河南工业大学 操作系统原理 实验报告

班级：软件 1305 班　学号：201316920311　姓名：田劲锋　指导老师：刘扬　日期：2015 年 6 月 11 日

实验 5　页式存储管理的页面置换算法模拟

1. 实验步骤

1. 以下是page.c的源代码，注释已详细给出：

Listing 1: page.c

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4  #include <limits.h>
5  #include <stdbool.h>
6
7  #define MEMORY_SIZE 32 /* 内存最大大小(K) */
8  #define PER_K_INSTS 10 /* 每K存放指令数 */
9
10 typedef struct PAGE { /* 虚拟内存页面结构 */
11     int id; /* 虚拟页面编号 */
12     int number; /* 实际页面编号 */
13     int count; /* 页面访问次数 */
14     int time; /* 页面访问时间 */
15 } page_t;
16 page_t pages[MEMORY_SIZE]; /* 虚拟内存页面 */
17
18 typedef struct PAGE_CTL { /* 实际内存页面结构 */
19     int id; /* 虚拟页面编号 */
20     int number; /* 实际页面编号 */
21     struct PAGE_CTL* next; /* 下个实际内存页面地址 */
22 } page_ctl;
23 page_ctl pager[MEMORY_SIZE]; /* 实际内存页面 */
24 page_ctl* free_head; /* 空闲队列头 */
25 page_ctl* busy_head; /* 忙碌队列头 */
26 page_ctl* busy_tail; /* 忙碌队列尾 */
27
28 #define INSTRUCTION_NUM 320
29
30 typedef struct ADDR_STREAM { /* 页地址流结构 */
31     int p; /* 第几页 */
32     int n; /* 指令 */
33 } addr_stream;
34 addr_stream addrs[INSTRUCTION_NUM]; /* 指令页地址流 */
35 int instructions[INSTRUCTION_NUM]; /* 指令地址 */
36
37 int init(int mem_size)
38 {
39     int i;
40     for (i = 0; i < MEMORY_SIZE; i++) {
41         pages[i].id = i;
42         pages[i].number = -1; /* 虚存为空 */
43         pages[i].count = 0;
44         pages[i].time = -1;
45     }
46     int n = mem_size - 1; /* 实存 */
47     for (i = 0; i < n; i++) {
48         pager[i].next = &pager[i + 1]; /* 连接前后页面 */
49         pager[i].number = i; /* 指明实存编号 */
50     }
51     pager[n].next = NULL;
52     pager[n].number = n;
```

```
53      free_head = &pager[0]; /* 空闲队列 */
54      return 0;
55  }
56
57  double FIFO(int mem_size)
58  {
59      int loss_effects = 0; /* 失效次数 */
60      init(mem_size); /* 初始化 */
61      busy_head = busy_tail = NULL;
62      int i;
63      for (i = 0; i < INSTRUCTION_NUM; i++) {
64          /* 执行每个指令，如果指令未命中 */
65          if (pages[addrs[i].p].number == -1) {
66              loss_effects++; /* 页面失效 */
67              if (free_head == NULL) { /* 没有空闲页面 */
68                  page_ctl* p = busy_head->next;
69                  pages[busy_head->id].number = -1;
70                  free_head = busy_head; /* 先进先出，弹出第一个忙页面 */
71                  free_head->next = NULL;
72                  busy_head = p;
73              }
74              page_ctl* p = free_head->next; /* 向空闲队列压入新页面 */
75              free_head->next = NULL;
76              free_head->id = addrs[i].p;
77              pages[addrs[i].p].number = free_head->number;
78              if (busy_tail == NULL) { /* 忙队列尾 */
79                  busy_head = busy_tail = free_head;
80              } else { /* 去掉一个空闲页面 */
81                  busy_tail->next = free_head;
82                  busy_tail = free_head;
83              }
84              free_head = p;
85          }
86      }
87      /* 计算命中率 */
88      return (100 - loss_effects * 100.0 / INSTRUCTION_NUM);
89  }
90
91  double LRU(int mem_size)
92  {
93      int loss_effects = 0; /* 失效次数 */
94      init(mem_size); /* 初始化 */
95      int now = 0;
96      int min, index;
97      int i, j;
98      for (i = 0; i < INSTRUCTION_NUM; i++) {
99          /* 执行每个指令，如果指令未命中 */
100         if (pages[addrs[i].p].number == -1) {
101             loss_effects++; /* 页面失效 */
102             if (free_head == NULL) { /* 没有空闲页面 */
103                 min = INT_MAX;
104                 /* 找访问时间最远的页面 */
105                 for (j = 0; j < MEMORY_SIZE; j++) {
106                     if (min > pages[j].time && pages[j].number != -1) {
107                         min = pages[j].time;
108                         index = j;
109                     }
110                 }
111                 free_head = &pager[pages[index].number]; /* 弹出该页面 */
112                 pages[index].number = -1;
113                 pages[index].time = -1;
114                 free_head->next = NULL;
115             }
116             pages[addrs[i].p].number = free_head->number; /* 压入新页面 */
117             pages[addrs[i].p].time = now;
118             free_head = free_head->next; /* 去掉一个空闲页面 */
```

```
119             } else { /* 如果命中，记录此时访问了该页面 */
120                 pages[addrs[i].p].time = now;
121             }
122             now++;
123         }
124         /* 计算命中率 */
125         return (100 - loss_effects * 100.0 / INSTRUCTION_NUM);
126     }
127
128     int main(int argc, char* argv[])
129     {
130         int i, m;
131
132         /* 生成指令序列 */
133         srand((int)getpid());
134         for (i = 0; i < INSTRUCTION_NUM;) {
135             /* 在[0,319]的指令地址之间随机选取一起点m */
136             m = (int)rand() % INSTRUCTION_NUM;
137             /* 顺序执行一条指令，即执行地址为m+1的指令 */
138             instructions[i++] = m + 1;
139             /* 在前地址[0,m+1]中随机选取一条指令并执行，该指令的地址为m' */
140             instructions[i++] = m = (int)rand() % (m + 2);
141             /* 顺序执行一条指令，其地址为m'+1的指令 */
142             instructions[i++] = m + 1;
143             /* 在后地址[m'+2,319]中随机选取一条指令并执行 */
144             instructions[i++] = ((int)rand() + (m + 2)) % INSTRUCTION_NUM;
145         }
146
147         /* 转换为页地址流 */
148         for (i = 0; i < INSTRUCTION_NUM; i++) {
149             /* 按每K存放PER_K_INSTS条指令排列虚存地址 */
150             addrs[i].p = instructions[i] / PER_K_INSTS;
151             addrs[i].n = instructions[i] % PER_K_INSTS;
152         }
153
154         /* 分配内存容量从4K循环到32K */
155         for (i = 4; i <= 32; i++) {
156             printf("%2dK\tFIFO: %4.2lf%%\tLRU: %4.2lf%%\n", i, FIFO(i), LRU(i));
157         }
158
159         return 0;
160     }
```

编译并执行该程序：

```
$ cc -Wall page.c -o page
$ ./page > 1
```

得到输出结果如下：

```
 4K      FIFO: 35.94%      LRU: 36.88%
 5K      FIFO: 39.06%      LRU: 38.75%
 6K      FIFO: 41.56%      LRU: 40.94%
 7K      FIFO: 44.06%      LRU: 43.75%
 8K      FIFO: 45.31%      LRU: 45.00%
 9K      FIFO: 46.56%      LRU: 47.81%
10K      FIFO: 47.19%      LRU: 49.38%
11K      FIFO: 51.25%      LRU: 50.62%
12K      FIFO: 51.25%      LRU: 52.50%
13K      FIFO: 53.75%      LRU: 54.38%
14K      FIFO: 55.94%      LRU: 55.94%
15K      FIFO: 57.50%      LRU: 57.81%
16K      FIFO: 59.06%      LRU: 60.31%
17K      FIFO: 61.56%      LRU: 62.19%
```

```
18K         FIFO: 64.38%        LRU: 63.75%
19K         FIFO: 66.88%        LRU: 64.69%
20K         FIFO: 72.81%        LRU: 68.12%
21K         FIFO: 74.06%        LRU: 69.06%
22K         FIFO: 75.31%        LRU: 71.56%
23K         FIFO: 75.31%        LRU: 74.38%
24K         FIFO: 77.19%        LRU: 76.88%
25K         FIFO: 77.19%        LRU: 79.69%
26K         FIFO: 78.12%        LRU: 81.25%
27K         FIFO: 80.00%        LRU: 83.75%
28K         FIFO: 84.38%        LRU: 84.69%
29K         FIFO: 84.38%        LRU: 86.56%
30K         FIFO: 88.44%        LRU: 87.50%
31K         FIFO: 88.75%        LRU: 89.38%
32K         FIFO: 90.00%        LRU: 90.00%
```

可以看到两种算法的差距并不是很大，相对的来说，LRU算法在内存较大的情况下会有比FIFO更好一些的效果。

2. **先进先出**算法（First-In, First-Out, FIFO）是一个经典的队列实现，它也是一种低开销的页面置换算法。我们维护一个链表，链表中记录了所有位于内存中的虚拟页面，从链表的排列顺序看，链表头部页面的驻留时间最长，链表尾部页面的驻留时间最短。当发生一个缺页中断时，将链表头部的页面淘汰出局，并把新的页面添加到链表尾部。FIFO算法的性能不是很好，被它淘汰出去的页面可能是经常要访问的页面。基于这个原因，FIFO算法很少被单独使用。

   最近最久未使用算法（Least Recently Used, LRU）的基本思路是，当一个缺页中断发生时，从内存中选择最久未被使用的那个页面，把它淘汰出局。这种算法实质上是对最优页面置换算法的一个近似，理论依据就是程序的局部性原理。LRU算法的策略就是根据程序的局部性原理，利用过去的、已知的页面访问情况，来预测将来的情况。LRU的实现开销比较大，改程序中的模拟实际上效率是比较差的。一种好的解决方案是**最不经常使用**算法（Not Frequently Used, NFU）的变体——老化算法，具体可以参考《操作系统设计与实现（第三版）》第273–275页的内容。