

**《操作系统》**

**页面淘汰算法模拟实现与比较**

|  |  |
| --- | --- |
|  |  |
| 学 号： |  |
| 姓 名： |  |
| 学 院： |  |
| 日 期： | 2022年 11 月 7 日 |

# 一、开发环境、运行环境、测试环境

使用vs 2022进行源代码的编写，使用vs 2022运行程序进行测试。

# 二、实验内容

（1）编程设计实现最佳淘汰算法、先进先出淘汰算法、最近最久未使用淘汰算法、简单Clock 淘汰算法及改进型 Clock 淘汰算法；

（2）编程设计实现页面访问序列的随机发生机制，包括各页面读写访问方式的设定以满足改进型 Clock 淘汰算法的要求；

（3）在执行进程和访问各页面过程中，每访问一个（或一次）页面应显示输出当时的进程页表内容（包括页号、物理块号、状态位、读/写访问方式等字段）及本次页面访问操作情况（譬如页面已在内存或触发缺页中断）；

（4）基于相同的条件，包括系统均采用固定分配局部置换策略、相同的进程逻辑地址空间大小（暨逻辑页面数，设进程逻辑地址空间的页面总数为 N，则其页号取值区间为[0, N)）、分配给进程同样多的物理块（设进程分配获得 S 个物理块，则相应物理块号分别标记为 PF 0 、PF 1 、……、PF S-1 ）、相同的页面访问序列（整数序列，整数取值区间为[0, N)）、均预装入前三个页面，进行有关算法的测试；

（5）变换上述条件实施多次测试，统计分析和比较有关算法的性能（譬如缺页率、淘汰页查找时间开销）。

# 三、实验步骤

## 一、关键数据结构

#define P 32//页面总数

#define F 8//物理块总数

int rs[RS\_MAX\_SIZE];//页面访问序列数组

int re[RS\_MAX\_SIZE];//页面读写方式数组

int rs\_size;//页面访问序列大小

初始化页面访问序列函数：

void CreateRS()

最佳置换算法：

void OPT()

先进先出法：

void FIFO()

最近最久未使用：

void LRU()

简单clock算法：

void Clock ()

改进Clock算法：

void MoreClock ()

## 二、算法流程

#### 1. 最佳置换算法OPT

**1）基本思想**

选择永不使用或是在最长时间内不再被访问（即距现在最长时间才会被访问）的页面淘汰出内存。

**2）评价**

理想化算法，具有最好性能（对于固定分配页面方式，本法可保证获得最低的缺页率），但实际上却难于实现，故主要用于算法评价参照。

#### 2. 先进先出置换算法FIFO

**1）基本思想**

选择最先进入内存即在内存驻留时间最久的页面换出到外存；

进程已调入内存的页面按进入先后次序链接成一个队列，并设置替换指针以指向最老页面。

**2）评价**

简单直观，但不符合进程实际运行规律，性能较差，故实际应用极少。

#### 3. 最近最久未使用置换算法LRU

**1）基本思想**

以“最近的过去”作为“最近的将来”的近似，选择最近一段时间最长时间未被访问的页面淘汰出内存。

**2）评价**

适用于各种类型的程序，性能较好，但需要较多的硬件支持。

#### 4. 简单Clock置换算法

**1）基本思想**

当某一页首次装入内存中时，则将该页框的使用位设置为1；当该页随后被访问到时（在访问产生缺页中断之后），它的使用位也会被设置为1。对于页面置换算法，用于置换算法，用于置换的候选页框集合（当前进程：局部范围；整个内存；全局范围）被看做是一个循环缓冲区，并且有一个指针与之相关联。当一页被置换时，该指针被设置成指向缓冲区中的下一页框。当需要置换一页时，操作系统扫描缓冲区，以查找使用位被置为0的一页框。每当遇到一个使用位为1的页框时，操作系统就将该位重新置为0；如果在这个过程开始时，缓冲区中所有页框的使用位均为0时，则选择遇到的第一个页框置换；如果所有页框的使用位均为1时，则指针在缓冲区中完整地循环一周，把所有使用位都置为0，并且停留在最初的位置上，置换该页框中的页。

**2）评价**

与上面的算法相比，占用系统资源数少。

#### 5. 改进型Clock置换算法

**1）基本思想**

① 从查寻指针当前位置起扫描内存分页循环队列，选择A=0且M=0的第一个页面淘汰；若未找到，转②

② 开始第二轮扫描，选择A=0且M=1的第一个页面淘汰，同时将经过的所有页面访问位置0；若不能找到，转①

**2）评价**

与简单Clock算法相比，可减少磁盘的I/O操作次数，但淘汰页的选择可能经历多次扫描，故实现算法自身的开销增大。

#### 6. 页面访问序列随机发生机制

（1）初始化进程逻辑地址空间页面总数 N、各逻辑页面的读写访问方式（是否支持写访问，即 R、RW）、工作集起始页号 s（s∈[0, N)）、工作集中包含的页数 w，工作集移动速率 v（每处理 v 个页面访问，就将工作集起始页号递增即 s+1）以及一个取值区间为[0, 1]的值 t；

（2）生成取值区间为[s, min(s+w, N-1)]的 v 个随机数并添加保存到页面访问序列中，同时为每次页面访问分别生成一个取值区间为[0, 1]的随机数，若该随机数值大于 0.7 且对应所访问页面支持写访问则设定以写方式访问相应页面，否则以读方式访问对应页面；

（3）生成取值区间为[0, 1]的一个随机数 r，并比较 r 与 t 的大小；

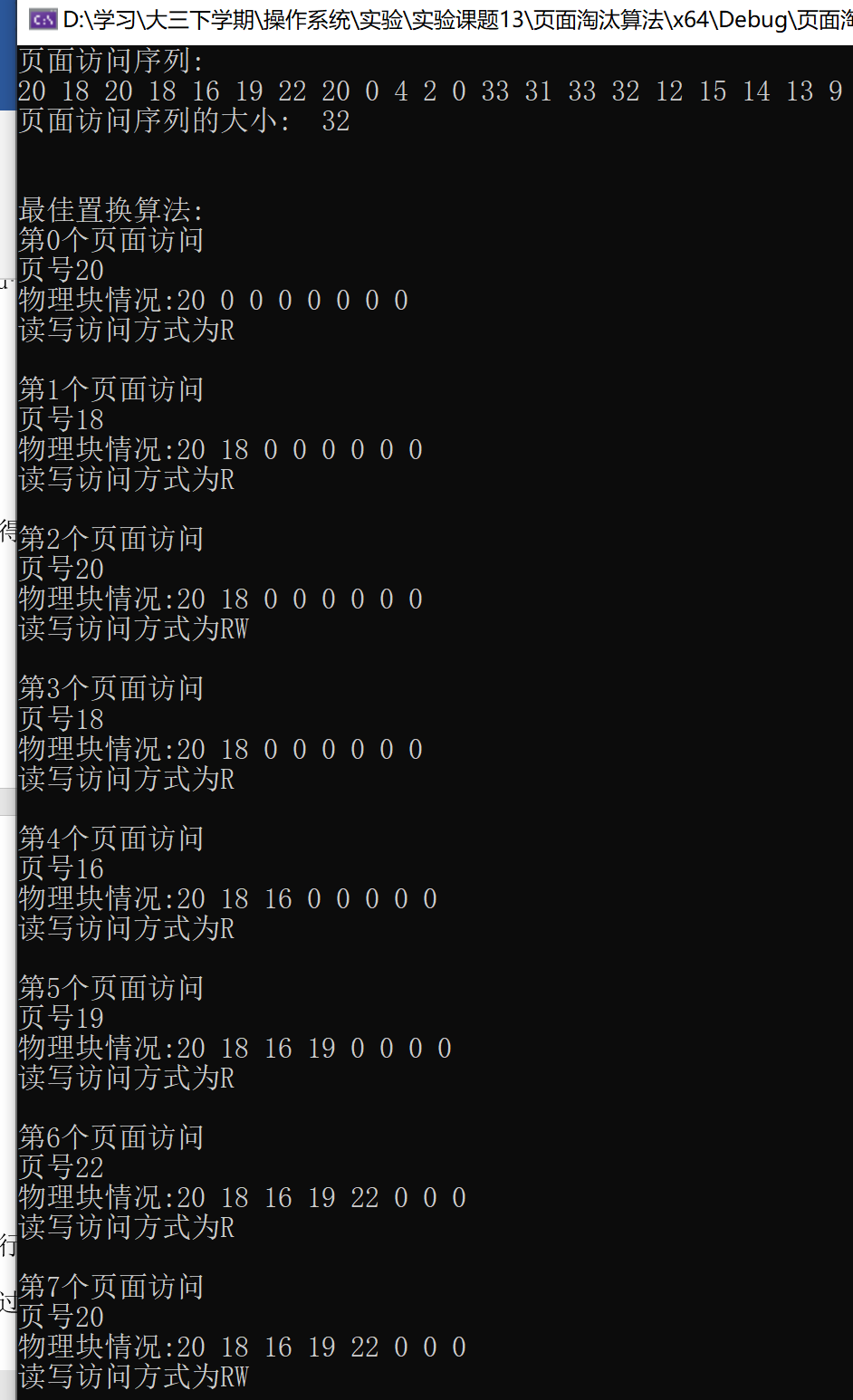
（4）若 r < t，则为 s 生成一个新值（s∈[0, N)），否则 s = (s + 1) mod N；

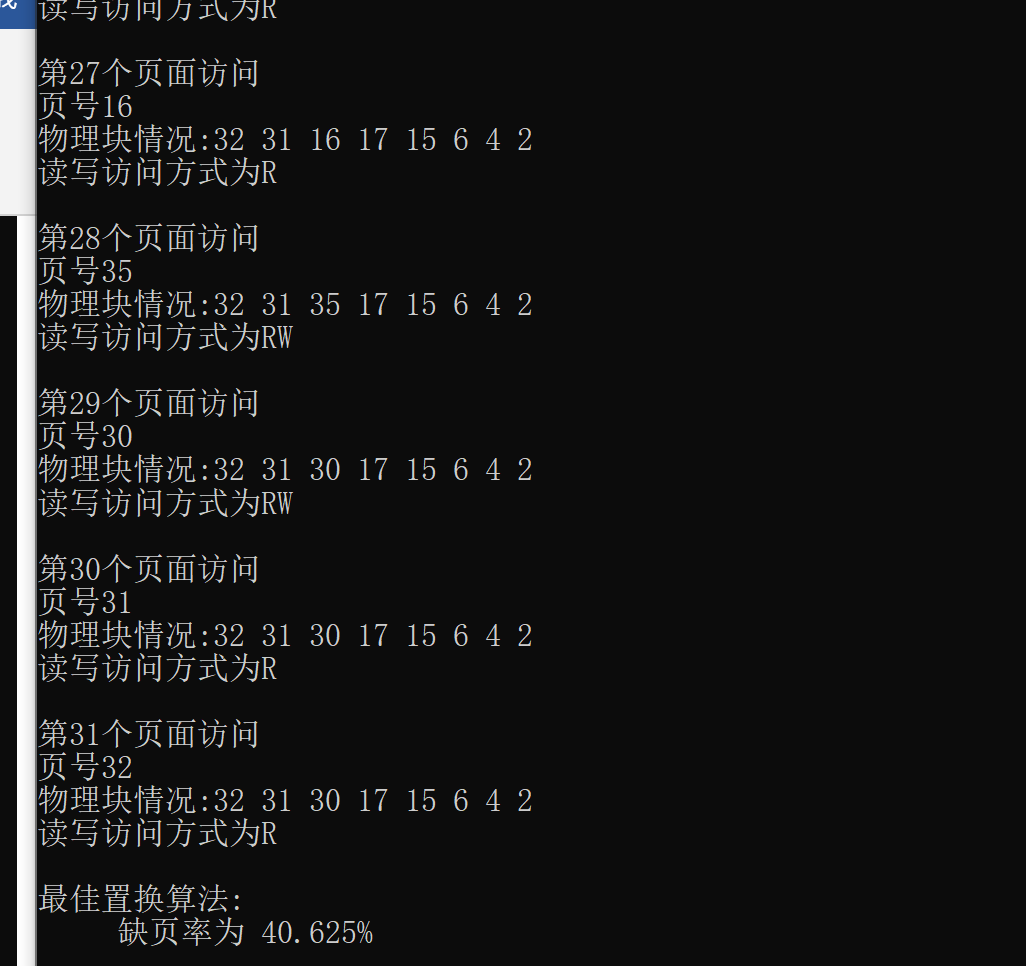
（5）如果想继续加大页面访问序列的长度，返回第 2 步，否则结束。

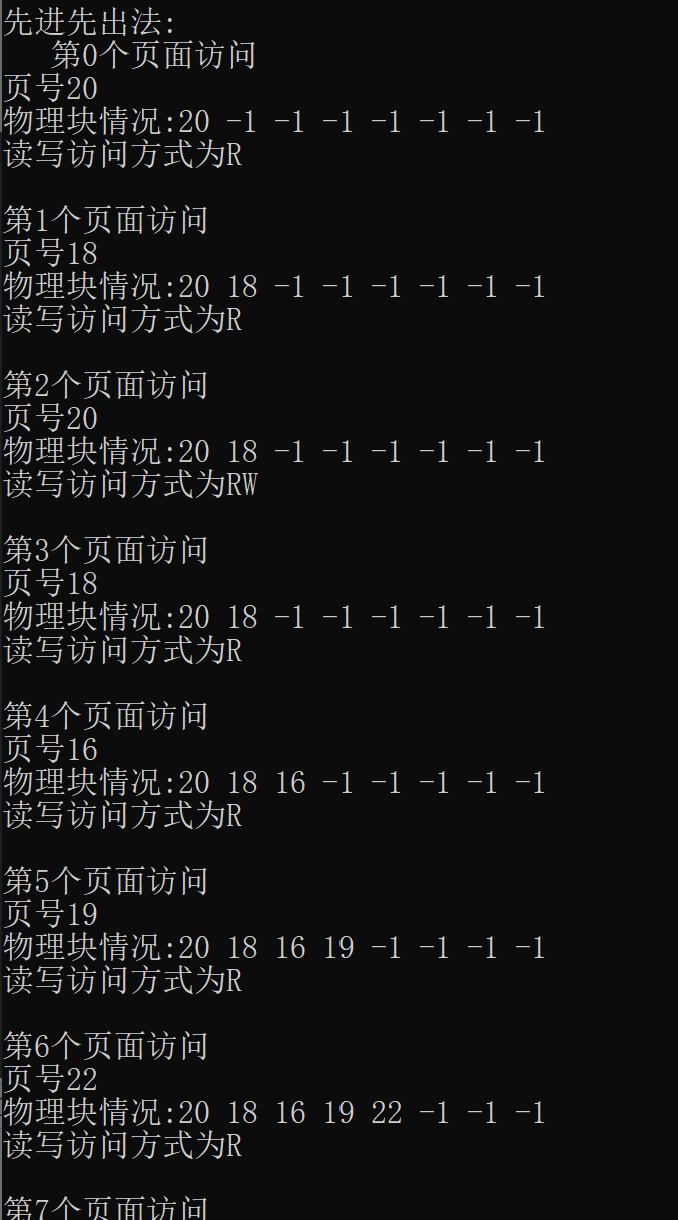
## 三、测试结果及分析

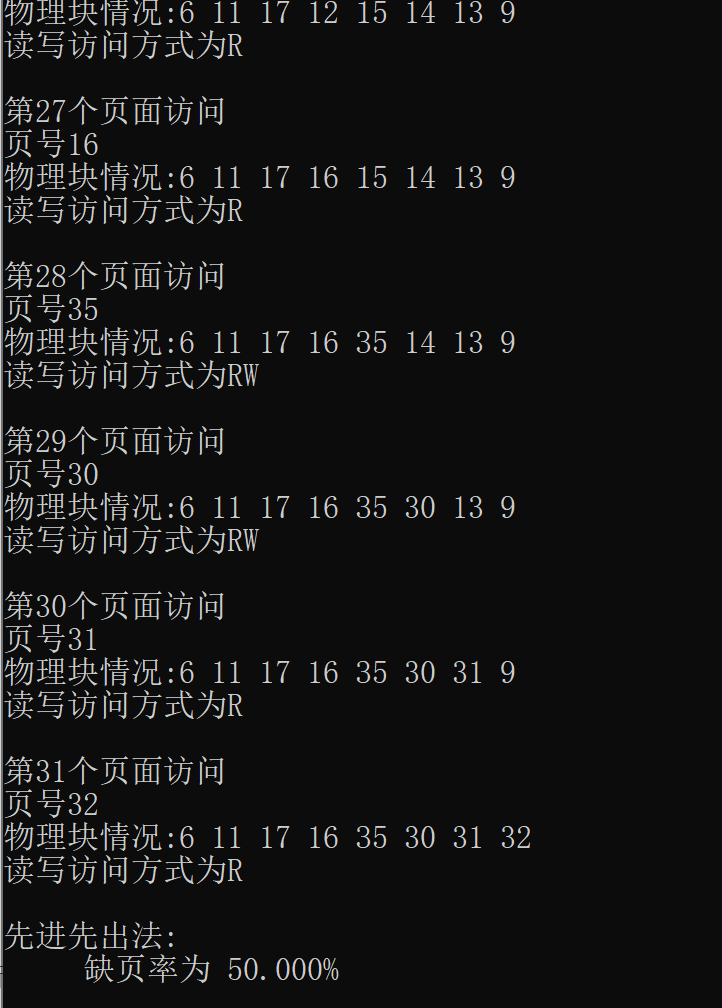
通过改变代码中宏定义的P值和F值来改变模拟的页面数和物理块数，以此得到不同的页面数和物理块数的情况下各种页面置换算法的缺页率如何。

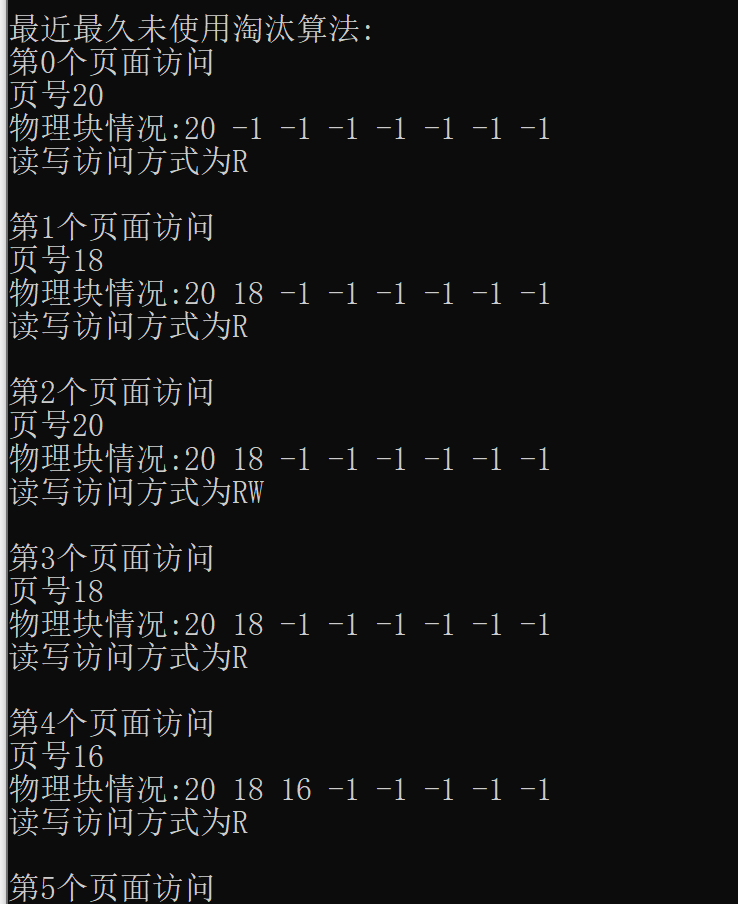
五个算法的部分测试结果如下图所示：

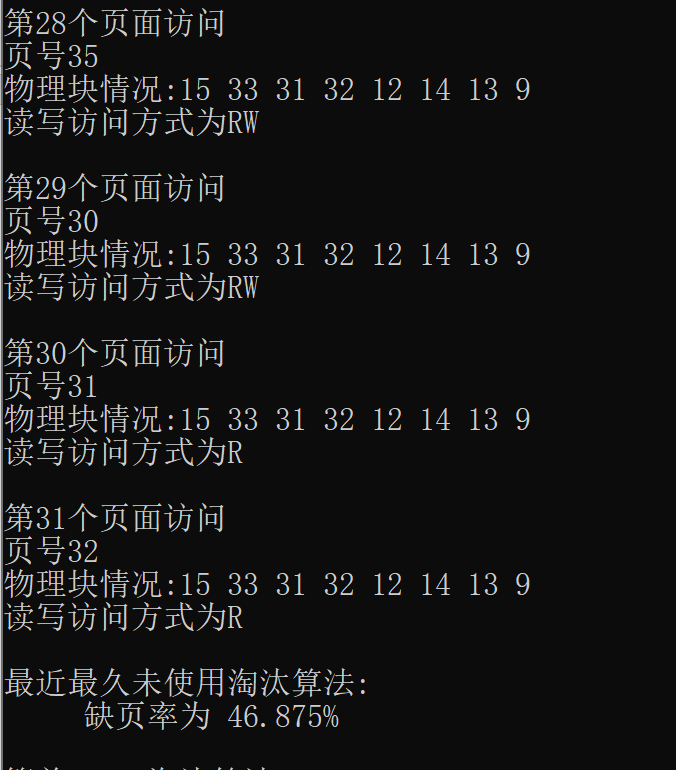


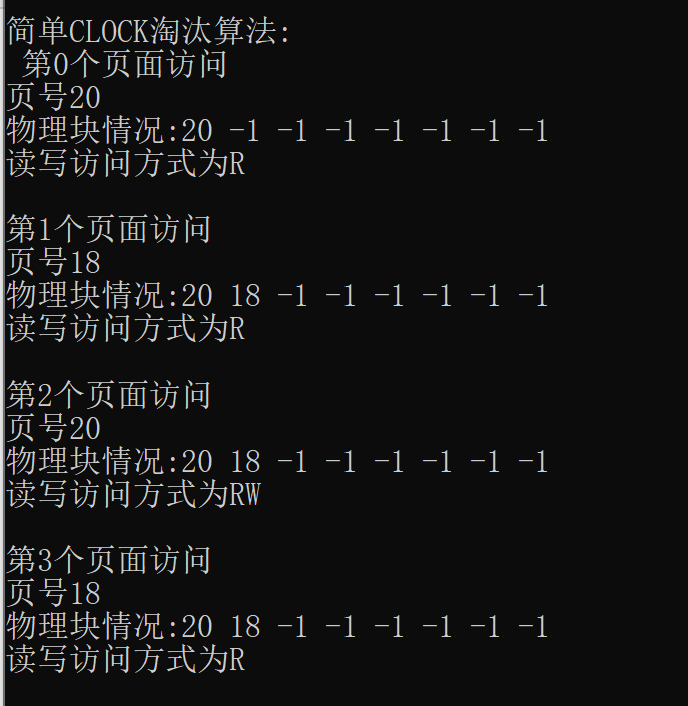


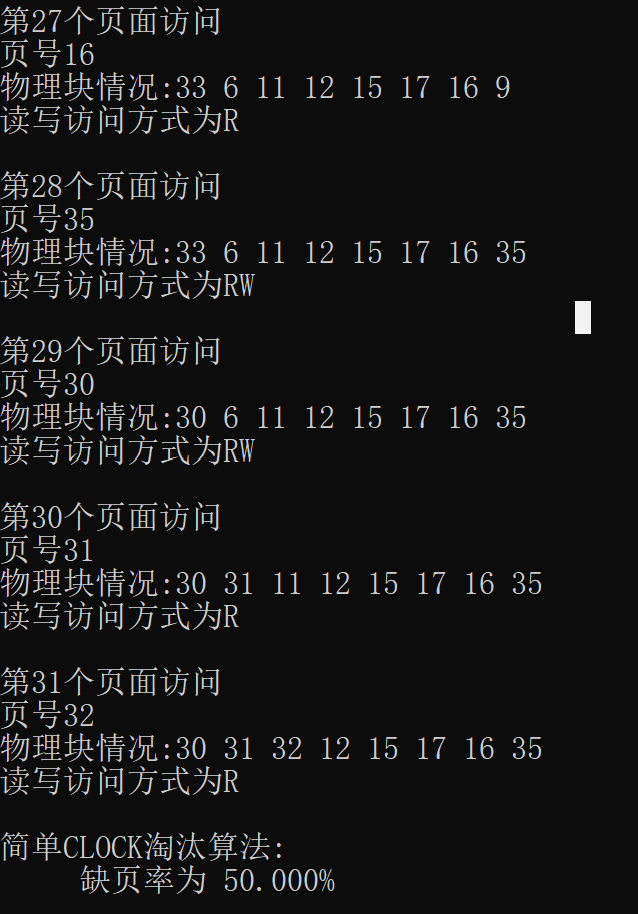


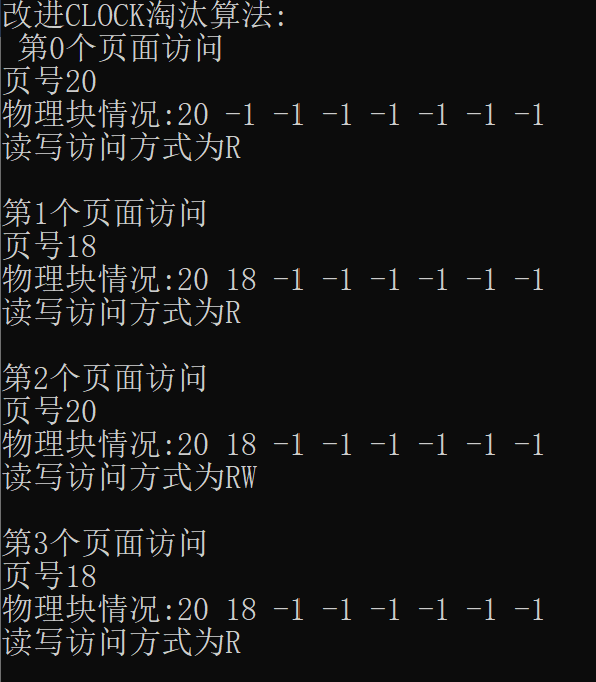


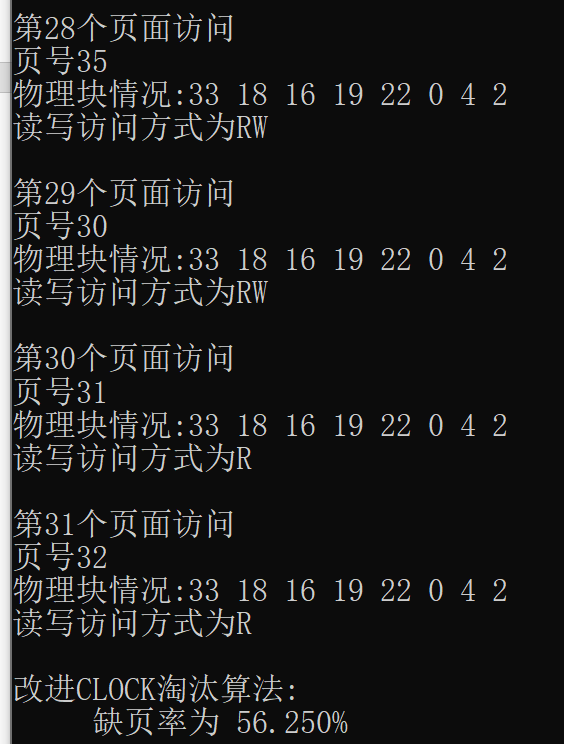












通过不断改变页面数和物理块数进行测试得到的结果如下：

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **物理块大小** | OPT | FIFO | LRU | CLOCK | LCLOCK |
| 3 | 61.80% | 81.27% | 81.27% | 81.07% | 80.93% |
| 4 | 52.60% | 73.60% | 74.13% | 74.40% | 75.93% |
| 5 | 45.73% | 66.60% | 66.53% | 69.13% | 67.53% |
| 6 | 42.47% | 63.40% | 65.00% | 67.53% | 65.60% |
| 7 | 37.13% | 55.93% | 56.60% | 65.47% | 58.13% |
| 8 | 33.07% | 51.47% | 52.27% | 62.27% | 55.20% |

可见，一般情况下，物理块数越大，各个算法的缺页率越小。

当物理块数固定为8时，改变页面总数的大小结果如下：

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **页面总数大小** | OPT | FIFO | LRU | CLOCK | LCLOCK |
| 32 | 32.25% | 48.25% | 52.25% | 62.07% | 55.93% |
| 64 | 49.13% | 61.93% | 59.60% | 72.47% | 68.13% |
| 128 | 62.47% | 75.40% | 66.00% | 78.53% | 72.60% |
| 256 | 79.53% | 81.22% | 84.33% | 89.63% | 85.53% |
| 512 | 89.92% | 92.13% | 91.29% | 92.33% | 92.93% |
| 1024 | 91.73% | 98.90% | 95.99% | 94.90% | 94.93% |

可见，页面总数越大，缺页率越大。

# 四、实验总结

通过本次实验，我更加深刻的理解了各种页面置换算法的流程，在实验过程中，代码运行时遇到了很多问题，通过自己DUBUG和参考网络上经典的代码最终成功完成了实验，在这个过程中学会了很多新的知识和技能。

# 五、源代码附录

1. #include "stdio.h"
2. #include "stdlib.h"
3. #include "time.h"
5. #define P 32//页面总数
6. #define F 8//物理块总数
7. #define RS\_MAX\_SIZE 32
9. **int** rs[RS\_MAX\_SIZE];//页面访问序列数组
10. **int** re[RS\_MAX\_SIZE];//页面读写方式数组
11. **int** rs\_size;//页面访问序列大小
13. **int** M[F];
14. **int** m\_size = 0;
16. **int** oldest\_FIFO = 0;
18. **int** queue\_LRU[F];
19. **int** queue\_size = 0;
21. **int** replace\_clock = 0;
22. **int** access\_bit[F] = { 0 };
23. **int** modify\_bit[F] = { 0 };
25. **void** CreateRS() {
26. rs\_size = 0;
27. srand((unsigned)time(NULL));
28. **int** p = rand() % P;//p：工作集起始位置
29. **int** e = 8;//e：页数
30. **int** m = 4;//m：工作集移动率
31. **double** t = rand() / (RAND\_MAX + 1.0);//t：比较值
32. **int** rs\_index = 0;
33. **while** (rs\_size + m <= RS\_MAX\_SIZE) {
34. **for** (**int** i = 0; i < m; i++) {
35. **int** cur = rand() % e + p;
36. rs[rs\_index++] = cur;
37. }
38. **double** r = rand() / (RAND\_MAX + 1.0);
39. **if** (r < t)
40. p = rand() % P;
41. **else**
42. p = (p + 1) % P;
43. rs\_size += m;
44. }
45. **for** (**int** i = 0; i < rs\_size; i++) {
46. **double** r = rand() / (RAND\_MAX + 1.0);
47. **if** (r > 0.7) {
48. re[i] = 0;
49. }
50. **else** {
51. re[i] = 1;
52. }
53. }
55. }
57. **void** print(**int** i) {
59. printf("第%d个页面访问\n", i);
60. printf("页号%d\n",  rs[i]);
61. printf("物理块情况:");
62. **for** (**int** i = 0; i < F; i++) {
63. printf("%d ",M[i]);
64. }
65. printf("\n");
66. **if**(re[i]==0)
67. printf("读写访问方式为RW\n" );
68. **if** (re[i] == 1)
69. printf("读写访问方式为R\n");
70. printf("\n");
71. }
73. **bool** Contains(**int** p) {
74. **for** (**int** i = 0; i < m\_size; i++) {
75. **if** (M[i] == p)
76. **return** **true**;
77. }
78. **return** **false**;
79. }
81. **void** OPT\_Rep(**int** i) {
82. **int** visited[F] = { 0 };
83. **for** (**int** j = 0; j < F; j++) {
84. **for** (**int** m = i + 1; m < rs\_size; m++) {
85. **if** (M[j] == rs[m]) {
86. visited[j] = m - i;
87. **break**;
88. }
89. }
90. }
91. **int** replace\_index;
92. **int** temp = 0;
93. **for** (**int** j = 0; j < F; j++) {
94. **if** (visited[j] == 0) {
95. M[j] = rs[i];
96. **return**;
97. }
98. **if** (visited[j] > temp) {
99. temp = visited[j];
100. replace\_index = j;
101. }
102. }
103. M[replace\_index] = rs[i];
104. }
106. /\*
107. Name:OPT()
108. Achieve:最佳置换算法
109. \*/
110. **void** OPT() {
111. **int** replace\_num = 0;
112. printf("最佳置换算法: \n");
113. **for** (**int** i = 0; i < rs\_size; i++) {
114. **if** (Contains(rs[i])) {
115. print(i);
116. **continue**;
117. }
118. **if** (m\_size < F) {
119. M[m\_size++] = rs[i];
120. }
121. **else** {
122. OPT\_Rep(i);
123. replace\_num++;
124. }
125. print(i);
126. }
127. printf("最佳置换算法: \n     缺页率为 %6.3f%%", 100 \* ((**float**)replace\_num / P));
128. printf("\n\n");
129. }

132. /\*
133. Name:void FIFO()
134. Achieve:先进先出法（Fisrt In First Out）
135. 如果一个数据最先进入缓存中，则应该最早被淘汰
136. 每次替换最先进入内存的页面
137. \*/
138. **void** FIFO() {
139. **int** replace\_num = 0;
140. printf("先进先出法: \n   ");
141. **for** (**int** i = 0; i < rs\_size; i++) {
142. **if** (Contains(rs[i])) {
143. print(i);
144. **continue**;
145. }
146. **if** (m\_size < F) {
147. M[m\_size++] = rs[i];
148. }
149. **else** {
150. M[oldest\_FIFO] = rs[i];
151. oldest\_FIFO = (oldest\_FIFO + 1) % F;
152. replace\_num++;
153. }
154. print(i);
155. }
156. printf("先进先出法: \n     缺页率为 %6.3f%%", 100 \* ((**float**)replace\_num / P));
157. printf("\n\n");
158. }
160. /\*
161. Name:  void LRU ()
162. Achieve: 最近最久未使用（Least Recently Used）
163. 如果一个数据在最近一段时间没有被访问到，那么在将来被访问的可能性也很小
164. 淘汰最长时间未被使用的页面
165. \*/
166. **void** LRU() {
167. **int** replace\_num = 0;
168. printf("最近最久未使用淘汰算法: \n");
169. **for** (**int** i = 0; i < rs\_size; i++) {
170. **if** (Contains(rs[i])) {
171. **for** (**int** j = 0; j < queue\_size; j++) {
172. **if** (queue\_LRU[j] == rs[i]) {
173. **for** (**int** m = j; m < queue\_size; m++)
174. queue\_LRU[m] = queue\_LRU[m + 1];
175. queue\_LRU[queue\_size - 1] = rs[i];
176. }
177. **break**;
178. }
179. print(i);
180. **continue**;
181. }
182. **if** (m\_size < F) {
183. queue\_LRU[queue\_size++] = rs[i];
184. M[m\_size++] = rs[i];
185. }
186. **else** {
187. **for** (**int** j = 0; j < F; j++) {
188. **if** (M[j] == queue\_LRU[0]) {
189. **for** (**int** m = 0; m < queue\_size; m++) {
190. queue\_LRU[m] = queue\_LRU[m + 1];
191. }
192. queue\_LRU[queue\_size - 1] = M[j];
193. M[j] = rs[i];
194. **break**;
195. }
196. }
197. replace\_num++;
198. }
199. print(i);
200. }
201. printf("最近最久未使用淘汰算法: \n     缺页率为 %6.3f%%", 100 \* ((**float**)replace\_num / P));
202. printf("\n\n");
203. }
205. /\*
206. Name:  void Clock ()
207. Achieve: 简单clock算法
208. 页面被访问后access\_bit[j]置1；若当前访问位access\_bit[j]为0，则将其置出；若当前访问位access\_bit[j]为1，则置为0
209. \*/
210. **void** Clock() {
211. **int** replace\_num = 0;
212. printf("简单CLOCK淘汰算法: \n ");
213. **for** (**int** i = 0; i < rs\_size; i++) {
214. **if** (Contains(rs[i])) {
215. **for** (**int** j = 0; j < m\_size; j++) {
216. **if** (M[j] == rs[i]) {
217. access\_bit[j] = 1;
218. }
219. }
220. print(i);
221. **continue**;
222. }
223. **if** (m\_size < F) {
224. access\_bit[m\_size] = 1;
225. M[m\_size++] = rs[i];
226. }
227. **else** {
228. **while** (**true**) {
229. **if** (access\_bit[replace\_clock] == 0) {
230. M[replace\_clock] = rs[i];
231. access\_bit[replace\_clock] = 1;
232. **break**;
233. }
234. **else** {
235. access\_bit[replace\_clock] = 0;
236. replace\_clock = (replace\_clock + 1) % F;
237. }
238. }
239. replace\_num++;
240. }
241. print(i);
242. }
243. printf("简单CLOCK淘汰算法: \n     缺页率为 %6.3f%%", 100 \* ((**float**)replace\_num / P));
244. printf("\n\n");
245. }
247. /\*
248. Name:  void MoreClock ()
249. Achieve: 改进Clock算法
250. 访问位access，修改位modify，通过至多四轮扫描找出最适合淘汰的页面
251. \*/
252. **void** MoreClock() {
253. **int** replace\_num = 0;
254. **int** flag = 0;
255. **int** temp = 1;
256. printf("改进CLOCK淘汰算法: \n ");
257. **for** (**int** i = 0; i < rs\_size; i++) {
258. **if** (Contains(rs[i])) {
259. **for** (**int** j = 0; j < m\_size; j++) {
260. **if** (M[j] == rs[i]) {
261. access\_bit[j] = 1;
262. }
263. }
264. }
265. **else** {
266. **for** (**int** j = 0; j < m\_size; j++) {
267. **if** (M[j] == rs[i]) {
268. modify\_bit[j] = 1;
269. }
270. }
271. **if** (m\_size < F) {
272. access\_bit[m\_size] = 1;
273. M[m\_size++] = rs[i];
274. }
275. **else** {
276. **while** (temp == 1) {
277. oncemore:   **if** (access\_bit[replace\_clock] == 0 && modify\_bit[replace\_clock] == 0) {//扫描循环队列，寻找A=0且M=0的第一类页面
278. M[replace\_clock] = rs[i];
279. access\_bit[replace\_clock] = 0;
280. temp = 0;
281. }
282. **else** {
283. **if** (flag) {
284. **if** (access\_bit[replace\_clock] == 0 && modify\_bit[replace\_clock] == 1) {//扫描循环队列，寻找A=0且M=0的第一类页面
285. M[replace\_clock] = rs[i];
286. access\_bit[replace\_clock] = 0;
287. temp = 0;
288. }
289. **else** {
290. access\_bit[replace\_clock] = 0;
291. **if** (replace\_clock + 1 == F) {//如果第二步也失败，即未找到第二类页面，则将指针返回到开始的位置，并将所有的访问位复0
292. replace\_clock = (replace\_clock + 1) % F;
293. **for** (**int** i = 0; i <= F; i++)
294. access\_bit[i] = 0;
295. flag = 0;
296. **goto** oncemore;
297. }
298. replace\_clock = (replace\_clock + 1) % F;
299. }
300. //flag = 0;
301. }
302. **else** {
303. **if** (replace\_clock + 1 == F)//如果还没有扫描一周
304. flag = 1;
305. replace\_clock = (replace\_clock + 1) % F;
306. }
307. }
308. }
309. replace\_num++;
310. }
311. }
312. print(i);
313. }
314. printf("改进CLOCK淘汰算法: \n     缺页率为 %6.3f%%", 100 \* ((**float**)replace\_num / P));
315. printf("\n\n");
316. }
318. **void** Clear() {
319. **for** (**int** i = 0; i < F; i++) {
320. M[i] = -1;
321. }
322. m\_size = 0;
323. }

326. **int** main() {
327. CreateRS();
328. printf("页面访问序列:\n");
329. **for** (**int** i = 0; i < rs\_size; i++)
330. printf("%d ", rs[i]);
331. printf("\n");
332. printf("页面访问序列的大小:  %d\n", rs\_size);
333. printf("\n\n");
335. OPT();
336. Clear();
338. FIFO();
339. Clear();
341. LRU();
342. Clear();
344. Clock();
345. Clear();
347. MoreClock();
348. Clear();
350. **while** (1);
351. }