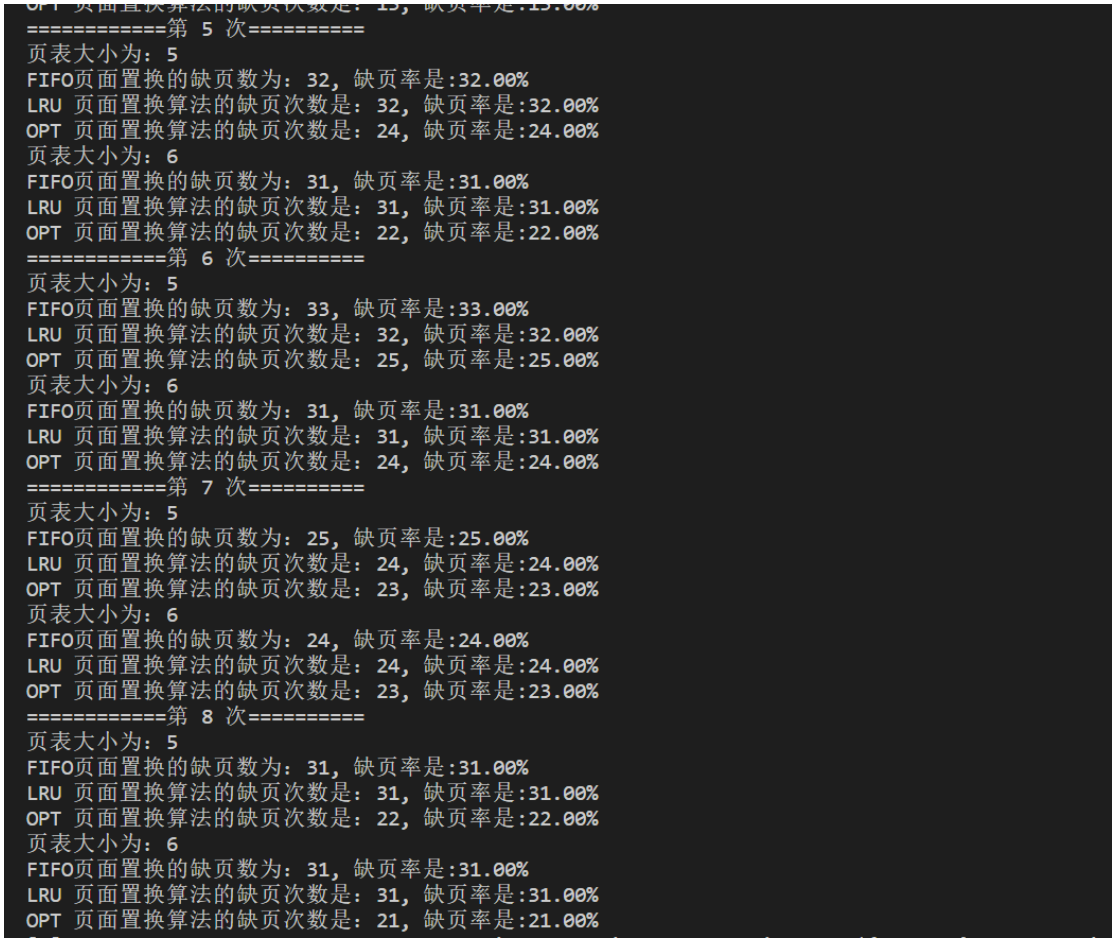

图 4-9 Belady 测试 2 5-8 执行结果图

结果分析：这八次实验也没有发现 Belady 异常。

4.2.3 测试用例 3—FIFO 的 Belady 异常

第三次测试，我加大了剂量，采用循环 10000 次，每次取页表大小 8 和 9 来测试。并且为了不在分析时看得眼花缭乱，我修改了代码片段，只有当出现 Belady 异常时，循环停止，代码如下：

```
1.  printf("请输入想要测试页表的大小，我们将测试该页表以及比它大 1 的页表大
    小：");
2.  scanf("%d", &pageTableSize);
3.  for(int i = 0 ;i < total ; i++){
4.      printf("==========第 %d 次==========\n", i+1);
5.      generateAddressPages(instructions, pages, i);
6.      printf("页表大小为：%d\n", pageTableSize);
```

```
7.      double a = fifo(pages);
8.      double c = lru(pages);
9.      double e = opt(pages);
10.     pageTableSize++;
11.     printf("页表大小为：%d\n", pageTableSize);
12.     double b = fifo(pages);
13.     double d = lru(pages);
14.     double f = opt(pages);
15.     pageTableSize--;
16.     if(b-a > 1e-4 || d-c >1e-4 || f-e > 1e-4)
17.         break;
18. }
```

这次实验一直执行到第 1623 次才停止，终于出现了 Belady 异常

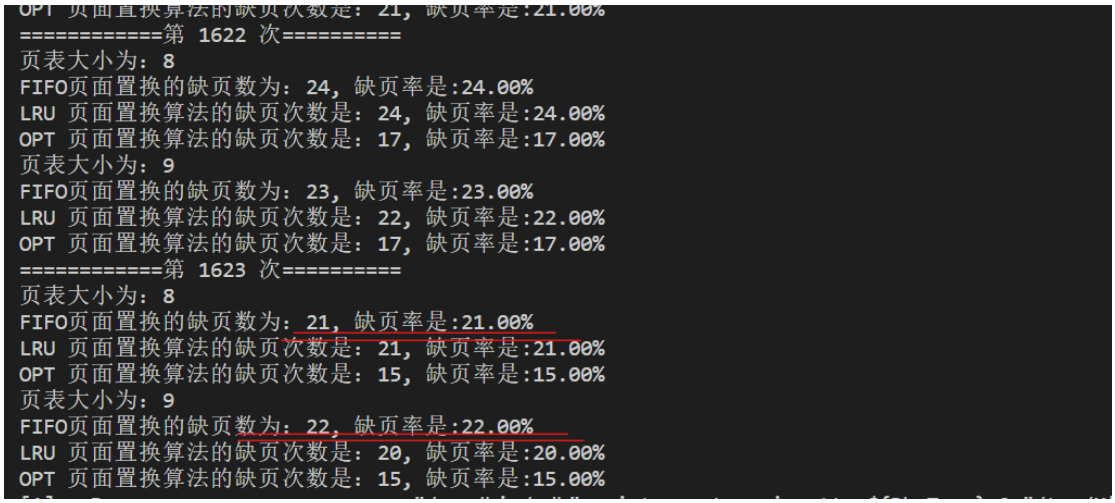

图 4-10 Belady 测试用例 3 执行结果图

此次实验证明，FIFO 页面置换算法有 Belady 异常。

4.2.4 测试用例 4—观察 LRU 和 OPT 是否有 Belady 异常

在测试用例 3 的基础上，我再次加大了剂量，这次采用 100000 次循环，每次采用 8、9 的页表大小，只观察 LRU 算法和 OPT 算法的缺页次数，若是有 Belady 异常，则停止，若是没有，则一直循环打印结果。代码修改如下：

```
1.  for(int i = 0 ;i < total ; i++){
2.      printf("==========第 %d 次==========\n", i+1);
3.      generateAddressPages(instructions, pages, i);
```

```
        printf("页表大小为：%d\n", pageTableSize);
        double a = fifo(pages);
        double c = lru(pages);
        double e = opt(pages);
        pageTableSize++;
        printf("页表大小为：%d\n", pageTableSize);
        double b = fifo(pages);
        double d = lru(pages);
        double f = opt(pages);
        pageTableSize--;
        if( d-c >1e-4 || f-e > 1e-4)
            break;
    }
```

执行结果如下：

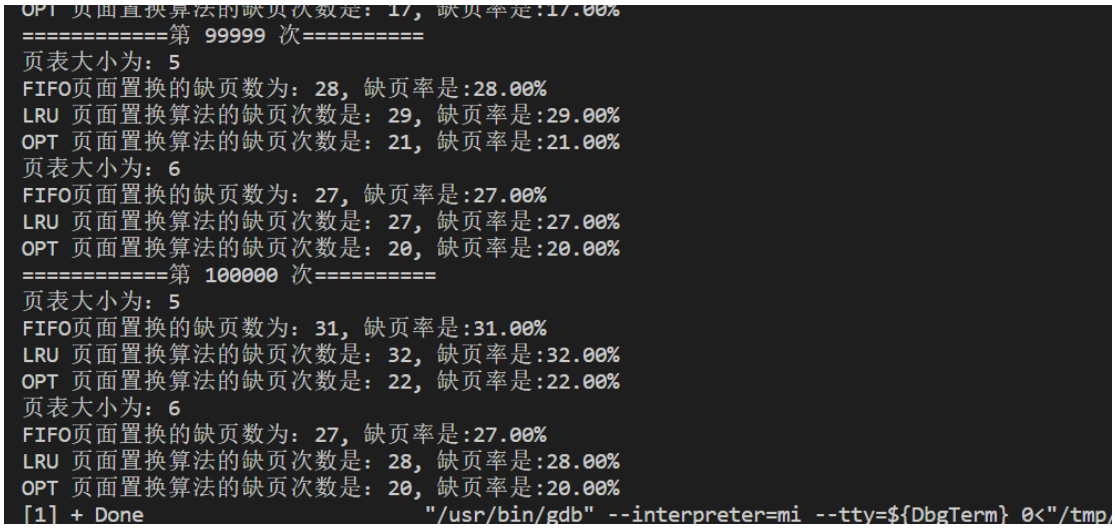

图 4-11 测试用例 4 执行结果图

执行结果分析：一直到第 100000 次循环，LRU 和 OPT 算法都没有 Belady 异常，虽然不能证明这两个算法永远都不会出现 Belady 异常，但是可以证明的是，在页表大小变大的时候，这两个算法的性能提升比 FIFO 算法大。

5 源码

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <string.h>
#define TOTAL_INSTRUCTIONS 100
```

```
#define PAGE_SIZE 10
#define PAGE_COUNT 10

int pageTableSize;

// 在页表查看是否缺页
int findPage(int pageTable[], int pageNum){
    for(int i = 0 ; i < pageTableSize; i++){
        if(pageTable[i] == pageNum){
            return i;
        }
    }
    return -1;
}
// 找到 OPT 要置换的那个元素下标，并返回
int findOPT(int pageTable[], int pages[], int currentIndex){
    // 在页表中，找到每个元素下一次要访问的距离
    int distance[pageTableSize];
    for(int i = 0 ; i < pageTableSize; i++){
        int j;
        for(j = currentIndex+1; j < TOTAL_INSTRUCTIONS; j++){
            // 找到了
            if(pageTable[i] == pages[j])
                break;
        }
        distance[i] = j - currentIndex;
    }
    // 比较距离，选择最大的
    int maxDistance = 0, retIndex = 0;
    for(int i = 0 ; i < pageTableSize; i++){
        if(maxDistance < distance[i]){
            maxDistance = distance[i];
            retIndex = i;
        }
    }
    return retIndex;
}
// 刷新 lru 的时间
void flushLruTime(int lruTime[]){
    for(int i = 0;i < pageTableSize; i++){
        lruTime[i]++;
    }
```

```
48. }
49. // 找到 lru 时间最大的那个表项
50. int findLruMax(int lruTime[]){
51.     int max = -1;
52.     int retIndex = 0;
53.     for(int i = 0;i < pageTableSize; i++){
54.         if(lruTime[i] > max){
55.             max = lruTime[i];
56.             retIndex = i;
57.         }
58.     }
59.     return retIndex;
60. }
61. // 先进先出页面置换
62. double fifo(int pages[]){
63.     int pageTable[pageTableSize];
64.     memset(pageTable, -1, sizeof(pageTable));
65.     int index = 0;
66.     int countPageFault = 0;
67.     for(int i = 0; i < TOTAL_INSTRUCTIONS; i++){
68.         // 模拟访问页表--缺页
69.         if(findPage(pageTable, pages[i]) == -1){
70.             // 没找着--缺页
71.             countPageFault++;
72.             pageTable[index] = pages[i];
73.             index = (index + 1) % pageTableSize;
74.         }
75.     }
76.     double faultRate = 100*(double)countPageFault/(double)TOTAL_INSTR
    UCTIONS;
77.     printf("FIFO 页面置换的缺页数为: %d, 缺页率
    是:%.2lf%%\n",countPageFault, faultRate);
78.     return faultRate;
79. }
80. // LRU 页面置换
81. double lru(int pages[]){
82.     int lruTime[pageTableSize], pageTable[pageTableSize];
83.     memset(pageTable, -1, sizeof(pageTable));
84.     int lruPageFaultCount = 0;
85.     for(int i = 0; i < pageTableSize; i++){
86.         lruTime[i] = 100;
87.     }
```

```
88.     for(int i = 0 ; i < TOTAL_INSTRUCTIONS; i++){
89.         if(findPage(pageTable, pages[i]) == -1){
90.             // 没找着
91.             int victimIndex = findLruMax(lruTime);
92.             pageTable[victimIndex] = pages[i];
93.             lruTime[victimIndex] = 0;
94.             lruPageFaultCount++;
95.         }else{
96.             // 找着了
97.             int index = findPage(pageTable, pages[i]);
98.             lruTime[index] = 0;
99.         }
100.            flushLruTime(lruTime);
101.        }
102.        double faultRate = 100*(double)lruPageFaultCount/(double)TOTAL
    _INSTRUCTIONS;
103.        printf("LRU 页面置换算法的缺页次数是：%d，缺页率
    是:%.2lf%%\n", lruPageFaultCount, faultRate);
104.        return faultRate;
105.    }
106.    // OPT 置换，输出缺页信息
107.    double opt(int pages[]){
108.        int pageTable[pageTableSize];
109.        int optCount = 0;
110.        memset(pageTable, -1, sizeof(pageTable));
111.        for(int i = 0; i < TOTAL_INSTRUCTIONS; i++){
112.            if(findPage(pageTable, pages[i]) == -1){
113.                int victimIndex = findOPT(pageTable, pages, i);
114.                pageTable[victimIndex] = pages[i];
115.                optCount ++;
116.            }
117.        }
118.        double faultRate = 100*(double)optCount/TOTAL_INSTRUCTIONS;
119.        printf("OPT 页面置换算法的缺页次数是：%d，缺页率
    是:%.2lf%%\n", optCount, faultRate);
120.        return faultRate;
121.    }
122.    // 按照实验要求生成地址及页面流
123.    void generateAddressPages(int instructions[], int pages[], int see
    d) {
124.        srand(time(NULL)+seed);
125.        int currentAddress = 0;
```

```
126.        for (int i = 0; i < TOTAL_INSTRUCTIONS; i++) {
127.            double probability = ((double)rand() / RAND_MAX) * 100;
128.            if (probability < 70) {
129.                // 70%的概率选择顺序执行
130.                currentAddress++;
131.            } else if (probability < 80) {
132.                // 10%的概率选择在前地址部分均匀分布
133.                currentAddress = rand() % (i == 0 ? 1 : i);
134.            } else {
135.                // 20%的概率选择在后地址部分均匀分布
136.                currentAddress = (rand() % (TOTAL_INSTRUCTIONS - i + 1
    )) + currentAddress + 1;
137.            }
138.            instructions[i] = currentAddress;
139.        }
140.        for(int i = 0; i < TOTAL_INSTRUCTIONS; i++){
141.            pages[i] = instructions[i] / PAGE_SIZE;
142.        }
143.    }
144.
145.    int main() {
146.        int instructions[TOTAL_INSTRUCTIONS], pages[TOTAL_INSTRUCTIONS
    ];
147.        double fifoSum = 0, lruSum = 0, optSum = 0;
148.        int total;
149.        printf("请输入想要实验的次数: ");
150.        scanf("%d", &total);
151.        printf("请输入想要测试的页表大小: ");
152.        scanf("%d", &pageTableSize);
153.        for(int i = 0; i < total; i++){
154.            printf("第 %d 次: \n", i+1);
155.            generateAddressPages(instructions, pages, i);
156.            double fifoRate = fifo(pages);
157.            double lruRate = lru(pages);
158.            double optRate = opt(pages);
159.            fifoSum += fifoRate;
160.            lruSum += lruRate;
161.            optSum += optRate;
162.        }
163.        double avgFifoRate = fifoSum / total ;
164.        double avgLruRate = lruSum / total;
165.        double avgOptRate = optSum / total;
```

```
166.
167.       printf("经过 %d 实验\nFIFO 的缺页率平均值为:\t%.2lf\nLRU 的缺页率平均值为:\t%.2lf\nOPT 的缺页率平均值为:\t%.2lf\n",total, avgFifoRate, avgLruRate, avgOptRate);
168.       printf("请输入想要测试页表的大小，我们将测试该页表以及比它大 1 的页表大小: ");
169.       scanf("%d", &pageTableSize);
170.       for(int i = 0 ;i < total ; i++){
171.           printf("============第 %d 次==========\n", i+1);
172.           generateAddressPages(instructions, pages, i);
173.           printf("页表大小为：%d\n", pageTableSize);
174.           double a = fifo(pages);
175.           double c = lru(pages);
176.           double e = opt(pages);
177.           pageTableSize++;
178.           printf("页表大小为：%d\n", pageTableSize);
179.           double b = fifo(pages);
180.           double d = lru(pages);
181.           double f = opt(pages);
182.           pageTableSize--;
183.           if( b-a > 1e-4 || d-c >1e-4 || f-e > 1e-4)
184.               break;
185.       }
186.       return 0;
187.   }
```