

# 操作系统课程设计 Project 8

## Designing a Virtual Memory Manager

姓名：郭倩昀

班级：F1903303

学号：519021910095

Email: guoqianyun@sjtu.edu.cn

2021 年 5 月 23 日

### 目录

<b>1</b>	<b>Designing a Virtual Memory Manager</b>	<b>2</b>
1.1	实验内容与目标 . . . . .	2
1.2	实验过程及步骤 . . . . .	2
1.3	实验代码 . . . . .	3
1.4	实验测试 . . . . .	9
<b>2</b>	<b>Conclusion</b>	<b>10</b>
2.1	问题与解决方案 . . . . .	10
2.2	实验心得 . . . . .	10

# 1 Designing a Virtual Memory Manager

## 1.1 实验内容与目标

本实验需要利用 C 语言设计虚拟内存管理系统，使得可以根据逻辑地址正确读取数据并正确管护物理内存，页表，TLB，主要可以分为一下几个部分。

- 管理物理内存
- 管理 TLB
- 管理页表

## 1.2 实验过程及步骤

- 定义基本参数

根据书本指示定义页面数，页面大小，TLB 大小，内存帧数和内存帧大小。其中内存帧数与页面数相等为 256 时不需要页面替换算法，但是如果设计内存帧数为 128，则需要设计页面替换算法，这里选择使用 LRU 置换策略进行页面替换。

- 管理物理内存

管理物理内存需要支持的操作有维护空闲帧列表，初始化物理内存（包括初始化空闲帧列表），将一个页面调入物理内存（如果无空闲帧，则要用 LRU 算法选取替换帧替换），根据帧数和 offset 访问物理内存，清空内存等操作。为了支持 LRU 页面置换算法，需要维护帧的 LRU 信息，每次访问都要更新 LRU 信息。

- 维护空闲帧列表主要用 `init_empty_frame_list` 函数初始化空闲帧列表（插入内存帧数的空闲帧列表），`get_empty_frame` 函数从空闲帧列表中找出一个空闲帧返回 frame num（未找到则返回-1），`clean_empty_frame_list` 函数释放所有的空闲帧。
- 初始化物理内存使用 `init_memory` 函数，打开后备存储，调用 `init_empty_frame_list` 初始化内存帧，同时为了支持 LRU 置换，初始化帧的 LRU 信息。
- 将一个页面调入物理内存使用 `add_page_into_memory` 函数，先根据 page number 读取后备存储的数据，寻找一个空闲帧，（若未找到则通过 LRU 置换一个帧，置换出的帧需要修改页表的有效-无效位），将数据放入空闲帧并更新 LRU 信息，返回 frame number。
- 访问物理内存使用 `access_memory` 函数根据帧数和 offset 访问直接读取，但要更新 LRU 信息。
- 清空内存使用 `clean_memory` 函数，调用 `clean_empty_frame_list` 函数释放所有的空闲帧并关闭打开的后备存储文件。

- 管理 TLB

管理 TLB 需要支持的操作有初始化 TLB，TLB 查找通过 page number 获取 frame number（命中时候增加 TLB 命中计数），增加 TLB 条目（可能需要 LRU 置换），删除 TLB 条目。为了支持 LRU 置换算法，需要维护 TLB 条目的 LRU 信息，每次访问都要更新 LRU 信息。

- 初始化 TLB 使用 `init_TLB` 函数，将 TLB 命中计数置零，所有条目信息置零。
- TLB 查找使用 `get_TLB_frame_id` 函数，根据所给的 page number 查找条目，命中则增加 TLB 命中计数，同时更新 LRU 信息，否则返回-1 表示未命中。

- 增加 TLB 条目使用 `add_TLB_item` 函数，先插着一个空闲条目空间，若未找到则使用 LRU 置换，写入待加入的条目，并更新 LRU 信息。
- 删除 TLB 条目使用 `delete_TLB_item` 函数，查找到待删除的条目将信息置零并更新 LRU 信息。
- 管理页表  
管理页表需要支持的操作有初始化页表，通过 `page number` 获取 `frame number`，在物理内存中删除某一帧的时候更新页表的有效-无效位。
  - 初始化页表使用 `init_page_table` 函数，将缺页计数置零，页表条目信息置零。
  - 通过 `page number` 获取 `frame number`，先 TLB 查找，若未命中则进行页表查找，若都未命中则记录缺页信息，然后调用 `add_page_into_memory` 函数将一个页面调入物理内存，更新页表的有效-无效位，同时调用 `add_TLB_item` 增加到 TLB 条目中。
  - `invalid_page_table_item` 函数功能为在物理内存中删除某一帧的时候将页表中该页的有效-无效位置零表示该页目前不在物理内存中。
- LRU 置换算法的实现  
本项目中 TLB 和帧的管理都采用了 LRU 置换算法。主要实现思想是，未被访问过的条目的 LRU 信息置零，访问过的条目的 LRU 信息根据访问先后排序，最近访问的为 1，之后一次排序。当要选取牺牲条目的时候，LRU 等于条目数的那一条目就被选为牺牲者，将新的条目换进并把 LRU 信息记为 1，其他条目的 LRU 信息都增加 1。当重新访问一个条目的时候，LRU 数据小于该条目的都增加 1，并将该条目的 LRU 记为 1。
- `main()` 函数设计  
`main()` 函数主要功能是连接先前设计的所有函数，发挥各部分的功能。首先打开需要寻址的 `addresses.txt` 文件，打开输出寻址结果的 `answer.txt` 文件，初始化物理内存，TLB 和页表，然后对每一个读入的地址按位取出 `page number` 和 `offset`，并调用函数 `get_frame_id` 获取 `frame number`，然后根据获取的 `frame number` 和 `offset` 调用 `access_memory` 读取结果。最后输出映射的物理内存以及读取的结果，打印 TLB 命中率以及缺页率。最后需要清空物理内存并关闭之前打开的文件。

### 1.3 实验代码

`vmm.c`

```

1 # include <stdio.h>
2 # include <stdlib.h>
3 # include <string.h>
4
5 # define PAGE_NUM 256
6 # define PAGE_SIZE 256
7 # define TLB_SIZE 16
8 # define FRAME_SIZE 256
9 # define FRAME_NUM 256
10
11 //Empty Frame List
12 typedef struct empty_frame_node {
13     int frame_id;

```

```

14     struct empty_frame_node *next;
15 }empty_frame_node;
16
17 empty_frame_node *head = NULL;
18 empty_frame_node *tail = NULL;
19
20 char memory[FRAME_NUM * FRAME_SIZE];
21 int frame_LRU[FRAME_NUM];
22 char buf[FRAME_SIZE];
23 FILE *fp_bs;           //for backing store
24
25 int TLB_page[TLB_SIZE];
26 int TLB_frame[TLB_SIZE];
27 int TLB_LRU[TLB_SIZE];
28 int TLB_hit_cnt;       //TLB hit count
29
30 int page_table[PAGE_NUM];
31 int page_table_vi[PAGE_NUM]; //valid-invalid
32 int page_fault_cnt;         //page fault count
33 //for empty frame list
34 void init_empty_frame_list();
35 int get_empty_frame();
36 void clean_empty_frame_list();
37 //for memory
38 void init_memory();
39 int add_page_into_memory(int page_id);
40 char access_memory(int frame_id, int offset);
41 void clean_memory();
42 void update_frame_LRU(int frame_id);
43 //for TLB
44 void init_TLB();
45 int get_TLB_frame_id(int page_id);
46 void add_TLB_item(int page_num, int frame_num);
47 void update_TLB_LRU(int dex);
48 void delete_TLB_item(int page_id, int frame_id);
49 //for page table
50 void init_page_table();
51 int get_frame_id(int page_id);
52 void invalid_page_table_item(int frame_id);
53
54 //Initialize the empty frame list
55 void init_empty_frame_list() {
56     for (int i = 0; i < FRAME_NUM; ++ i) //Add each empty frame
57     {
58         if (head == NULL && tail == NULL)//no empty frame
59         {
60             tail = (empty_frame_node *) malloc (sizeof(empty_frame_node));
61             tail -> frame_id = i;
62             tail -> next = NULL;
63             head = tail;
64         }
65         else//add to tail
66         {
67             tail->next = (empty_frame_node *)malloc(sizeof(empty_frame_node));
68             tail->next->frame_id = i;
69             tail->next->next = NULL;
70             tail = tail->next;

```

```

71     }
72 }
73 }
74
75 //Get an empty frame from the empty frame list
76 int get_empty_frame() {
77     int frame_id;
78     //no empty frame
79     if (head == NULL && tail == NULL) return -1;
80     //one empty frame
81     if (head == tail) {
82         frame_id = head -> frame_id;
83         free(head);
84         head = tail = NULL;
85         return frame_id;
86     }
87     //more than one
88     empty_frame_node *p=head;
89     frame_id = head -> frame_id;
90     head = head -> next;
91     free(p);
92     return frame_id;
93 }
94
95 // Clean the empty frame list.
96 void clean_empty_frame_list() {
97     if (head == NULL && tail == NULL) return;
98     struct empty_frame_node *p;
99     while (head != tail)
100     {
101         p = head;
102         head = head -> next;
103         free(p);
104     }
105     free(head);
106     head = tail = NULL;
107 }
108
109 //Initialize memory
110 void init_memory() {
111     fp_bs = fopen("BACKING_STORE.bin", "rb");
112     if (fp_bs == NULL)
113     {
114         fprintf(stderr, " ERROR: Open backing store file error\n");
115         exit(1);
116     }
117     //Initialize the empty frame list
118     init_empty_frame_list();
119     //initialize LRU record
120     for (int i = 0; i < FRAME_NUM; ++ i)
121         frame_LRU[i] = 0;
122 }
123
124 int add_page_into_memory(int page_id) {
125     fseek(fp_bs, page_id * FRAME_SIZE, SEEK_SET);
126     fread(buf, sizeof(char), FRAME_SIZE, fp_bs); //read data to buffer
127     int frame_id = get_empty_frame(); //get an empty frame for the page

```

```

128     if (frame_id == -1)                                //not found LRU replacement
129     {
130         for (int i = 0; i < FRAME_NUM; ++ i)
131             if (frame_LRU[i] == FRAME_NUM) //for replacement
132             {
133                 frame_id = i;
134                 break;
135             }
136         invalid_page_table_item(frame_id);
137     }
138     for (int i = 0; i < FRAME_SIZE; ++ i) //put data into memory
139     {
140         memory[frame_id * FRAME_SIZE + i] = buf[i];
141     }
142     //update_frame_LRU(frame_id);
143     for (int i = 0; i < FRAME_NUM; ++ i) //update frame LRU record
144     {
145         if (frame_LRU[i] > 0)
146             frame_LRU[i]++;
147     }
148     frame_LRU[frame_id] = 1; //latest access
149     return frame_id;
150 }
151
152 void update_frame_LRU(int frame_id)
153 {
154     for (int i = 0; i < FRAME_NUM; ++ i) //update frame LRU record
155         if (frame_LRU[i] > 0 && frame_LRU[i] < frame_LRU[frame_id])
156             ++ frame_LRU[i];
157     frame_LRU[frame_id] = 1;
158 }
159
160 char access_memory(int frame_id, int offset)
161 {
162     char rst = memory[frame_id * FRAME_SIZE + offset];
163     update_frame_LRU(frame_id);
164     return rst;
165 }
166
167 void clean_memory()
168 {
169     clean_empty_frame_list();
170     fclose(fp_bs);
171 }
172 //initialize TLB
173 void init_TLB()
174 {
175     TLB_hit_cnt = 0;
176     for (int i = 0; i < TLB_SIZE; ++ i)
177     {
178         TLB_page[i] = 0;
179         TLB_frame[i] = 0;
180         TLB_LRU[i] = 0;
181     }
182 }
183
184 int get_TLB_frame_id(int page_id)

```

```

185 {
186     int dex = -1;
187     for (int i = 0; i < TLB_SIZE; ++ i)
188         if (TLB_LRU[i] > 0 && TLB_page[i] == page_id)
189             {
190                 dex = i;
191                 break;
192             }
193     if (dex == -1) return -1;    // TLB not hit.
194     ++ TLB_hit_cnt;            // TLB hit.
195     update_TLB_LRU(dex);      //update TLB LRU record
196     return TLB_frame[dex];
197 }
198
199 void update_TLB_LRU(int dex)
200 {
201     for (int i = 0; i < TLB_SIZE; ++ i) //update TLB LRU record
202         if (TLB_LRU[i] > 0 && TLB_LRU[i] < TLB_LRU[dex])
203             ++ TLB_LRU[i];
204     TLB_LRU[dex] = 1;
205 }
206
207 // add TLB
208 void add_TLB_item(int page_id, int frame_id)
209 {
210     int dex = -1;
211     for (int i = 0; i < TLB_SIZE; ++ i)
212         if (TLB_LRU[i] == 0) {
213             dex = i;
214             break;
215         }
216     if (dex == -1)    // LRU replacement.
217     {
218         for (int i = 0; i < TLB_SIZE; ++ i)
219             if (TLB_LRU[i] == TLB_SIZE) {
220                 dex = i;
221                 break;
222             }
223     }
224
225     TLB_page[dex] = page_id;
226     TLB_frame[dex] = frame_id;
227     for (int i = 0; i < TLB_SIZE; ++ i) //update TLB LRU record
228         if (TLB_LRU[i] > 0) ++ TLB_LRU[i];
229     TLB_LRU[dex] = 1;
230 }
231
232 // Delete TLB
233 void delete_TLB_item(int page_id, int frame_id)
234 {
235     int dex = -1;
236     for (int i = 0; i < TLB_SIZE; ++ i)
237         if (TLB_LRU[i] && TLB_page[i] == page_id && TLB_frame[i] == frame_id) {
238             dex = i;
239             break;
240         }
241     if (dex == -1) return;

```

```

242     for (int i = 0; i < TLB_SIZE; ++ i) //update TLB LRU record
243         if (TLB_LRU[i] > TLB_LRU[dex]) -- TLB_LRU[i];
244     TLB_LRU[dex] = 0; //empty
245 }
246
247 //initialize page table
248 void init_page_table() {
249     page_fault_cnt = 0;
250     for (int i = 0; i < PAGE_NUM; ++ i) {
251         page_table[i] = 0;
252         page_table_vi[i] = 0;
253     }
254 }
255
256 int get_frame_id(int page_id)
257 {
258     if (page_id < 0 || page_id >= PAGE_NUM) return -1;
259
260     //TLB
261     int TLB_frame_id = get_TLB_frame_id(page_id);
262     if (TLB_frame_id != -1) return TLB_frame_id;
263
264     //TLB NOT HIT -> page table
265     if (page_table_vi[page_id] == 1) //page table hit
266     {
267         add_TLB_item(page_id, page_table[page_id]);
268         return page_table[page_id];
269     }
270     else // Page fault.
271     {
272         page_fault_cnt++;
273         page_table[page_id] = add_page_into_memory(page_id);
274         page_table_vi[page_id] = 1; //vailid
275         add_TLB_item(page_id, page_table[page_id]);
276         return page_table[page_id];
277     }
278 }
279
280 void invalid_page_table_item(int frame_id)
281 {
282     int page_id = -1;
283     for (int i = 0; i < PAGE_NUM; ++ i)
284         if (page_table_vi[i] && page_table[i] == frame_id) {
285             page_id = i;
286             break;
287         }
288     if (page_id == -1) {
289         fprintf(stderr, " ERROR: PAGE_ID Error\n");
290         exit(1);
291     }
292     page_table_vi[page_id] = 0;
293     delete_TLB_item(page_id, frame_id);
294 }
295
296 int main(int argc, char *argv[]) {
297     if (argc != 2)
298     {

```



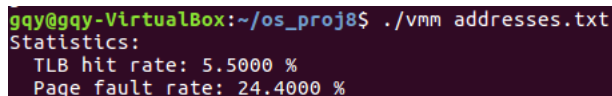
```

299     fprintf(stderr, "  ERROR: Invalid input\n");
300     return 1;
301 }
302
303 FILE *fp_in = fopen(argv[1], "r");
304 if(fp_in == NULL)
305 {
306     fprintf(stderr, "  ERROR: File Error\n");
307     return 1;
308 }
309
310 FILE *fp_out = fopen("answer.txt", "w");
311 if (fp_out == NULL)
312 {
313     fprintf(stderr, "  ERROR: File Error\n");
314     return 1;
315 }
316
317 init_page_table();
318 init_TLB();
319 init_memory();
320
321 int addr, page_id, offset, frame_id, res, cnt = 0;
322 while(~fscanf(fp_in, "%d", &addr)) {
323     ++ cnt;
324     addr = addr & 0x0000ffff;
325     offset = addr & 0x000000ff;
326     page_id = (addr >> 8) & 0x000000ff;
327     frame_id = get_frame_id(page_id);
328     res = (int) access_memory(frame_id, offset);
329     fprintf(fp_out, "Virtual address: %d Physical address: %d Value: %d\n", addr,
330 (frame_id << 8) + offset, res);
331 }
332
333 fprintf(stdout, "Statistics:\n  TLB hit rate: %.4f %%\n  Page fault rate: %.4f %%\n",
334 100.0 * TLB_hit_cnt / cnt, 100.0 * page_fault_cnt / cnt);
335
336 clean_memory();
337 fclose(fp_in);
338 fclose(fp_out);
339 return 0;
340 }

```

## 1.4 实验测试

- vmm 测试 (FRAME NUM=256) (图 1)



```

gqy@gqy-VirtualBox:~/os_proj8$ ./vmm addresses.txt
Statistics:
  TLB hit rate: 5.5000 %
  Page fault rate: 24.4000 %

```

图 1: vmm 测试 (FRAME NUM=256)

- vmm 测试 (FRAME NUM=128) (图 2)

```
gqy@gqy-VirtualBox:~/os_proj8$ ./vmm addresses.txt
Statistics:
  TLB hit rate: 5.5000 %
  Page fault rate: 53.9000 %
```

图 2: vmm 测试 (FRAME NUM=128)

测试指令如下

```
1 make
2 ./vmm addresses.txt
```

首先用 Makefile 文件编译,生成可执行文件 vmm,然后对提供的 addresses.txt 执行,输出结果在文件 answer.txt 中,与提供的源文件 correct.txt 可以比较读取数据正确性,执行完成后打印了 TLB 命中率以及缺页率。图 1 为设置物理内存帧数为 256 的测试结果,图 2 为设置物理内存帧数为 128 的测试结果,两次测试的输出文件 answer.txt 与 correct.txt 对比都正确。

## 2 Conclusion

### 2.1 问题与解决方案

本次 project8 设计的虚拟内存管理器难度稍大,需要对物理内存,页表,TLB 在根据地址读取信息的运行流程有比较透彻的理解,并且由于不同模块之间的相互作用,要设计高效且清晰的程序架构有较大的思考量。最开始我对于整个 project 的设计并没有什么头绪,几次翻阅书本并在头脑中不断复现整个虚拟内存工作流程后慢慢将整个工作分解为一个个函数,然后将其一个个实现出来,其中 LRU 置换策略的实现是一个难点,实现的过程中也出现了一些错误,仔细思考并调试后最后成功实现了。

### 2.2 实验心得

本次 project8 是对虚拟内存部分知识一次比较透彻的应用,在设计虚拟内存管理器的同时也帮助我对所学知识有了更深入的理解,而且本次 project 的架构根据虚拟内存原理而设计,函数相较于之前多且复杂,需要保持清醒的头脑,在程序出错的时候耐心寻找错误来源,也使编程能力有较大的提升。另外本次 project 也是操作系统课程设计的最后一个 project,虽然在完成课程设计的时候会不断遇到一些问题,但最终都顺利实现了所有的项目内容,将理论知识和实践很好地结合在一起,总体来说让我受益匪浅。最后再一次感谢吴老师与助教们的悉心指导!