

Proj8 Designing a Virtual Memory Manager

by 郑航 520021911347

Proj8 Designing a Virtual Memory Manager

- 1 Abstract
- 2 Requirements
- 3 Implementation
 - 3.1 数据结构和全局变量的设计
 - 3.1.1 基本数据结构
 - 3.1.2 双向链表
 - 3.1.3 单向链表
 - 3.1.4 全局变量
 - 3.2 main函数的设计与实现
 - 3.2.1 初始化部分
 - 3.2.2 循环address读入处理部分
 - 3.2.3 数据统计部分
 - 3.2.4 退出部分
 - 3.3 init() 和clean() 的实现
 - 3.3.1 init()
 - 3.3.2 clean()
 - 3.4 ger_frame_id() 的实现
 - 3.4.1 search_in_TLB()
 - 3.4.2 search_in_page_table()
- 4 Test Result
- 5 Difficulty&Summary
 - 5.1 Difficulty
 - 5.2 Summary
- 6 Appendix
 - 6.1 vm_manager.c
 - 6.2 data_structure.h
 - 6.3 data_structure.c

1 Abstract

实现了一个虚拟内存管理的模拟程序，程序需要管理一个页表、帧表以及TLB，并模拟根据虚拟地址从对应的物理地址中读取数据的过程

功能：模拟虚拟内存，需要管理一个TLB，页表以及帧表，能够完成虚拟地址到物理地址的转换，从内存中物理地址的对应位置处读取数据，能够采用 demand paging 处理缺页错误，其中TLB和帧表采用LRU替换策略

2 Requirements

This project consists of writing a program that translates logical to physical addresses for a virtual address space of size . Our program will

- read from a file containing logical addresses
- using a TLB and a page table, translate each logical address to its corresponding physical address
- output the value of the byte stored at the translated physical address

主要要实现：

- Address Translation
 - 首先访问TLB，TLB-hit的情况则直接返回对应frame index
 - TLB-miss的情况则再去页表中，寻找对应的frame index
- TLB Management

TLB 的entry数较少，需要经常进行替换，需要设计一个TLB的替换算法
- Handling Page Fault

采用demand paging的策略，只有需要时才会将数据从磁盘（此处即BACKING_STORE.bin 文件）中读取数据到内存中，因此页表中可能找不到我们所需的frame index，此时需要先将数据从BACKING_STORE.bin中的对应位置处读入到一个空闲帧中，并更新页表valid bit和TLB
- Page Replacement

当帧表大小小于页表大小时，可能出现缺页错误但无空闲帧可用的情况，此时需要选择一个victim frame进行替换，需要设计一个帧表的替换算法
- Test

本次project需要从address.txt中读取虚拟地址值，转换为对应的物理地址，并读取对应物理地址处的数据，将以上两个地址和读取数据按行存入一个out.txt文件中，并与correct.txt进行比对，若一致则说明程序运行正确
- Statistics

需要统计TLB hit rate TLB命中率和page fault rate 缺页错误率

3 Implementation

本次project主要分以下六部分进行实现：

- 数据结构和全局变量的设计
- main函数的设计与实现
- init() 和clean() 的实现
- ger_frame_id() 的实现

其中，数据结构设计部分是本project的基础，在里边我们定义了一系列的结构体和相应的操作函数；**main()** 是整个程序的框架，主要是不断读入地址并以该地址为基础进行操作；**init()**是初始化函数，主要包括文件的打开和数据结构的实例化两部分，**clean()** 是清理函数，主要是文件关闭和动态内存的释放；**ger_frame_id()**是本次project的重点，综合了对TLB和页表等的访问，获取当前页码对应的帧码，其内部也实现了诸如缺页错误的处理等各种功能

本次project的总体实现思路详见 **3.2 main函数的设计与实现**部分

3.1 数据结构和全局变量的设计

以下数据结构我们都在`data_structure.h`中进行声明，并在`data_structure.c`中进行实现

3.1.1 基本数据结构

在本project中，我们将TLB，page_table和memory中的每一个item都抽象为一个数据结构，分别如下：

```
1  typedef struct TLB_ITEM
2  {
3      int valid; // valid bit
4      int frame_id;
5      int page_id;
6  } tlb_item;
7
8  typedef struct PAGE
9  {
10     int valid; // valid bit
11     int frame_id;
12 } page;
13
14 typedef struct FRAME
15 {
16     char data[FRAME_SIZE];
17 } frame;
```

各个数据成员的意义较为明确，在此不加赘述

3.1.2 双向链表

本project中，TLB和frame的替换我们都采用LRU的替换策略，有两种实现方式：①记录最近使用时间，②采用堆栈；在此我们采用第②种，堆栈的方式进行实现，效率更高，无需在每个item中增加最近使用时间的相关信息，因此需要设计一个双向链表，其结点定义如下：

```
1  typedef struct STACK_NODE
2  {
3      int id;
4      struct STACK_NODE *prior;
5      struct STACK_NODE *next;
6  } stack_node;
```

其操作函数声明如下：

```
1  //move the node with id to the top of the stack (means it is the latest id
   been visited)
2  void move_to_top(stack_node *head, stack_node *tail, int id);
3
4  //get the bottom node's id (means it's the last recently visited id)
5  int get_buttom_id(stack_node *tail);
6
7  //release the memory
8  void clean_stack(stack_node *head);
```

实现LRU策略需要以下两个操作：①将一个带有某个id的元素移到栈顶，表示其最近被访问过；②将栈底的元素的id取出，代表其是最久未被访问的id，需要被替换；除此之外还需要一个clean_stack函数进行动态内存释放

双向链表的实现我们采用的是首尾两个虚拟结点的方式，即head和tail结点不储存数据

三个函数的具体实现如下：

```
1 void move_to_top(stack_node *head, stack_node *tail, int id)
2 {
3     stack_node *tmp = head->next;
4     while (tmp != tail)
5     {
6         //the node with id already exists
7         if (tmp->id == id)
8         {
9             tmp->prior->next = tmp->next;
10            tmp->next->prior = tmp->prior;
11            head->next->prior = tmp;
12            tmp->next = head->next;
13            head->next = tmp;
14            tmp->prior = head;
15            return;
16        }
17        tmp = tmp->next;
18    }
19
20    //the node with id doesn't exists, create a new node
21    tmp = (stack_node *)malloc(sizeof(stack_node *));
22    tmp->id = id;
23    head->next->prior = tmp;
24    tmp->next = head->next;
25    head->next = tmp;
26    tmp->prior = head;
27    return;
28 }
29
30 int get_bottom_id(stack_node *tail)
31 {
32     return tail->prior->id;
33 }
34
35 void clean_stack(stack_node *head)
36 {
37     while (head)
38     {
39         stack_node *tmp = head;
40         head = head->next;
41         free(tmp);
42     }
43 }
```

3.1.3 单向链表

本project中，由于不存在进程替换等，TLB和memory满了之后就不会再有空的项可供使用了，只有在TLB和memory满之前需要记录空的项，但实际情况中这样的需求会更加明显，因此有必要设计一个记录未使用的/无效的TLB_item和frame的列表，每次需要时先从该表查看是否有可用的项，在这里我们使用一个单项链表来实现，其结点定义如下：

```
1 typedef struct LIST_NODE
2 {
3     int id;
4     struct LIST_NODE *next;
5 } list_node;
```

其操作函数声明如下：

```
1 //insert a new free node with id
2 void insert_node(list_node *head, int id);
3
4 //remove a free node (means it is selected to be used)
5 int remove_node(list_node *head);
6
7 //release the memory
8 void clean_list(list_node *head);
```

该列表只需要实现最简单的insert，remove和clean即可，三个函数具体实现如下：

```
1 void insert_node(list_node *head, int id)
2 {
3     list_node *tmp = (list_node *)malloc(sizeof(list_node *));
4     tmp->id = id;
5     tmp->next = head->next;
6     head->next = tmp;
7 }
8
9 int remove_node(list_node *head)
10 {
11     if (head->next == NULL)
12         return -1;
13
14     int id = head->next->id;
15     list_node *tmp = head->next;
16     head->next = tmp->next;
17     free(tmp);
18
19     return id;
20 }
21
22 void clean_list(list_node *head)
23 {
24     while (head)
25     {
26         list_node *tmp = head;
27         head = head->next;
28         free(tmp);
29     }
30 }
```

3.1.4 全局变量

程序的全局变量设计如下：

```
1  tlb_item TLB[TLB_ENTRY_NUM];           // TLB
2  page page_table[PAGE_TABLE_ENTRY_NUM]; // page_table
3  frame memory[FRAME_NUM];                // memory
4
5  FILE *fp_addr; // addresses.txt
6  FILE *fp_out;  // out.txt
7  FILE *fp_store; // BACKING_STORE.bin
8
9  stack_node *TLB_stack_head, *TLB_stack_tail; // TLB stack
10 stack_node *frame_stack_head, *frame_stack_tail; // frame stack
11
12 list_node *free_frame_list; // free frame list
13 list_node *free_TLB_list;  // free TLB list
14
15 int count = 0; // total addresses number
16 int tlb_hit = 0; // TLB hit times
17 int page_fault = 0; // page fault times
```

3.2 main函数的设计与实现

main函数主要分为四个部分：

- 初始化部分，进行valid bit 置位，文件打开，栈和链表的初始化
- 循环address读入处理部分
- 数据统计部分
- 退出部分

3.2.1 初始化部分

调用init() 函数即可，init() 具体实现见 3.3 init() 和clean() 的实现

3.2.2 循环address读入处理部分

while语句的判断条件为

```
1 ~fscanf(fp_addr, "%d", &addr)
```

每个循环中读入一个地址存于addr中，读取成功返回读入字符数，失败返回0，读取到文件尾返回EOF（值为-1），因此按位取反即可表示读取成功的情况

根据address的编码规则，依靠位运算可以取出对应的页码和offset，并调用ger_frame_id() 函数获取frame_id，计算物理地址，并取出对应地址处的数据存入out.txt中

该部分代码如下：

```

1 while (~fscanf(fp_addr, "%d", &addr))
2 {
3     count++;
4     addr &= 0x0000ffff;
5     offset = addr & 0x000000ff;
6     page_id = (addr >> 8) & 0x000000ff;
7
8     frame_id = ger_frame_id(page_id);
9     int data = memory[frame_id].data[offset];
10    int phy_addr = frame_id * FRAME_SIZE + offset;
11
12    fprintf(fp_out, "Virtual address: %d Physical address: %d Value: %d\n",
13           addr, phy_addr, data);
14 }

```

3.2.3 数据统计部分

主要就是一些简单计算和打印：

```

1 printf("[Statistics]\n");
2 printf("    Frame number: %d\n", FRAME_NUM);
3 printf("    TLB hit rate: %.4f %%\n", 100.0 * tlb_hit / count);
4 printf("    Page fault rate: %.4f %%\n", 100.0 * page_fault / count);

```

3.2.4 退出部分

调用clean() 函数即可，clean() 具体实现见 3.3 init() 和clean() 的实现

main() 具体代码如下：

```

1 int main(int argc, char *argv[])
2 {
3     // initialization
4     init(argc, argv);
5
6     int addr, page_id, frame_id, offset;
7
8     while (~fscanf(fp_addr, "%d", &addr))
9     {
10        count++;
11        addr &= 0x0000ffff;
12        offset = addr & 0x000000ff;
13        page_id = (addr >> 8) & 0x000000ff;
14
15        frame_id = ger_frame_id(page_id);
16        int data = memory[frame_id].data[offset];
17        int phy_addr = frame_id * FRAME_SIZE + offset;
18
19        fprintf(fp_out, "Virtual address: %d Physical address: %d Value:
20        %d\n", addr, phy_addr, data);
21    }
22
23    printf("[Statistics]\n");
24    printf("    Frame number: %d\n", FRAME_NUM);
25    printf("    TLB hit rate: %.4f %%\n", 100.0 * tlb_hit / count);

```

```

25     printf("    Page fault rate: %.4f %%\n", 100.0 * page_fault / count);
26
27     clean();
28     return 0;
29 }

```

3.3 init() 和clean() 的实现

3.3.1 init()

初始化主要分为四个部分：

- 参数正确性检查，我们检查参数数目是否为2
- 对TLB和page_table中各项的valid bit 置位为0
- 文件打开，注意判断打开是否成功
- 初始化栈和空闲链表

栈的部分，注意head和tail都是虚拟结点，不存放数据，因此此处需要分别初始化两个结点，并将head的next指向tail，tail的prior指向head，其他置为NULL

空闲链表部分，需要逆序将id都先插入，取出时才会是从id==0开始取出

该部分代码如下：

```

1 // initialization
2 init(argc, argv);

```

init()代码如下：

```

1 void init(int argc, char *argv[])
2 {
3     if (argc <= 1 || argc >= 3)
4     {
5         printf("Error: invalid arguments!\n");
6         exit(1);
7     }
8
9     for (int i = 0; i < TLB_ENTRY_NUM; i++)
10    {
11        TLB[i].valid = 0;
12    }
13
14    for (int i = 0; i < PAGE_TABLE_ENTRY_NUM; i++)
15    {
16        page_table[i].valid = 0;
17    }
18
19    fp_addr = fopen(argv[1], "r");
20    if (fp_addr == NULL)
21    {
22        printf("Error: open file %s failed!\n", argv[1]);
23        exit(1);
24    }
25    fp_out = fopen("out.txt", "w");

```



```

26     if (fp_out == NULL)
27     {
28         printf("Error: create file out.txt failed!\n");
29         exit(1);
30     }
31     fp_store = fopen("BACKING_STORE.bin", "rb");
32     if (fp_store == NULL)
33     {
34         printf("Error: open file BACKING_STORE.bin failed!\n");
35         exit(1);
36     }
37
38     TLB_stack_head = (stack_node *)malloc(sizeof(stack_node *));
39     TLB_stack_tail = (stack_node *)malloc(sizeof(stack_node *));
40     TLB_stack_head->prior = NULL;
41     TLB_stack_head->next = TLB_stack_tail;
42     TLB_stack_tail->prior = TLB_stack_head;
43     TLB_stack_tail->next = NULL;
44
45     frame_stack_head = (stack_node *)malloc(sizeof(stack_node *));
46     frame_stack_tail = (stack_node *)malloc(sizeof(stack_node *));
47     frame_stack_head->prior = NULL;
48     frame_stack_head->next = frame_stack_tail;
49     frame_stack_tail->prior = frame_stack_head;
50     frame_stack_tail->next = NULL;
51
52     free_frame_list = (list_node *)malloc(sizeof(list_node *));
53     free_frame_list->next = NULL;
54     for (int i = FRAME_NUM - 1; i >= 0; i--)
55         insert_node(free_frame_list, i);
56
57     free_TLB_list = (list_node *)malloc(sizeof(list_node *));
58     free_TLB_list->next = NULL;
59     for (int i = TLB_ENTRY_NUM - 1; i >= 0; i--)
60         insert_node(free_TLB_list, i);
61 }

```

3.3.2 clean()

退出部分，主要分两部分：①文件关闭；②堆栈和链表的资源回收。两部分都是调用对应的函数即可

clean() 具体代码如下：

```

1  void clean()
2  {
3      fclose(fp_addr);
4      fclose(fp_out);
5      fclose(fp_store);
6
7      clean_stack(TLB_stack_head);
8      clean_stack(frame_stack_head);
9      clean_list(free_TLB_list);
10     clean_list(free_frame_list);
11 }

```

3.4 ger_frame_id() 的实现

依据虚拟内存和TLB的工作方式，我们首先需要在TLB中查找，TLB miss后再从page_table中查找，我们将这两个过程封装为如下两个函数：

```
1 int search_in_TLB(int page_id);
2 int search_in_page_table(int page_id);
```

其中，若TLB miss，则search_in_TLB() 返回-1

有了这两个函数后，ger_frame_id() 的实现就变得简单了，代码如下：

```
1 int ger_frame_id(int page_id)
2 {
3     int frame_id;
4     frame_id = search_in_TLB(page_id);
5     if (frame_id != -1)
6     {
7         return frame_id;
8     }
9
10    frame_id = search_in_page_table(page_id);
11    return frame_id;
12 }
```

下面具体说明search_in_TLB() 和search_in_page_table() 的实现

3.4.1 search_in_TLB()

我们实现的TLB采用全相联映射的方式，每个记录都可以存放到任何一个空的entry中，因此search_in_TLB()的主体就是一个for循环，判断当前entry中记录的page_id 是否是所需page_id，若是的话则更新TLB和frame的栈，并返回该id；否则返回-1

```
1 int search_in_TLB(int page_id)
2 {
3     for (int i = 0; i < TLB_ENTRY_NUM; i++)
4     {
5         if (!TLB[i].valid)
6             continue;
7
8         if (TLB[i].page_id == page_id)
9         {
10             tlb_hit++;
11             move_to_top(TLB_stack_head, TLB_stack_tail, i);
12             move_to_top(frame_stack_head, frame_stack_tail,
TLB[i].frame_id);
13             return TLB[i].frame_id;
14         }
15     }
16     return -1;
17 }
```

3.4.2 search_in_page_table()

我们的目的是获取对应帧的frame_id，情况分类如下：

- 若page_table中有当前项，则frame_id 立即可得
- 若page_table中没有当前项，则发生了page fault：
 - 若有空闲帧，则直接取该空闲帧即可
 - 若无空闲帧，则需要从frame_stack栈底取出一个帧作为victim frame，并将所有涉及到该victim frame的page的valid bit置为0

在如上取出的可用帧中，存入从BACKING_STORE.bin 读入的数据，并更新页表

之后，我们需要更新frame_stack的堆栈，将当前帧放至栈顶，表示最近才被访问过

由于此时是TLB miss的情况，才需要在页表中搜索，因此接下来还需要对TLB进行更新，其更新过程类似page_table的更新，如下：

- 若有空闲TLB entry，则直接取该TLB entry
- 若无空闲TLB entry，则需要从TLB_stack栈底取出一个帧作为victim TLB entry

将数据写入上述取出的可用TLB entry，并更新TLB_stack的堆栈，将当前TLB entry放至栈顶，表示最近才被访问过

search_in_page_table() 具体代码如下：

```
1  int search_in_page_table(int page_id)
2  {
3      int frame_id;
4      if (page_table[page_id].valid == 1)
5      {
6          frame_id = page_table[page_id].frame_id;
7      }
8      // page fault
9      else
10     {
11         page_fault++;
12         frame_id = search_unused_frame();
13
14         // page replacement
15         if (frame_id == -1)
16         {
17             int frame_to_be_replace = get_bottom_id(frame_stack_tail);
18             for (int i = 0; i < PAGE_TABLE_ENTRY_NUM; i++)
19             {
20                 if (page_table[i].frame_id == frame_to_be_replace)
21                 {
22                     page_table[i].valid = 0;
23                 }
24             }
25             frame_id = frame_to_be_replace;
26         }
27
28         fseek(fp_store, page_id * PAGE_SIZE, SEEK_SET);
29         fread(memory[frame_id].data, sizeof(char), FRAME_SIZE, fp_store);
30
31         page_table[page_id].frame_id = frame_id;
```

```

32     page_table[page_id].valid = 1;
33 }
34
35 move_to_top(frame_stack_head, frame_stack_tail, frame_id);
36
37 // update TLB
38 int TLB_id = search_unused_TLB();
39 if (TLB_id == -1)
40 {
41     TLB_id = get_bottom_id(TLB_stack_tail);
42 }
43 else
44 {
45     TLB[TLB_id].valid = 1;
46 }
47
48 TLB[TLB_id].page_id = page_id;
49 TLB[TLB_id].frame_id = frame_id;
50 move_to_top(TLB_stack_head, TLB_stack_tail, TLB_id);
51
52 return frame_id;
53 }

```

其中，`search_unused_TLB()` 和 `search_unused_frame()` 是两个封装的工具函数，分别从空闲TLB和空闲frame链表中取出空闲项

`search_unused_TLB()` 和 `search_unused_frame()` 具体代码如下：

```

1 // return index of an unused frame, return -1 if no such frame
2 int search_unused_frame()
3 {
4     int id = remove_node(free_frame_list);
5     return id;
6 }
7
8 // return index of an unused TLB entry, return -1 if no such TLB entry
9 int search_unused_TLB()
10 {
11     int id = remove_node(free_TLB_list);
12     return id;
13 }

```

`vm_manager.c` , `data_structure.c` 和 `data_structure.h` 的完整代码见附录

4 Test Result

首先编写如下的Makefile对这几个项目文件进行编译

```

1 CC=gcc
2 CFLAGS=-Wall
3
4 vm_manager: vm_manager.o data_structure.o
5     $(CC) $(CFLAGS) -o vm_manager vm_manager.o data_structure.o

```

```

6
7 vm_manager.o: vm_manager.c
8     $(CC) $(CFLAGS) -c vm_manager.c
9
10 data_structure.o: data_structure.c data_structure.h
11     $(CC) $(CFLAG) -c data_structure.c
12
13 clean:
14     rm -rf *.o
15     rm -rf vm_manager

```

运行结果如下:

```

zh@ubuntu:~/os-project/pro8$ make
gcc -Wall -c vm_manager.c
gcc -c data_structure.c
gcc -Wall -o vm_manager vm_manager.o data_structure.o
zh@ubuntu:~/os-project/pro8$ ./vm_manager addresses.txt
[Statistics]
    Frame number: 256
    TLB hit rate: 5.5000 %
    Page fault rate: 24.4000 %

```

接下来需要将out.txt 和正确结果的correct.txt 进行对比, 我们通过diff命令进行比较:

```

1 diff -s "correct.txt" "out.txt"

```

结果如下:

```

zh@ubuntu:~/os-project/pro8$ ./vm_manager addresses.txt
[Statistics]
    Frame number: 256
    TLB hit rate: 5.5000 %
    Page fault rate: 24.4000 %
zh@ubuntu:~/os-project/pro8$ diff -s "correct.txt" "out.txt"
Files correct.txt and out.txt are identical

```

可见, out.txt 和 correct.txt 内容完全一致, 本程序顺利通过了测试

下面我们修改一下FRAME_NUM

FRAME_NUM==128:

```

zh@ubuntu:~/os-project/pro8$ make
gcc -Wall -c vm_manager.c
gcc -c data_structure.c
gcc -Wall -o vm_manager vm_manager.o data_structure.o
zh@ubuntu:~/os-project/pro8$ ./vm_manager addresses.txt
[Statistics]
    Frame number: 128
    TLB hit rate: 5.5000 %
    Page fault rate: 53.9000 %

```

FRAME_NUM==64:

```

zh@ubuntu:~/os-project/pro8$ make
gcc -Wall -c vm_manager.c
gcc -c data_structure.c
gcc -Wall -o vm_manager vm_manager.o data_structure.o
zh@ubuntu:~/os-project/pro8$ ./vm_manager addresses.txt
[Statistics]
    Frame number: 64
    TLB hit rate: 5.5000 %
    Page fault rate: 75.4000 %

```

FRAME_NUM==1:

```

zh@ubuntu:~/os-project/pro8$ make
gcc -Wall -c vm_manager.c
gcc -c data_structure.c
gcc -Wall -o vm_manager vm_manager.o data_structure.o
zh@ubuntu:~/os-project/pro8$ ./vm_manager addresses.txt
[Statistics]
    Frame number: 1
    TLB hit rate: 5.5000 %
    Page fault rate: 94.5000 %

```

可以看到，随着FRAME_NUM的减小，page fault rate逐渐增大，TLB hit rate 则不变化（TLB_ENTRY_NUM不变化），且由FRAME_NUM==1时，page fault rate不为100%，可见应该存在相邻的对同一帧的访问

增大TLB_ENTRY_NUM:

TLB_ENTRY_NUM==32:

```

zh@ubuntu:~/os-project/pro8$ make
gcc -Wall -c vm_manager.c
gcc -c data_structure.c
gcc -Wall -o vm_manager vm_manager.o data_structure.o
zh@ubuntu:~/os-project/pro8$ ./vm_manager addresses.txt
[Statistics]
    Frame number: 256
    TLB hit rate: 12.1000 %
    Page fault rate: 24.4000 %

```

可以看到，随着TLB_ENTRY_NUM的增加，TLB hit rate 显著增大

5 Difficulty&Summary

5.1 Difficulty

本次project难度在所有project中属于较大的，尤其是需要操作的数据较多，时刻需要更新一些数据状态等。通过封装各类数据结构，并实现对应的操作函数，有效的提高了代码编写的效率，提高了代码的可读性。

第一次在vscode上进行多文件程序的调试，但是在之前的配环境基础上，进行起来还是比较顺畅的

由于有多个源程序文件，需要学习编写多文件编译的Makefile文件，且在测试时发现若仅改动data_structure.h头文件中的FRAME_NUM等数据，重新编译时并不会对vm_manager.c进行编译，即修改结果不会更新到vm_manager中，一开始因此导致测试结果有误，后来发现需要对vm_manager.c也再次进行编译才能得到正确结果

5.2 Summary

本次通过编写模拟虚拟内存管理的程序，对课内内容进行实践，使得我对该内存分配算法有了更为深入的理解。同时，本次project数据量较大，数据状态需要时刻保持更新，需要一些耐心和细致；此外，函数的数目较多，需要好好组织代码结构并且合理添加注释，才可以使得代码可读性更强一些

本次project思路不算太难，主要是更熟悉了一些双向链表等的操作技巧，锻炼了编程能力。总之，本次project难度适中，需要一定的思考和调试时间，也令我收获良多

6 Appendix

6.1 vm_manager.c

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4  #include <unistd.h>
5  #include "data_structure.h"
6
7  tlb_item TLB[TLB_ENTRY_NUM];          // TLB
8  page page_table[PAGE_TABLE_ENTRY_NUM]; // page_table
9  frame memory[FRAME_NUM];              // memory
10
11 FILE *fp_addr; // addresses.txt
12 FILE *fp_out;  // out.txt
13 FILE *fp_store; // BACKING_STORE.bin
14
15 stack_node *TLB_stack_head, *TLB_stack_tail; // TLB stack
16 stack_node *frame_stack_head, *frame_stack_tail; // frame stack
17
18 list_node *free_frame_list; // free frame list
19 list_node *free_TLB_list;   // free TLB list
20
21 int count = 0; // total addresses number
22 int tlb_hit = 0; // TLB hit times
23 int page_fault = 0; // page fault times
24
25 void init(int argc, char *argv[]);
26 void clean();
27 int ger_frame_id(int page_id);
28 int search_in_TLB(int page_id);
29 int search_in_page_table(int page_id);
30 int search_unused_frame();
31 int search_unused_TLB();
32
33 int main(int argc, char *argv[])
34 {
35     // initialization
36     init(argc, argv);
37 }
```

```

38     int addr, page_id, frame_id, offset;
39
40     while (~fscanf(fp_addr, "%d", &addr))
41     {
42         count++;
43         addr &= 0x0000ffff;
44         offset = addr & 0x000000ff;
45         page_id = (addr >> 8) & 0x000000ff;
46
47         frame_id = ger_frame_id(page_id);
48         int data = memory[frame_id].data[offset];
49         int phy_addr = frame_id * FRAME_SIZE + offset;
50
51         fprintf(fp_out, "Virtual address: %d Physical address: %d value:
%d\n", addr, phy_addr, data);
52     }
53
54     printf("[Statistics]\n");
55     printf("    Frame number: %d\n", FRAME_NUM);
56     printf("    TLB hit rate: %.4f %%\n", 100.0 * tlb_hit / count);
57     printf("    Page fault rate: %.4f %%\n", 100.0 * page_fault / count);
58
59     clean();
60     return 0;
61 }
62
63 void init(int argc, char *argv[])
64 {
65     if (argc <= 1 || argc >= 3)
66     {
67         printf("Error: invalid arguments!\n");
68         exit(1);
69     }
70
71     for (int i = 0; i < TLB_ENTRY_NUM; i++)
72     {
73         TLB[i].valid = 0;
74     }
75
76     for (int i = 0; i < PAGE_TABLE_ENTRY_NUM; i++)
77     {
78         page_table[i].valid = 0;
79     }
80
81     fp_addr = fopen(argv[1], "r");
82     if (fp_addr == NULL)
83     {
84         printf("Error: open file %s failed!\n", argv[1]);
85         exit(1);
86     }
87     fp_out = fopen("out.txt", "w");
88     if (fp_out == NULL)
89     {
90         printf("Error: create file out.txt failed!\n");
91         exit(1);
92     }
93     fp_store = fopen("BACKING_STORE.bin", "rb");
94     if (fp_store == NULL)

```



```

95     {
96         printf("Error: open file BACKING_STORE.bin failed!\n");
97         exit(1);
98     }
99
100     TLB_stack_head = (stack_node *)malloc(sizeof(stack_node *));
101     TLB_stack_tail = (stack_node *)malloc(sizeof(stack_node *));
102     TLB_stack_head->prior = NULL;
103     TLB_stack_head->next = TLB_stack_tail;
104     TLB_stack_tail->prior = TLB_stack_head;
105     TLB_stack_tail->next = NULL;
106
107     frame_stack_head = (stack_node *)malloc(sizeof(stack_node *));
108     frame_stack_tail = (stack_node *)malloc(sizeof(stack_node *));
109     frame_stack_head->prior = NULL;
110     frame_stack_head->next = frame_stack_tail;
111     frame_stack_tail->prior = frame_stack_head;
112     frame_stack_tail->next = NULL;
113
114     free_frame_list = (list_node *)malloc(sizeof(list_node *));
115     free_frame_list->next = NULL;
116     for (int i = FRAME_NUM - 1; i >= 0; i--)
117         insert_node(free_frame_list, i);
118
119     free_TLB_list = (list_node *)malloc(sizeof(list_node *));
120     free_TLB_list->next = NULL;
121     for (int i = TLB_ENTRY_NUM - 1; i >= 0; i--)
122         insert_node(free_TLB_list, i);
123 }
124
125 void clean()
126 {
127     fclose(fp_addr);
128     fclose(fp_out);
129     fclose(fp_store);
130
131     clean_stack(TLB_stack_head);
132     clean_stack(frame_stack_head);
133     clean_list(free_TLB_list);
134     clean_list(free_frame_list);
135 }
136
137 int ger_frame_id(int page_id)
138 {
139     int frame_id;
140     frame_id = search_in_TLB(page_id);
141     if (frame_id != -1)
142     {
143         return frame_id;
144     }
145
146     frame_id = search_in_page_table(page_id);
147     return frame_id;
148 }
149
150 int search_in_TLB(int page_id)
151 {
152     for (int i = 0; i < TLB_ENTRY_NUM; i++)

```

```

153     {
154         if (!TLB[i].valid)
155             continue;
156
157         if (TLB[i].page_id == page_id)
158         {
159             tlb_hit++;
160             move_to_top(TLB_stack_head, TLB_stack_tail, i);
161             move_to_top(frame_stack_head, frame_stack_tail,
TLB[i].frame_id);
162             return TLB[i].frame_id;
163         }
164     }
165     return -1;
166 }
167
168 int search_in_page_table(int page_id)
169 {
170     int frame_id;
171     if (page_table[page_id].valid == 1)
172     {
173         frame_id = page_table[page_id].frame_id;
174     }
175     // page fault
176     else
177     {
178         page_fault++;
179         frame_id = search_unused_frame();
180
181         // page replacement
182         if (frame_id == -1)
183         {
184             int frame_to_be_replace = get_bottom_id(frame_stack_tail);
185             for (int i = 0; i < PAGE_TABLE_ENTRY_NUM; i++)
186             {
187                 if (page_table[i].frame_id == frame_to_be_replace)
188                 {
189                     page_table[i].valid = 0;
190                 }
191             }
192             frame_id = frame_to_be_replace;
193         }
194
195         fseek(fp_store, page_id * PAGE_SIZE, SEEK_SET);
196         fread(memory[frame_id].data, sizeof(char), FRAME_SIZE, fp_store);
197
198         page_table[page_id].frame_id = frame_id;
199         page_table[page_id].valid = 1;
200     }
201
202     move_to_top(frame_stack_head, frame_stack_tail, frame_id);
203
204     // update TLB
205     int TLB_id = search_unused_TLB();
206     if (TLB_id == -1)
207     {
208         TLB_id = get_bottom_id(TLB_stack_tail);
209     }

```

```

210     else
211     {
212         TLB[TLB_id].valid = 1;
213     }
214
215     TLB[TLB_id].page_id = page_id;
216     TLB[TLB_id].frame_id = frame_id;
217     move_to_top(TLB_stack_head, TLB_stack_tail, TLB_id);
218
219     return frame_id;
220 }
221
222 // return index of an unused frame, return -1 if no such frame
223 int search_unused_frame()
224 {
225     int id = remove_node(free_frame_list);
226     return id;
227 }
228
229 // return index of an unused TLB entry, return -1 if no such TLB entry
230 int search_unused_TLB()
231 {
232     int id = remove_node(free_TLB_list);
233     return id;
234 }

```

6.2 data_structure.h

```

1  #define PAGE_TABLE_ENTRY_NUM 256
2  #define PAGE_SIZE 256
3  #define TLB_ENTRY_NUM 32
4  #define FRAME_SIZE 256
5  #define FRAME_NUM 256
6
7  typedef struct TLB_ITEM
8  {
9      int valid; // valid bit
10     int frame_id;
11     int page_id;
12 } tlb_item;
13
14 typedef struct PAGE
15 {
16     int valid; // valid bit
17     int frame_id;
18 } page;
19
20 typedef struct FRAME
21 {
22     char data[FRAME_SIZE];
23 } frame;
24
25 /***** stack begin
26 *****/
27 typedef struct STACK_NODE
28 {

```

```

29     int id;
30     struct STACK_NODE *prior;
31     struct STACK_NODE *next;
32 } stack_node;
33
34 //move the node with id to the top of the stack (means it is the latest id
    been visited)
35 void move_to_top(stack_node *head, stack_node *tail, int id);
36
37 //get the bottom node's id (means it's the last recently visited id)
38 int get_bottom_id(stack_node *tail);
39
40 //release the memory
41 void clean_stack(stack_node *head);
42
43 /***** stack end *****/
44
45 /***** list begin *****/
46
47 typedef struct LIST_NODE
48 {
49     int id;
50     struct LIST_NODE *next;
51 } list_node;
52
53 //insert a new free node with id
54 void insert_node(list_node *head, int id);
55
56 //remove a free node (means it is selected to be used)
57 int remove_node(list_node *head);
58
59 //release the memory
60 void clean_list(list_node *head);
61
62 /***** list end *****/
63

```

6.3 data_structure.c

```

1  #include "data_structure.h"
2  #include <stdio.h>
3  #include <stdlib.h>
4
5  /***** stack begin *****/
6
7  void move_to_top(stack_node *head, stack_node *tail, int id)
8  {
9      stack_node *tmp = head->next;
10     while (tmp != tail)
11     {
12         // the node with id already exists
13         if (tmp->id == id)
14         {
15             tmp->prior->next = tmp->next;
16             tmp->next->prior = tmp->prior;
17             head->next->prior = tmp;

```

```

18         tmp->next = head->next;
19         head->next = tmp;
20         tmp->prior = head;
21         return;
22     }
23     tmp = tmp->next;
24 }
25
26 // the node with id doesn't exists, create a new node
27 tmp = (stack_node *)malloc(sizeof(stack_node *));
28 tmp->id = id;
29 head->next->prior = tmp;
30 tmp->next = head->next;
31 head->next = tmp;
32 tmp->prior = head;
33 return;
34 }
35
36 int get_bottom_id(stack_node *tail)
37 {
38     return tail->prior->id;
39 }
40
41 void clean_stack(stack_node *head)
42 {
43     while (head)
44     {
45         stack_node *tmp = head;
46         head = head->next;
47         free(tmp);
48     }
49 }
50
51 /***** stack end *****/
52
53 /***** list begin *****/
54
55 void insert_node(list_node *head, int id)
56 {
57     list_node *tmp = (list_node *)malloc(sizeof(list_node *));
58     tmp->id = id;
59     tmp->next = head->next;
60     head->next = tmp;
61 }
62
63 int remove_node(list_node *head)
64 {
65     if (head->next == NULL)
66         return -1;
67
68     int id = head->next->id;
69     list_node *tmp = head->next;
70     head->next = tmp->next;
71     free(tmp);
72
73     return id;
74 }
75

```

```
76 void clean_list(list_node *head)
77 {
78     while (head)
79     {
80         list_node *tmp = head;
81         head = head->next;
82         free(tmp);
83     }
84 }
85
86 /***** list end *****/
87
```