实验二 存储管理

#### 实验目的

1. 通过模拟实现内存分配的伙伴算法和请求页式存储管理的几种基本页面置换算法，了解存储技术的特点。
2. 掌握虚拟存储请求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。

#### 实验内容

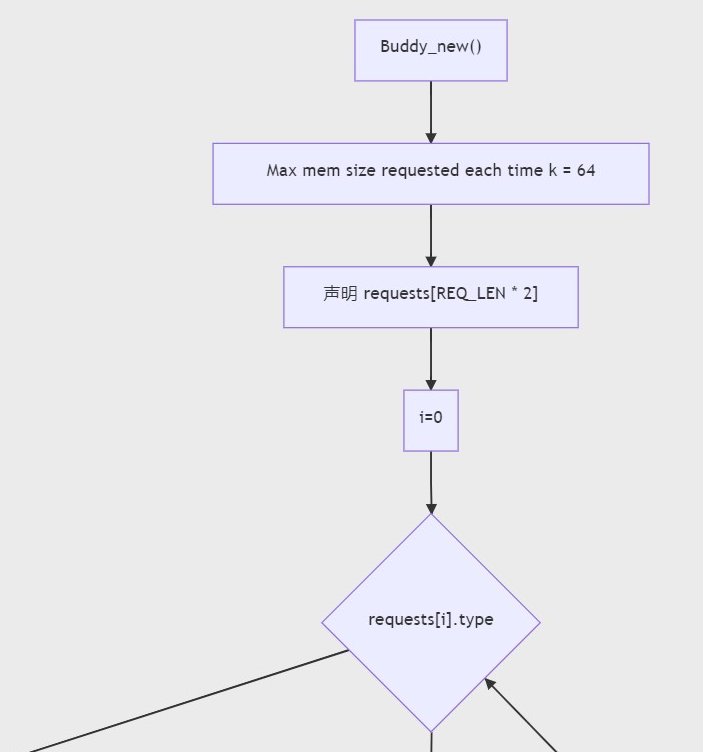
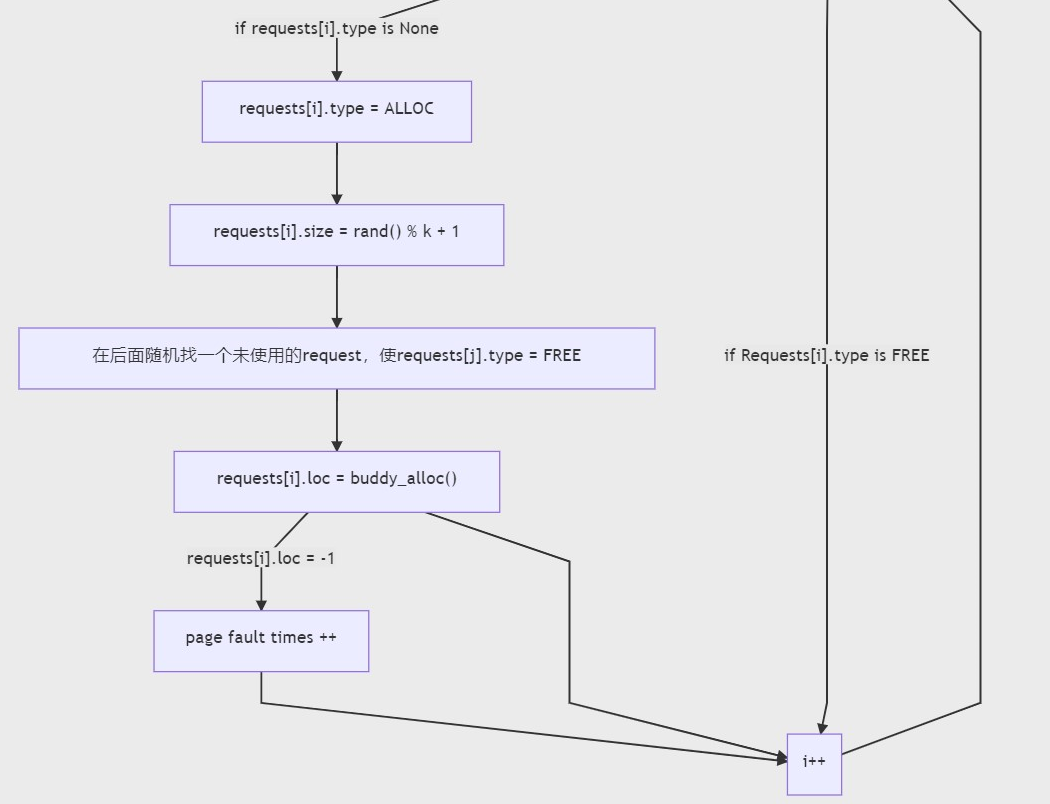
1. 基于内存管理的伙伴算法，实现内存块申请时的分配和释放后的回收，同时在回收过程中可对块进行合并。
2. 设计一个虚拟存储区和内存工作区，并使用下述算法计算访问命中率。
   * 1. 最佳置换算法（Optimal）
     2. 先进先出法（First In First Out）
     3. 最近最久未使用（Least Recently Used）
     4. 最不经常使用法（Least Frequently Used）

#### 项目要求及分析

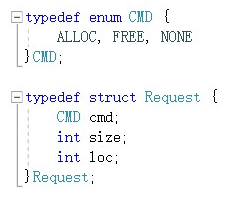
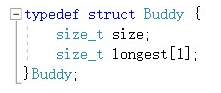
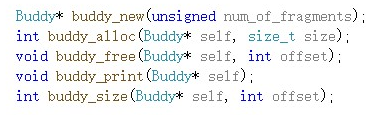
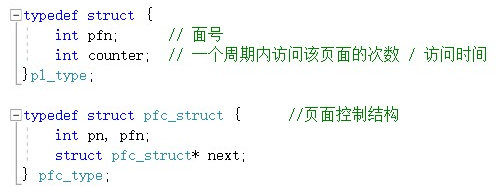
1. 基于内存管理的伙伴算法，实现内存块申请时的分配和释放后的回收，同时在回收过程中可对块进行合并。
   1. 用随机函数仿真进程进行内存申请，并且以较为随机的次序进行释放。
   2. 对其碎片进行统计，当申请分配内存失败时区分实际空间不足和由于碎片而不能满足。
   3. 按一定分布随机生成一个请求序列，对不同大小下分配成功和失败的次数进行统计。
   4. Buddy算法
      1. 伙伴算法从物理上连续的大小固定的段上进行分配。
      2. 伙伴算法的优点是可通过合并而快速地形成更大的段。
      3. 伙伴算法的一个明显区别是由于调整到下一个2的幂容易产生碎片。事实上，可能有50%的内存会因碎片而浪费。
2. 设计一个虚拟存储区和内存工作区，并使用下述算法计算访问命中率。
   * 1. 最佳置换算法（Optimal）
        1. OPT算法会置换最长时间不会使用的页，确保对于给定数量的帧会产生最低可能的页错误率。
     2. 先进先出法（First In First Out）
        1. FIFO页置换算法为每个页记录着该页调入内存的时间。当必须置换一页时，将选择最旧的页（并不需要记录调入一页的确切时间）。可以创建一个FIFO队列来管理内存中的所有页，队列中的首页将被置换，当需要调入页时，将它加到队列的尾部。
     3. 最近最久未使用（Least Recently Used）
        1. LRU置换为每个页关联该页上次使用的时间，当必须置换一页时，LRU选择最长时间没有使用的页。
        2. LRU置换有两种可行实现：
           1. 计数器：为每个页表项关联一个使用时间域，并为CPU增肌一个逻辑时钟或计数器。对每次内存引用，计数器都会增加，时钟寄存器的内容会被复制到相应页表项的使用时间域内。用这种方式就得到了每页的最近引用时间，置换具有最小时间的页。这种方案需要搜索页表以查找LRU页，且每次内存访问都要写入内存（到页表的使用时间域）。在页表改变时（因CPU调度）也必须保持时间，必须考虑时钟溢出。
           2. 栈：每当引用一个页，该页就从栈中删除并放在顶部。栈顶部总是最近使用的页，栈底部总是LRU页。由于必须从栈中部删除项，所以该栈可实现为具有头指针和尾指针的双向链表。
     4. 最不经常使用法（Least Frequently Used）
        1. 为每个页保留一个用于记录其引用次数的计数器，置换计数最小的页。
   1. 其中，命中率＝１－(页面失效次数／页地址流长度)。
   2. 试对上述算法的性能加以较各：
      1. 页面个数和命中率间的关系；
      2. 同样情况下不同置换算法的命中率比较。
   3. 本实验中主要的流程：
      1. 首先用srand( )和rand( )函数定义和产生指令序列，然后将指令序列变换成相应的页地址流，并针对不同的算法计算出相应的命中率。
      2. 实验可先从一个具体的例子出发。
         1. 通过随机数产生一个指令序列，共16K（16\*1024）条指令，每条指令长度为16B，物理页面大小为4KB,指令的地址按下述原则生成：
         2. 70%的指令是顺序执行的
         3. 10%的指令是均匀分布在前地址部分
         4. 20%的指令是均匀分布在后地址部分
         5. 对于（2）和（3），其中80%在相邻的1~2页，15%在相邻的3~4页，5%在相邻的5页以上。
      3. 具体的实施方法是（注意指令地址为16的倍数）：
         1. 从指令地址0开始;
         2. 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：
            1. 顺序执行：地址为m+16的指令
            2. 在前地址：[0,m-16]中依前面说明的概率随机选取地址（16的倍数）；
            3. 在后地址：[m+16,4096\*64-16]中依前面说明的概率随机选取地址（16的倍数）；
         3. 重复（2）直至生成16K条指令
      4. 设物理页面大小为4K，用户可用物理内存容量为第4页到第31页。
         1. 试给出不同置换算法下页面个数和命中率间的关系。
         2. 以此为基础，设置相关参数的不同值，仿真不同页面置换策略下内存容量和虚存容量参数变化时的关系曲线，并对结果加以分析。

#### 具体实现

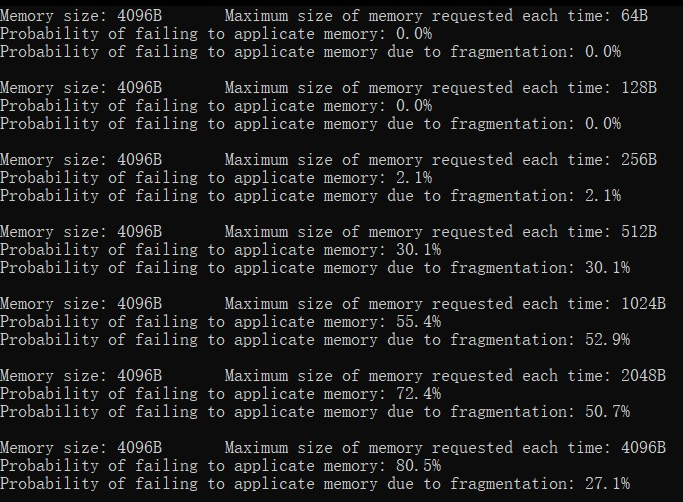
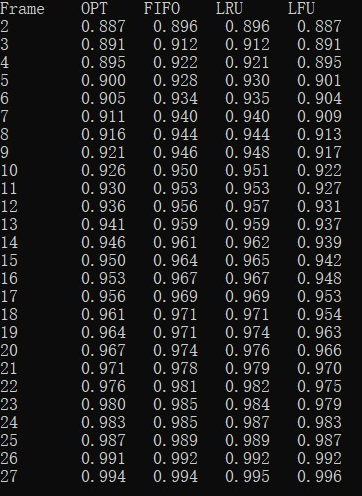
###### 流程图

1. 基于内存管理的伙伴算法，实现内存块申请时的分配和释放后的回收，同时在回收过程中可对块进行合并。
   1. 
   2. 
2. 设计一个虚拟存储区和内存工作区，并使用4种算法计算访问命中率
   1. 流程和所给要求一致

###### 数据结构定义

1. 基于内存管理的伙伴算法，实现内存块申请时的分配和释放后的回收，同时在回收过程中可对块进行合并。
   1. BuddySystem.c
      1. 枚举类型CMD和请求结构Request
         1. 
   2. Buddy.c/buddy.h
      1. 变长结构体Buddy
         1. 
      2. 函数
         1. 
2. 设计一个虚拟存储区和内存工作区，并使用4种算法计算访问命中率。
   1. PageReplacement.c
      1. 页面结构pl\_type和页面控制结构pfc\_type
      2. 

#### 调试运行结果

1. 基于内存管理的伙伴算法，实现内存块申请时的分配和释放后的回收，同时在回收过程中可对块进行合并。
   1. 
   2. 可以看到按生成的长度相同的请求序列，当内存空间相同时，内存最大请求时间相同，最大内存请求大小变大（相当于内存空间变小了）下的结果。因此，可以看到总体页错误率随内存空间变大而变小，其中因为碎片空间而分配失败的错误率随内存空间变大而先变大后变小。由此可见，当内存大小不足时，错误率的提升首先是因为碎片空间无法满足而变高，在到达错误率达到50%左右之后，更高的错误率才是因为内存空间不足。
2. 设计一个虚拟存储区和内存工作区，并使用4种算法计算访问命中率。
   1. 
   2. 可以看到不同置换算法下页面个数和命中率间的关系，在给定页面序列生成方法和参数下，4种算法表现几乎相同，FIFO和LRU效果更好一些。

#### 所遇问题及解决方法

#### 附：程序主要代码

1. 基于内存管理的伙伴算法，实现内存块申请时的分配和释放后的回收，同时在回收过程中可对块进行合并。
   1. Buddysystem.c
2. // BuddySystem.cpp : 此文件包含 "main" 函数。程序执行将在此处开始并结束。
3. //
4. #include <stdio.h>
5. #include <stdlib.h>
6. #include <time.h>
7. #include "buddy.h"
8. #define MOM\_SIZE 4096
9. #define MAX\_USE\_TIME 64
10. #define REQ\_LEN 1000
11. typedef enum CMD {
12. ALLOC, FREE, NONE
13. }CMD;
14. typedef struct Request {
15. CMD cmd;
16. int size;
17. int loc;
18. }Request;
19. int main(void) {
20. //\* seq mode
21. struct Buddy\* buddy = buddy\_new(MOM\_SIZE);
22. srand((unsigned int)time(NULL));
23. for (int k = 64; k <= MOM\_SIZE; k <<= 1) {
24. printf("Memory size: %dB \t Maximum size of memory requested each time: %dB\n", MOM\_SIZE, k);
25. int used\_size = 0; // used memory size
26. int maf\_times = 0; // times of failing to applicate memory
27. int maf\_p\_times = 0; // times of failing to applicate memory due to fragmentation
28. Request requests[REQ\_LEN \* 2]; // request of memory allocation and free
29. for (int i = 0; i < REQ\_LEN \* 2; ++i) {
30. requests[i].cmd = NONE;
31. }
32. for (int i = 0; i < REQ\_LEN \* 2; ++i) {
33. switch (requests[i].cmd) {
34. case NONE:
35. requests[i].cmd = ALLOC;
36. requests[i].size = rand() % k + 1;
37. int j = rand() % MAX\_USE\_TIME + 1;
38. if (i + j >= REQ\_LEN \* 2) {
39. j = 1;
40. }
41. while (requests[i + j].cmd != 2) {
42. if (i + j >= REQ\_LEN \* 2) {
43. j = 1;
44. } else {
45. ++j;
46. }
47. }
48. requests[i + j].cmd = FREE;
49. requests[i + j].size = requests[i].size;
50. requests[i + j].loc = requests[i].loc = buddy\_alloc(buddy, requests[i].size);
51. //printf("alloc: %d->%d\n", requests[i].size, requests[i].loc);
52. if (requests[i].loc == -1) {
53. //printf("allocation failure: %d \t used size: %d \t memory size: %d \n", requests[i].size, used\_size, MOM\_SIZE);
54. //buddy\_print(buddy);
55. ++maf\_times;
56. if (requests[i].size < MOM\_SIZE - used\_size) {
57. //printf("allocation failure: %d \t used size: %d \t memory size: %d \n", requests[i].size, used\_size, MOM\_SIZE);
58. ++maf\_p\_times;
59. }
60. } else {
61. used\_size += requests[i].size;
62. }
63. break;
64. case FREE:
65. //printf("free: %d\n", requests[i].arg);
66. if (requests[i].loc != -1) {
67. buddy\_free(buddy, requests[i].loc);
68. used\_size -= requests[i].size;
69. }
70. break;
71. default:
72. break;
73. }
74. }
75. printf("Probability of failing to applicate memory: %.1lf%%\n", (double)100 \* maf\_times / REQ\_LEN);
76. printf("Probability of failing to applicate memory due to fragmentation: %.1lf%%\n\n", (double)100 \* maf\_p\_times / REQ\_LEN);
77. }
78. //\*/
79. /\* cmd mode
80. struct Buddy\* buddy = buddy\_new(MOM\_SIZE);
81. char cmd[8];
82. int arg;
83. buddy\_print(buddy);
84. while (1) {
85. scanf\_s("%s", cmd, 8);
86. scanf\_s("%d", &arg);
87. if (strcmp(cmd, "alloc") == 0) {
88. printf("allocated@%d\n", buddy\_alloc(buddy, arg));
89. buddy\_print(buddy);
90. } else if (strcmp(cmd, "free") == 0) {
91. buddy\_free(buddy, arg);
92. buddy\_print(buddy);
93. } else if (strcmp(cmd, "size") == 0) {
94. printf("size: %d\n", buddy\_size(buddy, arg));
95. buddy\_print(buddy);
96. } else {
97. buddy\_print(buddy);
98. }
99. }
100. //\*/
101. return 0;
102. }
     1. buddy.h
103. #pragma once
104. #include <stdio.h>
105. typedef struct Buddy {
     1. size\_t size;
     2. size\_t longest[1];
106. }Buddy;
107. Buddy\* buddy\_new(unsigned num\_of\_fragments);
108. int buddy\_alloc(Buddy\* self, size\_t size);
109. void buddy\_free(Buddy\* self, int offset);
110. void buddy\_print(Buddy\* self);
111. int buddy\_size(Buddy\* self, int offset);
     1. buddy.c
112. #include <stdio.h>
113. #include <stdlib.h>
114. #include <stdbool.h>
115. #include <string.h>
116. #include "buddy.h"
117. static inline int left\_child(int index) {
118. /\* index \* 2 + 1 \*/
119. return ((index << 1) + 1);
120. }
121. static inline int right\_child(int index) {
122. /\* index \* 2 + 2 \*/
123. return ((index << 1) + 2);
124. }
125. static inline int parent(int index) {
126. /\* (index+1)/2 - 1 \*/
127. return (((index + 1) >> 1) - 1);
128. }
129. static inline bool is\_power\_of\_2(int index) {
130. return !(index & (index - 1));
131. }
132. /\* a wrapper for malloc \*/
133. static void\* d\_malloc(size\_t size) {
134. void\* tmp = NULL;
135. tmp = malloc(size);
136. if (tmp == NULL) {
137. fprintf(stderr, "my\_malloc: not enough memory, quit\n");
138. exit(EXIT\_FAILURE);
139. }
140. return tmp;
141. }
142. /\* a wrapper for free \*/
143. static void d\_free(void\* addr) {
144. free(addr);
145. }
146. static inline unsigned next\_power\_of\_2(unsigned size) {
147. /\* depend on the fact that size < 2^32 \*/
148. size -= 1;
149. size |= (size >> 1);
150. size |= (size >> 2);
151. size |= (size >> 4);
152. size |= (size >> 8);
153. size |= (size >> 16);
154. return size + 1;
155. }
156. /\*\* allocate a new buddy structure
157. \* @param num\_of\_fragments number of fragments of the memory to be managed
158. \* @return pointer to the allocated buddy structure \*/
159. Buddy\* buddy\_new(unsigned num\_of\_fragments) {
160. Buddy\* self = NULL;
161. if (num\_of\_fragments < 1 || !is\_power\_of\_2(num\_of\_fragments)) {
162. return NULL;
163. }
164. /\* alloacte an array to represent a complete binary tree \*/
165. self = (Buddy\*)d\_malloc(sizeof(Buddy) + 2 \* num\_of\_fragments \* sizeof(size\_t));
166. self->size = num\_of\_fragments;
167. size\_t node\_size = num\_of\_fragments \* 2;
168. /\* initialize \*longest\* array for buddy structure \*/
169. int iter\_end = num\_of\_fragments \* 2 - 1;
170. for (int i = 0; i < iter\_end; i++) {
171. if (is\_power\_of\_2(i + 1)) {
172. node\_size >>= 1;
173. }
174. self->longest[i] = node\_size;
175. }
176. return self;
177. }
178. /\* choose the child with smaller longest value which is still larger
179. \* than \*size\* \*/
180. unsigned choose\_better\_child(Buddy\* self, unsigned index, size\_t size) {
181. struct compound {
182. size\_t size;
183. unsigned index;
184. } children[2];
185. children[0].index = left\_child(index);
186. children[0].size = self->longest[children[0].index];
187. children[1].index = right\_child(index);
188. children[1].size = self->longest[children[1].index];
189. int min\_idx = (children[0].size <= children[1].size) ? 0 : 1;
190. if (size > children[min\_idx].size) {
191. min\_idx = 1 - min\_idx;
192. }
193. return children[min\_idx].index;
194. }
195. /\*\* allocate \*size\* from a buddy system \*self\*
196. \* @return the offset from the beginning of memory to be managed \*/
197. int buddy\_alloc(Buddy\* self, size\_t size) {
198. if (self == NULL || self->size < size) {
199. return -1;
200. }
201. size = next\_power\_of\_2(size);
202. unsigned index = 0;
203. if (self->longest[index] < size) {
204. return -1;
205. }
206. /\* search recursively for the child \*/
207. unsigned node\_size = 0;
208. for (node\_size = self->size; node\_size != size; node\_size >>= 1) {
209. /\* choose the child with smaller longest value which is still larger than \*size\* \*/
210. index = choose\_better\_child(self, index, size);
211. }
212. /\* update the \*longest\* value back \*/
213. self->longest[index] = 0;
214. int offset = (index + 1) \* node\_size - self->size;
215. while (index) {
216. index = parent(index);
217. self->longest[index] = max(self->longest[left\_child(index)], self->longest[right\_child(index)]);
218. }
219. return offset;
220. }
221. void buddy\_free(Buddy\* self, int offset) {
222. if (self == NULL || offset < 0 || offset > self->size) {
223. return;
224. }
225. /\* get the corresponding index from offset \*/
226. size\_t node\_size = 1;
227. unsigned index = offset + self->size - 1;
228. while (self->longest[index] != 0) {
229. node\_size <<= 1; /\* node\_size \*= 2; \*/
230. if (index == 0) {
231. break;
232. }
233. index = parent(index);
234. }
235. self->longest[index] = node\_size;
236. while (index) {
237. index = parent(index);
238. node\_size <<= 1;
239. size\_t left\_longest = self->longest[left\_child(index)];
240. size\_t right\_longest = self->longest[right\_child(index)];
241. if (left\_longest + right\_longest == node\_size) {
242. self->longest[index] = node\_size;
243. } else {
244. self->longest[index] = max(left\_longest, right\_longest);
245. }
246. }
247. }
248. void buddy\_print(Buddy\* self) {
249. int len = self->size << 1;
250. char cs[] = {'/', '\\'};
251. int idx = 0;
252. for (int i = 0, max\_col = len, level = 0; i < len - 1; ++i) {
253. if (is\_power\_of\_2(i + 1)) {
254. max\_col >>= 1;
255. level++;
256. idx = 0;
257. printf("\n%d(%.2d): ", level, max\_col);
258. }
259. printf("%\*d", max\_col, self->longest[i]);
260. }
261. for (int i = 0, max\_col = len, level = 0; i < len - 1; ++i) {
262. if (is\_power\_of\_2(i + 1)) {
263. max\_col >>= 1;
264. level++;
265. idx = 0;
266. printf("\n%d(%.2d): ", level, max\_col);
267. }
268. char c;
269. if (self->longest[i] > 0) {
270. c = '-';
271. } else {
272. c = cs[idx];
273. idx ^= 0x1;
274. }
275. for (int j = 0; j < max\_col; ++j) {
276. printf("%c", c);
277. }
278. }
279. printf("\n");
280. }
281. int buddy\_size(Buddy\* self, int offset) {
282. unsigned node\_size = 1;
283. unsigned index = offset + self->size - 1;
284. while (self->longest[index]) {
285. node\_size >>= 1;
286. index = parent(index);
287. }
288. return node\_size;
289. }
290. 设计一个虚拟存储区和内存工作区，并使用下述算法计算访问命中率。
     1. PageReplacement.c
291. #include <stdio.h>
292. #include <stdlib.h>
293. #include <limits.h>
294. #include <time.h>
295. #define INVALID -1
296. #define INS\_NUM 16384 // 指令数量 16K=16\*1024=16384
297. #define PHYSICAL\_PAGE\_SIZE 4096 // 物理页面大小 4KB=4096
298. #define MAX\_PAGE\_NUM 64 // 最大虚拟内存页数 64
299. #define PAGE\_NUM 28 // 物理内存页数 1-28
300. typedef struct {
301. int pfn; // 面号
302. int counter; // 一个周期内访问该页面的次数 / 访问时间
303. }pl\_type;
304. typedef struct pfc\_struct { //页面控制结构
305. int pn, pfn;
306. struct pfc\_struct\* next;
307. } pfc\_type;
308. pl\_type pl[PAGE\_NUM]; //页面结构数组
309. pfc\_type pfc[PAGE\_NUM], \* freepf\_head;
310. int page[INS\_NUM];
311. /\*
312. Name: void initialize(int total\_pf)
313. Achieve:初始化相关数据结构
314. \*/
315. void initialize(int total\_pf) {
316. for (int i = 0; i < PAGE\_NUM; ++i) {
317. pl[i].pfn = INVALID; // 置页面控制结构中的页号，页面为空
318. pl[i].counter = 0; // 页面控制结构中的访问次数为0
319. }
320. for (int i = 0; i < total\_pf - 1; ++i) {
321. pfc[i].next = &pfc[i + 1];
322. pfc[i].pfn = i; // 建立pfc[i]和pfc[i+1]之间的连接
323. }
324. pfc[total\_pf - 1].next = NULL;
325. pfc[total\_pf - 1].pfn = total\_pf - 1;
326. freepf\_head = &pfc[0]; // 页面队列的头指针为pfc[0]
327. }
328. // Optimal
329. double OPT(int total\_pf) {
330. int pf\_times = 0;
331. int maxpage, dist[PAGE\_NUM];
332. initialize(total\_pf);
333. for (int i = 0; i < INS\_NUM; i++) {
334. // 页面失效
335. if (pl[page[i]].pfn == INVALID) {
336. ++pf\_times;
337. // 无空闲页面
338. if (freepf\_head == NULL) {
339. for (int j = 0; j < PAGE\_NUM; ++j) {
340. if (pl[j].pfn == INVALID) {
341. dist[j] = 0;
342. } else {
343. dist[j] = INT\_MAX;
344. }
345. }
346. for (int j = 0; j < PAGE\_NUM; ++j) {
347. if ((pl[j].pfn != INVALID) && (dist[j] == 32767)) {
348. dist[j] = j;
349. }
350. }
351. int max = 0;
352. for (int j = 0; j < PAGE\_NUM; ++j)
353. if (max < dist[j]) {
354. max = dist[j];
355. maxpage = j;
356. }
357. freepf\_head = &pfc[pl[maxpage].pfn];
358. freepf\_head->next = NULL;
359. pl[maxpage].pfn = INVALID;
360. }
361. pl[page[i]].pfn = freepf\_head->pfn;
362. freepf\_head = freepf\_head->next;
363. }
364. }
365. return 1 - (double)pf\_times / INS\_NUM;
366. }
367. // Fisrt In First Out
368. double FIFO(int total\_pf) {
369. int pf\_times = 0;
370. pfc\_type\* busypf\_head = NULL;
371. pfc\_type\* busypf\_tail = NULL;
372. initialize(total\_pf); //初始化相关页面控制用数据结构
373. for (int i = 0; i < INS\_NUM; ++i) {
374. //页面失效
375. if (pl[page[i]].pfn == INVALID) {
376. ++pf\_times; //失效次数
377. //无空闲页面
378. if (freepf\_head == NULL && busypf\_head) {
379. pfc\_type\* tmp = busypf\_head->next;
380. pl[busypf\_head->pn].pfn = INVALID;
381. freepf\_head = busypf\_head; //释放忙页面队列的第一个页面
382. busypf\_head->next = NULL; //表明还是缺页
383. busypf\_head = tmp;
384. }
385. if (freepf\_head) {
386. pfc\_type\* tmp = freepf\_head->next;
387. freepf\_head->pn = page[i];
388. pl[page[i]].pfn = freepf\_head->pfn;
389. freepf\_head->next = NULL; //使busy的尾为null
390. if (busypf\_tail) {
391. busypf\_tail->next = freepf\_head;
392. busypf\_tail = freepf\_head;
393. } else {
394. busypf\_tail = busypf\_head = freepf\_head;
395. }
396. freepf\_head = tmp;
397. }
398. }
399. }
400. return 1 - (double)pf\_times / INS\_NUM;
401. }
402. // Least Recently Used
403. double LRU(int total\_pf) {
404. int pf\_times = 0;
405. int minj;
406. int present\_time = 1;
407. initialize(total\_pf);
408. for (int i = 0; i < INS\_NUM; ++i) {
409. //页面失效
410. if (pl[page[i]].pfn == INVALID) {
411. ++pf\_times;
412. //无空闲页面
413. if (freepf\_head == NULL) {
414. int min = INT\_MAX;//设置最大值
415. //找出time的最小值
416. for (int j = 0; j < PAGE\_NUM; ++j) {
417. if (min > pl[j].counter&& pl[j].pfn != INVALID) {
418. min = pl[j].counter;
419. minj = j;
420. }
421. }
422. freepf\_head = &pfc[pl[minj].pfn]; // 空出一个单元
423. pl[minj].pfn = INVALID;
424. pl[minj].counter = 1;
425. freepf\_head->next = NULL;
426. }
427. pl[page[i]].pfn = freepf\_head->pfn; // 有空闲页面,改为有效
428. pl[page[i]].counter = present\_time;
429. freepf\_head = freepf\_head->next; // 减少一个free 页面
430. } else {
431. pl[page[i]].counter = present\_time; // 命中则增加该单元的访问次数
432. ++present\_time;
433. }
434. }
435. return 1 - (double)pf\_times / INS\_NUM;
436. }
437. // Least Frequently Used
438. double LFU(int total\_pf) {
439. int pf\_times = 0;
440. int minj;
441. initialize(total\_pf);
442. for (int i = 0; i < INS\_NUM; ++i) {
443. //页面失效
444. if (pl[page[i]].pfn == INVALID) {
445. ++pf\_times;
446. //无空闲页面
447. if (freepf\_head == NULL) {
448. int min = INT\_MAX;
449. //获取counter的使用用频率最小的内存
450. for (int j = 0; j < PAGE\_NUM; ++j) {
451. if (min > pl[j].counter&& pl[j].pfn != INVALID) {
452. min = pl[j].counter;
453. minj = j;
454. }
455. }
456. freepf\_head = &pfc[pl[minj].pfn];
457. pl[minj].pfn = INVALID;
458. pl[minj].counter = 0;
459. freepf\_head->next = NULL;
460. }
461. pl[page[i]].pfn = freepf\_head->pfn; // 有空闲页面,改为有效
462. freepf\_head = freepf\_head->next; // 减少一个free 页面
463. }
464. pl[page[i]].counter++;
465. }
466. return 1 - (double)pf\_times / INS\_NUM;
467. }
468. int main(void) {
469. srand((int)time(NULL));
470. // 产生指令队列
471. int ins[INS\_NUM];
472. ins[0] = 0;
473. double tmp\_sum = 0;
474. for (int i = 1; i < INS\_NUM; ++i) {
475. double tmp = (double)rand() / RAND\_MAX;
476. double tmp2 = 0;
477. if (tmp < 0.7) {
478. ins[i] = ins[i - 1] + 16; // 顺序执行一条指令
479. } else if (tmp < 0.8) {
480. tmp2 = (tmp - 0.7) / 0.1;
481. if (tmp2 < 0.8) {
482. ins[i] = ins[i - 1] - PHYSICAL\_PAGE\_SIZE - 16 \* (rand() % (PHYSICAL\_PAGE\_SIZE / 16 \* 2)); // 执行前12地址指令
483. } else if (tmp2 < 0.95) {
484. ins[i] = ins[i - 1] - PHYSICAL\_PAGE\_SIZE \* 3 - 16 \* (rand() % (PHYSICAL\_PAGE\_SIZE / 16 \* 2)); // 执行前34地址指令
485. } else {
486. ins[i] = ins[i - 1] - PHYSICAL\_PAGE\_SIZE \* 5 - 16 \* (rand() % (PHYSICAL\_PAGE\_SIZE / 16 \* PAGE\_NUM)); // 执行前5+地址指令
487. }
488. } else {
489. tmp2 = (tmp - 0.8) / 0.2;
490. if (tmp2 < 0.8) {
491. ins[i] = ins[i - 1] + PHYSICAL\_PAGE\_SIZE + 16 \* rand() % (PHYSICAL\_PAGE\_SIZE / 16 \* 2); // 执行后12地址指令
492. } else if (tmp2 < 0.95) {
493. ins[i] = ins[i - 1] + PHYSICAL\_PAGE\_SIZE \* 3 + 16 \* (rand() % (PHYSICAL\_PAGE\_SIZE / 16 \* 2)); // 执行后34地址指令
494. } else {
495. ins[i] = ins[i - 1] + PHYSICAL\_PAGE\_SIZE \* 5 + 16 \* (rand() % (PHYSICAL\_PAGE\_SIZE / 16 \* PAGE\_NUM)); // 执行后5+地址指令
496. }
497. }
498. if (ins[i] < 0) {
499. ins[i] += PHYSICAL\_PAGE\_SIZE \* MAX\_PAGE\_NUM - 16;
500. } else if (ins[i] > PHYSICAL\_PAGE\_SIZE\* MAX\_PAGE\_NUM - 16) {
501. ins[i] -= PHYSICAL\_PAGE\_SIZE \* MAX\_PAGE\_NUM - 16;
502. }
503. // 将指令序列变换成页地址流
504. page[i] = ins[i] / PHYSICAL\_PAGE\_SIZE;
505. tmp\_sum += tmp;
506. //printf("%lf\t%lf\t%d\t%d\n", tmp, tmp2, ins[i], page[i]);
507. }
508. //printf("%lf\n", tmp\_sum / INS\_NUM);
509. printf("Frame \t OPT\tFIFO\tLRU\tLFU \n");
510. //用户内存工作区从2个页面到28个页面
511. for (int i = 2; i < PAGE\_NUM; ++i) {
512. printf("%d\t", i);
513. printf("%6.3lf\t", OPT(i));
514. printf("%6.3lf\t", FIFO(i));
515. printf("%6.3lf\t", LRU(i));
516. printf("%6.3lf\n", LFU(i));
517. }
518. return 0;
519. }