|  |
| --- |
| 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 2](#_Toc152494693)

[1.1 实验目的 2](#_Toc152494694)

[1.2 实验内容 2](#_Toc152494695)

[1.2.1生成内存访问串 2](#_Toc152494696)

[1.2.2 性能比较 3](#_Toc152494697)

[1.3 实验要求 3](#_Toc152494698)

[2 环境说明 3](#_Toc152494699)

[3 程序设计 3](#_Toc152494700)

[3.1 全局变量定义 3](#_Toc152494701)

[3.2 函数定义 4](#_Toc152494702)

[3.2.1 generateAddressPages() 4](#_Toc152494703)

[3.2.2 double fifo() 5](#_Toc152494704)

[3.2.3 double lru() 6](#_Toc152494705)

[3.2.4 double opt() 7](#_Toc152494706)

[3.2.5 int findPage() 8](#_Toc152494707)

[3.2.6 int findOPT() 8](#_Toc152494708)

[3.2.7 flushLruTime() 9](#_Toc152494709)

[3.2.8 findLruMax() 10](#_Toc152494710)

[3.2.9 主函数main() 11](#_Toc152494711)

[4 测试报告 13](#_Toc152494712)

[4.1 比较三种性能页面置换算法的性能 13](#_Toc152494713)

[4.1.1 测试用例1 13](#_Toc152494714)

[4.1.2 测试用例2 15](#_Toc152494715)

[4.2 测试三种算法的Belady异常 17](#_Toc152494716)

[4.2.1 测试用例1 18](#_Toc152494717)

[4.2.2 测试用例2 19](#_Toc152494718)

[4.2.3 测试用例3—FIFO的Belady异常 21](#_Toc152494719)

[4.2.4 测试用例4—观察LRU和OPT是否有Belady异常 22](#_Toc152494720)

[5 源码 23](#_Toc152494721)

# 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

1. #define TOTAL\_INSTRUCTIONS 100
2. #define PAGE\_SIZE 10
3. #define PAGE\_COUNT 10
4. 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
2. **double** fifo(**int** pages[]){
3. **int** pageTable[PAGE\_TABLE\_SIZE];
4. memset(pageTable, -1, **sizeof**(pageTable));
5. **int** index = 0;
6. **int** countPageFault = 0;
7. **for**(**int** i = 0; i < TOTAL\_INSTRUCTIONS; i++){
8. // 模拟访问页表--缺页
9. **if**(findPage(pageTable, pages[i]) == -1){
10. // 没找着--缺页
11. countPageFault++;
12. pageTable[index] = pages[i];
13. index = (index + 1) % PAGE\_TABLE\_SIZE;
14. }
15. }
16. **double** faultRate = 100\*(**double**)countPageFault/(**double**)TOTAL\_INSTRUCTIONS;
17. printf("FIFO页面置换的缺页数为：%d, 缺页率是:%.2lf%%\n",countPageFault, faultRate);
18. **return** faultRate;
19. }

### 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。

1. // LRU页面置换
2. **double** lru(**int** pages[]){
3. **int** lruTime[PAGE\_TABLE\_SIZE], pageTable[PAGE\_TABLE\_SIZE];
4. memset(pageTable, -1, **sizeof**(pageTable));
5. **int** lruPageFaultCount = 0;
6. **for**(**int** i = 0; i < PAGE\_TABLE\_SIZE; i++){
7. lruTime[i] = 100;
8. }
9. **for**(**int** i = 0 ; i < TOTAL\_INSTRUCTIONS; i++){
10. **if**(findPage(pageTable, pages[i]) == -1){
11. // 没找着
12. **int** victimIndex = findLruMax(lruTime);
13. pageTable[victimIndex] = pages[i];
14. lruTime[victimIndex] = 0;
15. lruPageFaultCount++;
16. }**else**{
17. // 找着了
18. **int** index = findPage(pageTable, pages[i]);
19. lruTime[index] = 0;
20. }
21. flushLruTime(lruTime);
22. }
23. **double** faultRate = 100\*(**double**)lruPageFaultCount/(**double**)TOTAL\_INSTRUCTIONS;
24. printf("LRU 页面置换算法的缺页次数是：%d, 缺页率是:%.2lf%%\n", lruPageFaultCount, faultRate);
25. }

### 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。

1. // OPT置换，输出缺页信息
2. **double** opt(**int** pages[]){
3. **int** pageTable[PAGE\_TABLE\_SIZE];
4. **int** optCount = 0;
5. memset(pageTable, -1, **sizeof**(pageTable));
6. **for**(**int** i = 0; i < TOTAL\_INSTRUCTIONS; i++){
7. **if**(findPage(pageTable, pages[i]) == -1){
8. **int** victimIndex = findOPT(pageTable, pages, i);
9. pageTable[victimIndex] = pages[i];
10. optCount ++;
11. }
12. }
13. **double** faultRate = 100\*(**double**)optCount/TOTAL\_INSTRUCTIONS;
14. printf("OPT 页面置换算法的缺页次数是：%d, 缺页率是:%.2lf%%\n", optCount, faultRate);
15. **return** faultRate;
16. }

### 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。

1. // 在页表查看是否缺页
2. **int** findPage(**int** pageTable[], **int** pageNum){
3. **for**(**int** i = 0 ; i < PAGE\_TABLE\_SIZE; i++){
4. **if**(pageTable[i] == pageNum){
5. **return** i;
6. }
7. }
8. **return** -1;
9. }

### 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。

1. // 找到OPT要置换的那个元素下标，并返回
2. **int** findOPT(**int** pageTable[], **int** pages[], **int** currentIndex){
3. // 在页表中，找到每个元素下一次要访问的距离
4. **int** distance[PAGE\_TABLE\_SIZE];
5. **for**(**int** i = 0 ; i < PAGE\_TABLE\_SIZE; i++){
6. **int** j;
7. **for**(j = currentIndex+1; j < TOTAL\_INSTRUCTIONS; j++){
8. // 找到了
9. **if**(pageTable[i] == pages[j])
10. **break**;
11. }
12. distance[i] = j - currentIndex;
13. }
14. // 比较距离，选择最大的
15. **int** maxDistance = 0, retIndex = 0;
16. **for**(**int** i = 0 ; i < PAGE\_TABLE\_SIZE; i++){
17. **if**(maxDistance < distance[i]){
18. maxDistance = distance[i];
19. retIndex = i;
20. }
21. }
22. **return** retIndex;
23. }

### 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。

1. // 刷新lru的时间
2. **void** flushLruTime(**int** lruTime[]){
3. **for**(**int** i = 0;i < PAGE\_TABLE\_SIZE; i++){
4. lruTime[i]++;
5. }
6. }

### 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。

1. // 找到lru时间最大的那个表项
2. **int** findLruMax(**int** lruTime[]){
3. **int** max = -1;
4. **int** retIndex = 0;
5. **for**(**int** i = 0;i < PAGE\_TABLE\_SIZE; i++){
6. **if**(lruTime[i] > max){
7. max = lruTime[i];
8. retIndex = i;
9. }
10. }
11. **return** retIndex;
12. }

### 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。

1. **int** main() {
2. **int** instructions[TOTAL\_INSTRUCTIONS], pages[TOTAL\_INSTRUCTIONS];
3. **double** fifoSum = 0, lruSum = 0, optSum = 0;
4. **int** total;
5. printf("请输入想要实验的次数: ");
6. scanf("%d", &total);
7. printf("请输入想要测试的页表大小: ");
8. scanf("%d", &pageTableSize);
9. **for**(**int** i = 0; i < total; i++){
10. printf("第 %d 次：\n", i+1);
11. generateAddressPages(instructions, pages, i);
12. **double** fifoRate = fifo(pages);
13. **double** lruRate = lru(pages);
14. **double** optRate = opt(pages);
15. fifoSum += fifoRate;
16. lruSum += lruRate;
17. optSum += optRate;
18. }
19. **double** avgFifoRate = fifoSum / total ;
20. **double** avgLruRate = lruSum / total;
21. **double** avgOptRate = optSum / total;
23. 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);
24. 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: ");
25. scanf("%d", &pageTableSize);
26. **for**(**int** i = 0 ;i < total ; i++){
27. printf("============第 %d 次==========\n", i+1);
28. generateAddressPages(instructions, pages, i);
29. printf("页表大小为：%d\n", pageTableSize);
30. fifo(pages);
31. lru(pages);
32. opt(pages);
33. pageTableSize++;
34. printf("页表大小为：%d\n", pageTableSize);
35. fifo(pages);
36. lru(pages);
37. opt(pages);
38. pageTableSize--;
39. }
41. **return** 0;
42. }

# 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：

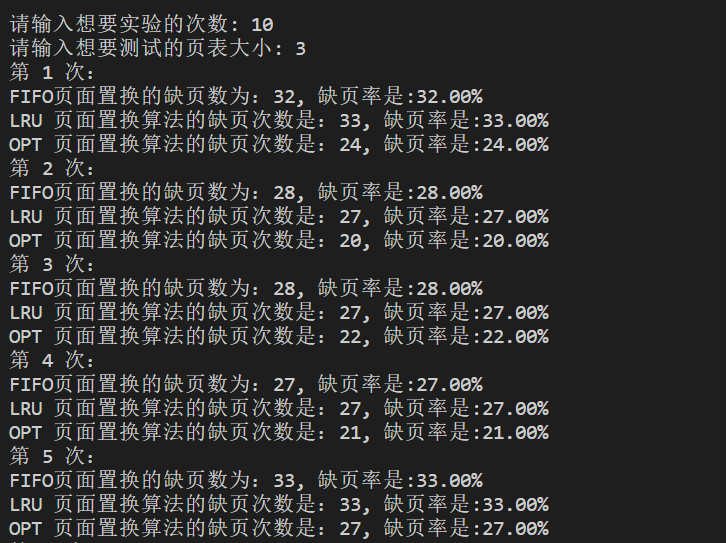


图4-1测试1 1-5 执行结果图

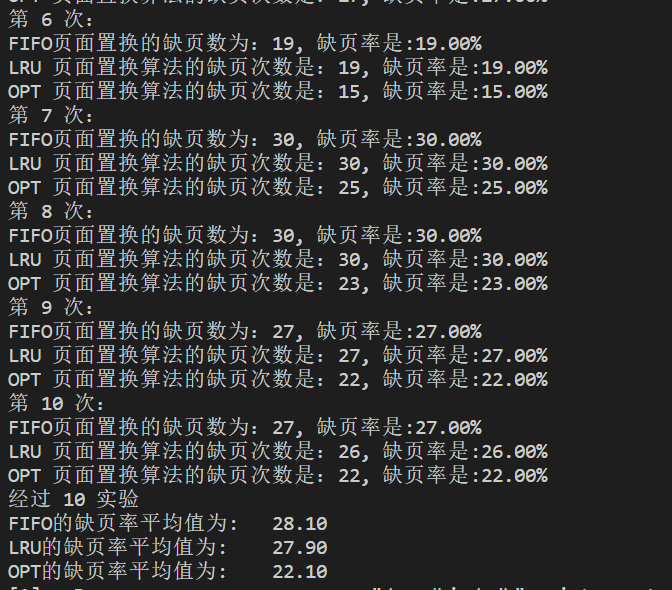


图4-2 测试1 6-10 执行结果图

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：

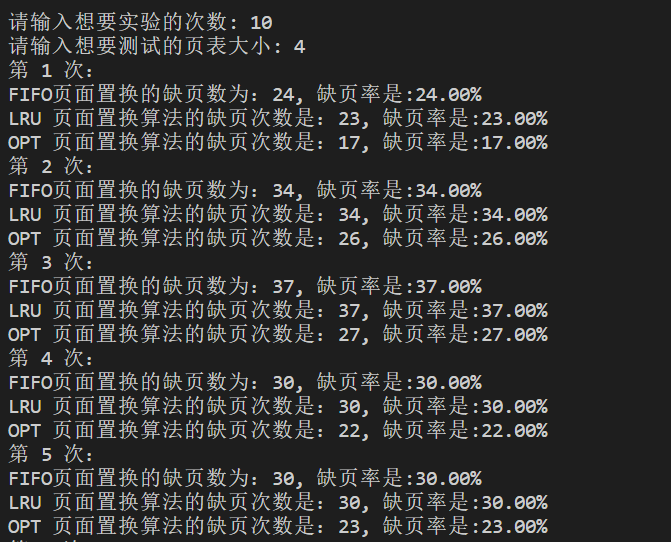


图4-3 测试2 1-5 执行结果图

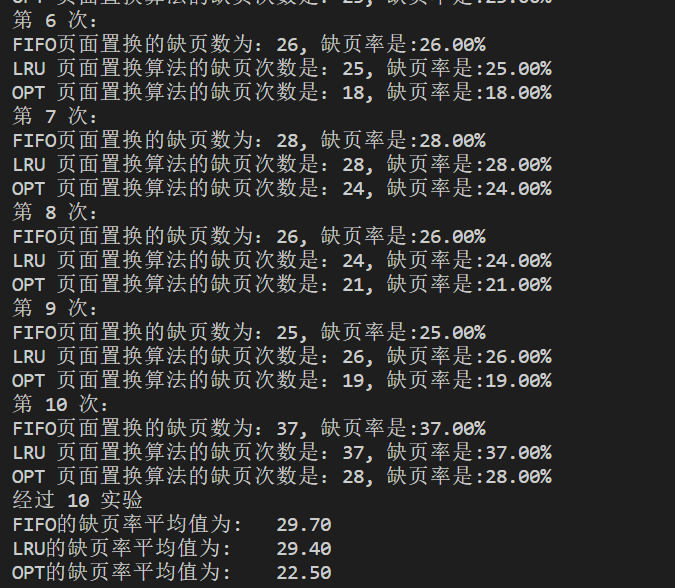


图4-4 测试2 6-10执行结果图

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。

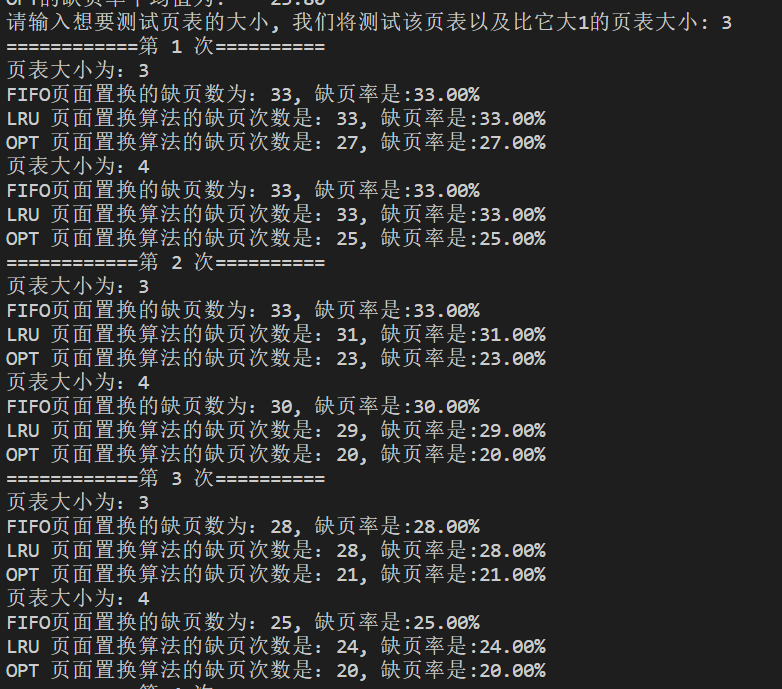


图4-5 Belady测试用例1 1-3执行结果图

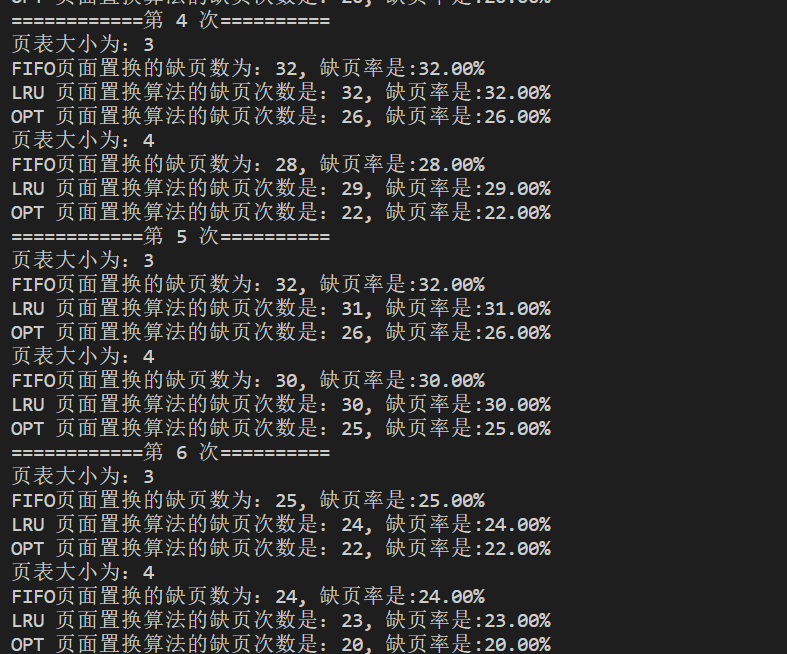


图4-6 Belady测试用例1 4-6执行结果图

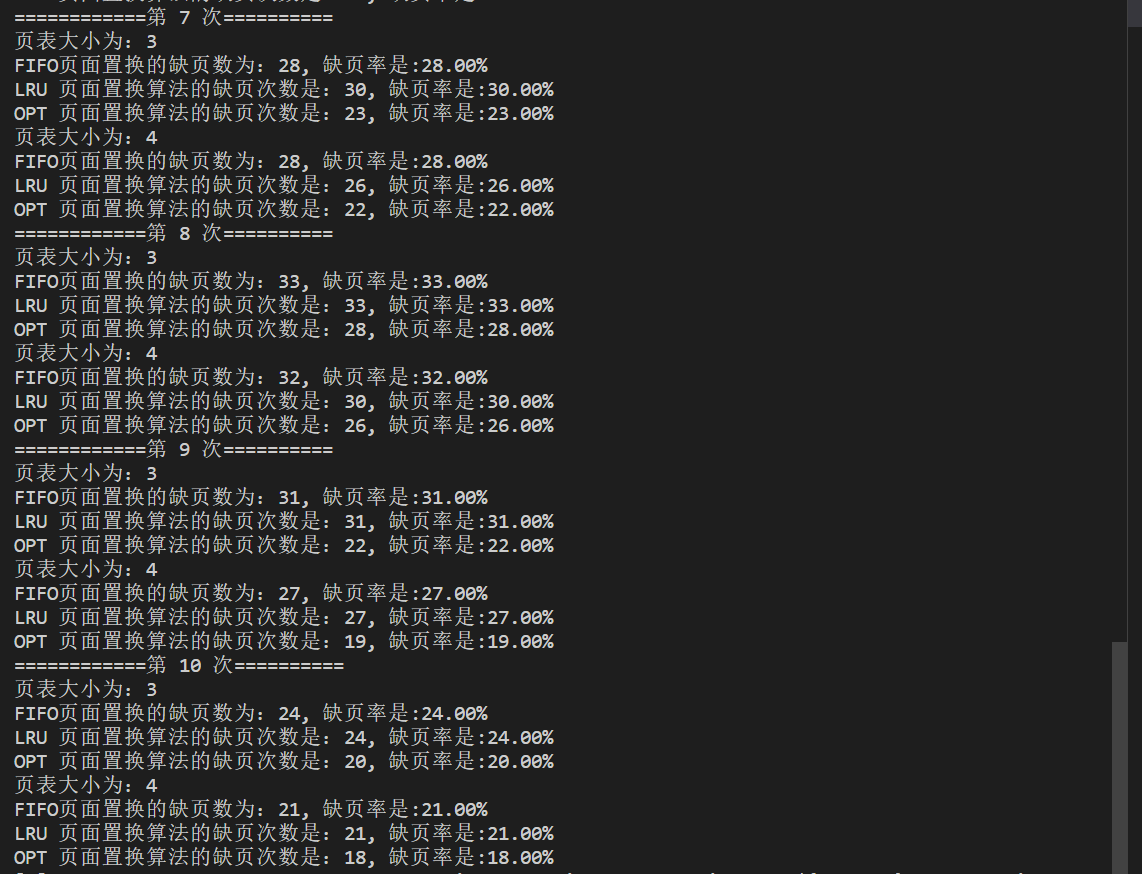


图4-7 Belady测试用例1 7-10 执行结果图

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：

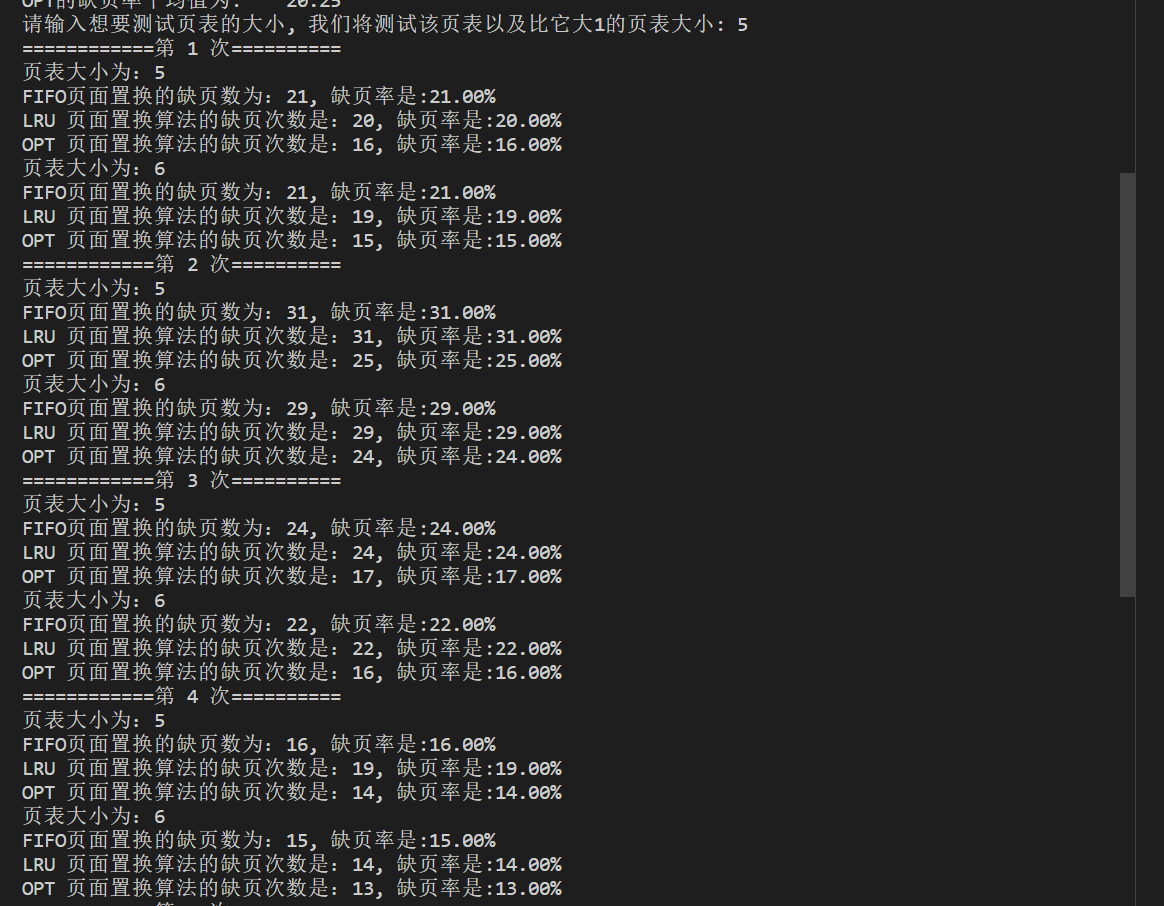


图4-8 Belady测试2 1-4执行结果图

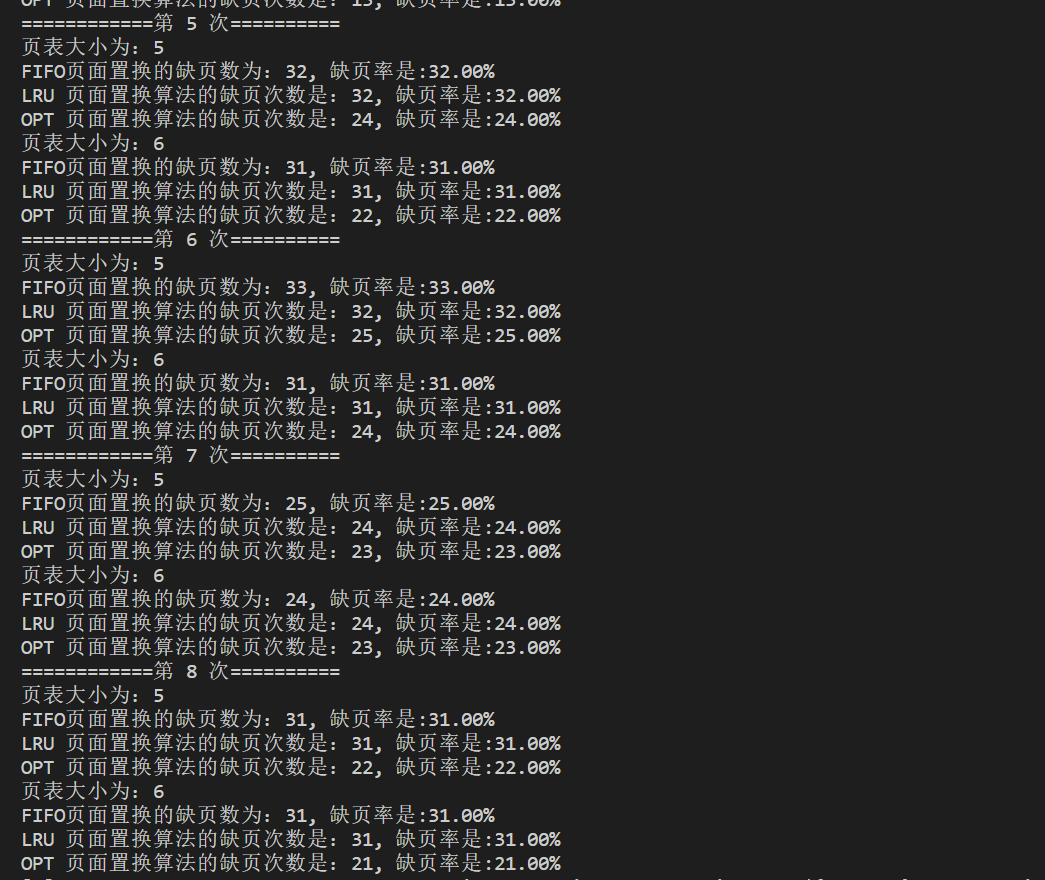


图4-9 Belady测试2 5-8执行结果图

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

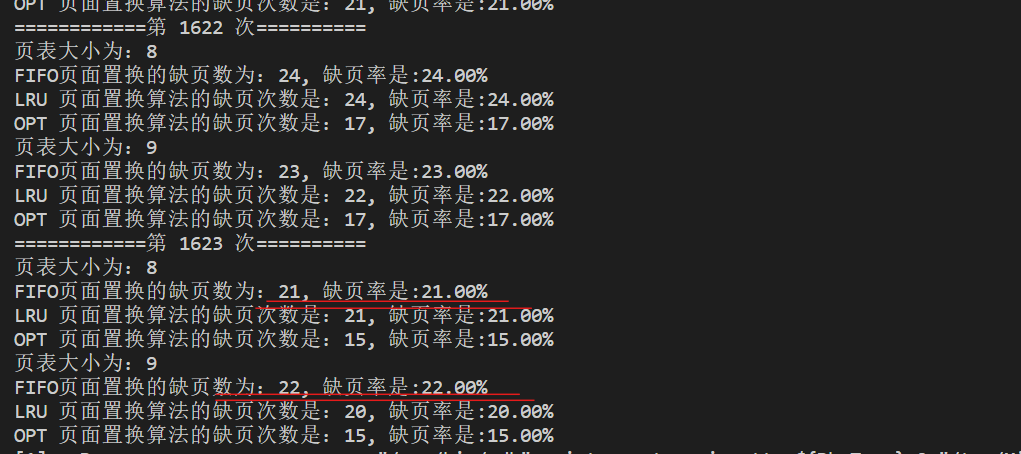


图 4-10 Belady 测试用例3 执行结果图

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

执行结果如下：

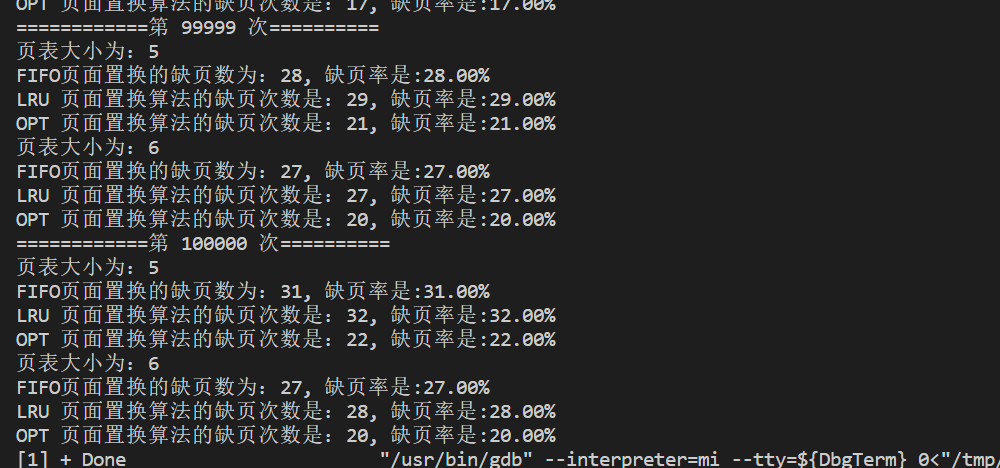


图4-11 测试用例4 执行结果图

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

1. #include <stdio.h>
2. #include <stdlib.h>
3. #include <time.h>
4. #include <string.h>
5. #define TOTAL\_INSTRUCTIONS 100
6. #define PAGE\_SIZE 10
7. #define PAGE\_COUNT 10
9. **int** pageTableSize;
11. // 在页表查看是否缺页
12. **int** findPage(**int** pageTable[], **int** pageNum){
13. **for**(**int** i = 0 ; i < pageTableSize; i++){
14. **if**(pageTable[i] == pageNum){
15. **return** i;
16. }
17. }
18. **return** -1;
19. }
20. // 找到OPT要置换的那个元素下标，并返回
21. **int** findOPT(**int** pageTable[], **int** pages[], **int** currentIndex){
22. // 在页表中，找到每个元素下一次要访问的距离
23. **int** distance[pageTableSize];
24. **for**(**int** i = 0 ; i < pageTableSize; i++){
25. **int** j;
26. **for**(j = currentIndex+1; j < TOTAL\_INSTRUCTIONS; j++){
27. // 找到了
28. **if**(pageTable[i] == pages[j])
29. **break**;
30. }
31. distance[i] = j - currentIndex;
32. }
33. // 比较距离，选择最大的
34. **int** maxDistance = 0, retIndex = 0;
35. **for**(**int** i = 0 ; i < pageTableSize; i++){
36. **if**(maxDistance < distance[i]){
37. maxDistance = distance[i];
38. retIndex = i;
39. }
40. }
41. **return** retIndex;
42. }
43. // 刷新lru的时间
44. **void** flushLruTime(**int** lruTime[]){
45. **for**(**int** i = 0;i < pageTableSize; i++){
46. lruTime[i]++;
47. }
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\_INSTRUCTIONS;
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** seed) {
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. }
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;
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. }