Proj8 Designing a Virtual Memory Manager

by 郑航 520021911347

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：

```
typedef struct TLB_ITEM
{
    int valid; // valid bit
    int frame_id;
    int page_id;
} tlb_item;

typedef struct PAGE
{
    int valid; // valid bit
    int frame_id;
} page;

typedef struct FRAME
{
    char data[FRAME_SIZE];
} 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：

```
typedef struct STACK_NODE
{
    int id;
    struct STACK_NODE *prior;
    struct STACK_NODE *next;
} stack_node;
```

其操作函数声明如下：

```
//move the node with id to the top of the stack (means it is the latest id been visited)
void move_to_top(stack_node *head, stack_node *tail, int id);

//get the bottom node's id (means it's the last recently visited id)
int get_buttom_id(stack_node *tail);

//release the memory
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：

```
void move_to_top(stack_node *head, stack_node *tail, int id)
{
    stack_node *tmp = head->next;
    while (tmp != tail)
    {
        //the node with id already exists
        if (tmp->id == id)
        {
            tmp->prior->next = tmp->next;
            tmp->next->prior = tmp->prior;
            head->next->prior = tmp;
            tmp->next = head->next;
            head->next = tmp;
            tmp->prior = head;
            return;
        }
        tmp = tmp->next;
    }

    //the node with id doesn't exists, create a new node
    tmp = (stack_node *)malloc(sizeof(stack_node *));
    tmp->id = id;
    head->next->prior = tmp;
    tmp->next = head->next;
    head->next = tmp;
    tmp->prior = head;
    return;
}

int get_buttom_id(stack_node *tail)
{
    return tail->prior->id;
}

void clean_stack(stack_node *head)
{
    while (head)
    {
        stack_node *tmp = head;
        head = head->next;
        free(tmp);
    }
}
```

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：

```
typedef struct LIST_NODE
{
    int id;
    struct LIST_NODE *next;
} list_node;
```

其操作函数声明如下：

```
//insert a new free node with id
void insert_node(list_node *head, int id);

//remove a free node (means it is selected to be used)
int remove_node(list_node *head);

//release the memory
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：

```
void insert_node(list_node *head, int id)
{
    list_node *tmp = (list_node *)malloc(sizeof(list_node *));
    tmp->id = id;
    tmp->next = head->next;
    head->next = tmp;
}

int remove_node(list_node *head)
{
    if (head->next == NULL)
        return -1;

    int id = head->next->id;
    list_node *tmp = head->next;
    head->next = tmp->next;
    free(tmp);

    return id;
}

void clean_list(list_node *head)
{
    while (head)
    {
        list_node *tmp = head;
        head = head->next;
        free(tmp);
    }
}
```

3.1.4 全局变量

程序的全局变量设计如下：

```
tlb_item TLB[TLB_ENTRY_NUM];           // TLB
page page_table[PAGE_TABLE_ENTRY_NUM]; // page_table
frame memory[FRAME_NUM];               // memory

FILE *fp_addr;  // addresses.txt
FILE *fp_out;   // out.txt
FILE *fp_store; // BACKING_STORE.bin

stack_node *TLB_stack_head, *TLB_stack_tail;     // TLB stack
stack_node *frame_stack_head, *frame_stack_tail; // frame stack

list_node *free_frame_list; // free frame list
list_node *free_TLB_list;   // free TLB list

int count = 0;      // total addresses number
int tlb_hit = 0;    // TLB hit times
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

```
~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：

```
while (~fscanf(fp_addr, "%d", &addr))
{
    count++;
    addr &= 0x0000ffff;
    offset = addr & 0x000000ff;
    page_id = (addr >> 8) & 0x000000ff;

    frame_id = ger_frame_id(page_id);
    int data = memory[frame_id].data[offset];
    int phy_addr = frame_id * FRAME_SIZE + offset;

    fprintf(fp_out, "Virtual address: %d Physical address: %d Value: %d\n", addr, phy_addr, data);
}
```

3.2.3 数据统计部分

主要就是一些简单计算和打印：

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

```
int main(int argc, char *argv[])
{
    // initialization
    init(argc, argv);

    int addr, page_id, frame_id, offset;

    while (~fscanf(fp_addr, "%d", &addr))
    {
        count++;
        addr &= 0x0000ffff;
        offset = addr & 0x000000ff;
        page_id = (addr >> 8) & 0x000000ff;

        frame_id = ger_frame_id(page_id);
        int data = memory[frame_id].data[offset];
        int phy_addr = frame_id * FRAME_SIZE + offset;

        fprintf(fp_out, "Virtual address: %d Physical address: %d Value: %d\n", addr, phy_addr, data);
    }

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

```
    printf("    Page fault rate: %.4f %%\n", 100.0 * page_fault / count);

    clean();
    return 0;
}
```

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：

```
// initialization
init(argc, argv);
```

init()代码如下：

```
void init(int argc, char *argv[])
{
    if (argc <= 1 || argc >= 3)
    {
        printf("Error: invalid arguments!\n");
        exit(1);
    }

    for (int i = 0; i < TLB_ENTRY_NUM; i++)
    {
        TLB[i].valid = 0;
    }

    for (int i = 0; i < PAGE_TABLE_ENTRY_NUM; i++)
    {
        page_table[i].valid = 0;
    }

    fp_addr = fopen(argv[1], "r");
    if (fp_addr == NULL)
    {
        printf("Error: open file %s failed!\n", argv[1]);
        exit(1);
    }
    fp_out = fopen("out.txt", "w");
```

```
    if (fp_out == NULL)
    {
        printf("Error: create file out.txt failed!\n");
        exit(1);
    }
    fp_store = fopen("BACKING_STORE.bin", "rb");
    if (fp_store == NULL)
    {
        printf("Error: open file BACKING_STORE.bin failed!\n");
        exit(1);
    }

    TLB_stack_head = (stack_node *)malloc(sizeof(stack_node *));
    TLB_stack_tail = (stack_node *)malloc(sizeof(stack_node *));
    TLB_stack_head->prior = NULL;
    TLB_stack_head->next = TLB_stack_tail;
    TLB_stack_tail->prior = TLB_stack_head;
    TLB_stack_tail->next = NULL;

    frame_stack_head = (stack_node *)malloc(sizeof(stack_node *));
    frame_stack_tail = (stack_node *)malloc(sizeof(stack_node *));
    frame_stack_head->prior = NULL;
    frame_stack_head->next = frame_stack_tail;
    frame_stack_tail->prior = frame_stack_head;
    frame_stack_tail->next = NULL;

    free_frame_list = (list_node *)malloc(sizeof(list_node *));
    free_frame_list->next = NULL;
    for (int i = FRAME_NUM - 1; i >= 0; i--)
        insert_node(free_frame_list, i);

    free_TLB_list = (list_node *)malloc(sizeof(list_node *));
    free_TLB_list->next = NULL;
    for (int i = TLB_ENTRY_NUM - 1; i >= 0; i--)
        insert_node(free_TLB_list, i);
}
```

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：

```
void clean()
{
    fclose(fp_addr);
    fclose(fp_out);
    fclose(fp_store);

    clean_stack(TLB_stack_head);
    clean_stack(frame_stack_head);
    clean_list(free_TLB_list);
    clean_list(free_frame_list);
}
```

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：

```
int search_in_TLB(int page_id);
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：

```
int ger_frame_id(int page_id)
{
    int frame_id;
    frame_id = search_in_TLB(page_id);
    if (frame_id != -1)
    {
        return frame_id;
    }

    frame_id = search_in_page_table(page_id);
    return frame_id;
}
```

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

```
int search_in_TLB(int page_id)
{
    for (int i = 0; i < TLB_ENTRY_NUM; i++)
    {
        if (!TLB[i].valid)
            continue;

        if (TLB[i].page_id == page_id)
        {
            tlb_hit++;
            move_to_top(TLB_stack_head, TLB_stack_tail, i);
            move_to_top(frame_stack_head, frame_stack_tail,
TLB[i].frame_id);
            return TLB[i].frame_id;
        }
    }
    return -1;
}
```

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：

```
int search_in_page_table(int page_id)
{
    int frame_id;
    if (page_table[page_id].valid == 1)
    {
        frame_id = page_table[page_id].frame_id;
    }
    // page fault
    else
    {
        page_fault++;
        frame_id = search_unused_frame();

        // page replacement
        if (frame_id == -1)
        {
            int frame_to_be_replace = get_buttom_id(frame_stack_tail);
            for (int i = 0; i < PAGE_TABLE_ENTRY_NUM; i++)
            {
                if (page_table[i].frame_id == frame_to_be_replace)
                {
                    page_table[i].valid = 0;
                }
            }
            frame_id = frame_to_be_replace;
        }

        fseek(fp_store, page_id * PAGE_SIZE, SEEK_SET);
        fread(memory[frame_id].data, sizeof(char), FRAME_SIZE, fp_store);

        page_table[page_id].frame_id = frame_id;
```

```
        page_table[page_id].valid = 1;
    }

    move_to_top(frame_stack_head, frame_stack_tail, frame_id);

    // update TLB
    int TLB_id = search_unused_TLB();
    if (TLB_id == -1)
    {
        TLB_id = get_buttom_id(TLB_stack_tail);
    }
    else
    {
        TLB[TLB_id].valid = 1;
    }

    TLB[TLB_id].page_id = page_id;
    TLB[TLB_id].frame_id = frame_id;
    move_to_top(TLB_stack_head, TLB_stack_tail, TLB_id);

    return frame_id;
}
```

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：

```
// return index of an unused frame, return -1 if no such frame
int search_unused_frame()
{
    int id = remove_node(free_frame_list);
    return id;
}

// return index of an unused TLB entry, return -1 if no such TLB entry
int search_unused_TLB()
{
    int id = remove_node(free_TLB_list);
    return id;
}
```

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

```
CC=gcc
CFLAGS=-Wall

vm_manager: vm_manager.o data_structure.o
    $(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

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include "data_structure.h"

tlb_item TLB[TLB_ENTRY_NUM];           // TLB
page page_table[PAGE_TABLE_ENTRY_NUM]; // page_table
frame memory[FRAME_NUM];               // memory

FILE *fp_addr;  // addresses.txt
FILE *fp_out;   // out.txt
FILE *fp_store; // BACKING_STORE.bin

stack_node *TLB_stack_head, *TLB_stack_tail;     // TLB stack
stack_node *frame_stack_head, *frame_stack_tail; // frame stack

list_node *free_frame_list; // free frame list
list_node *free_TLB_list;   // free TLB list

int count = 0;      // total addresses number
int tlb_hit = 0;    // TLB hit times
int page_fault = 0; // page fault times

void init(int argc, char *argv[]);
void clean();
int ger_frame_id(int page_id);
int search_in_TLB(int page_id);
int search_in_page_table(int page_id);
int search_unused_frame();
int search_unused_TLB();

int main(int argc, char *argv[])
{
    // initialization
    init(argc, argv);

```

```
    int addr, page_id, frame_id, offset;

    while (~fscanf(fp_addr, "%d", &addr))
    {
        count++;
        addr &= 0x0000ffff;
        offset = addr & 0x000000ff;
        page_id = (addr >> 8) & 0x000000ff;

        frame_id = ger_frame_id(page_id);
        int data = memory[frame_id].data[offset];
        int phy_addr = frame_id * FRAME_SIZE + offset;

        fprintf(fp_out, "Virtual address: %d Physical address: %d Value:
%d\n", addr, phy_addr, data);
    }

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

    clean();
    return 0;
}

void init(int argc, char *argv[])
{
    if (argc <= 1 || argc >= 3)
    {
        printf("Error: invalid arguments!\n");
        exit(1);
    }

    for (int i = 0; i < TLB_ENTRY_NUM; i++)
    {
        TLB[i].valid = 0;
    }

    for (int i = 0; i < PAGE_TABLE_ENTRY_NUM; i++)
    {
        page_table[i].valid = 0;
    }

    fp_addr = fopen(argv[1], "r");
    if (fp_addr == NULL)
    {
        printf("Error: open file %s failed!\n", argv[1]);
        exit(1);
    }
    fp_out = fopen("out.txt", "w");
    if (fp_out == NULL)
    {
        printf("Error: create file out.txt failed!\n");
        exit(1);
    }
    fp_store = fopen("BACKING_STORE.bin", "rb");
    if (fp_store == NULL)
```

```
    {
        printf("Error: open file BACKING_STORE.bin failed!\n");
        exit(1);
    }

    TLB_stack_head = (stack_node *)malloc(sizeof(stack_node *));
    TLB_stack_tail = (stack_node *)malloc(sizeof(stack_node *));
    TLB_stack_head->prior = NULL;
    TLB_stack_head->next = TLB_stack_tail;
    TLB_stack_tail->prior = TLB_stack_head;
    TLB_stack_tail->next = NULL;

    frame_stack_head = (stack_node *)malloc(sizeof(stack_node *));
    frame_stack_tail = (stack_node *)malloc(sizeof(stack_node *));
    frame_stack_head->prior = NULL;
    frame_stack_head->next = frame_stack_tail;
    frame_stack_tail->prior = frame_stack_head;
    frame_stack_tail->next = NULL;

    free_frame_list = (list_node *)malloc(sizeof(list_node *));
    free_frame_list->next = NULL;
    for (int i = FRAME_NUM - 1; i >= 0; i--)
        insert_node(free_frame_list, i);

    free_TLB_list = (list_node *)malloc(sizeof(list_node *));
    free_TLB_list->next = NULL;
    for (int i = TLB_ENTRY_NUM - 1; i >= 0; i--)
        insert_node(free_TLB_list, i);
}

void clean()
{
    fclose(fp_addr);
    fclose(fp_out);
    fclose(fp_store);

    clean_stack(TLB_stack_head);
    clean_stack(frame_stack_head);
    clean_list(free_TLB_list);
    clean_list(free_frame_list);
}

int ger_frame_id(int page_id)
{
    int frame_id;
    frame_id = search_in_TLB(page_id);
    if (frame_id != -1)
    {
        return frame_id;
    }

    frame_id = search_in_page_table(page_id);
    return frame_id;
}

int search_in_TLB(int page_id)
{
    for (int i = 0; i < TLB_ENTRY_NUM; i++)
```

```
    {
        if (!TLB[i].valid)
            continue;

        if (TLB[i].page_id == page_id)
        {
            tlb_hit++;
            move_to_top(TLB_stack_head, TLB_stack_tail, i);
            move_to_top(frame_stack_head, frame_stack_tail,
TLB[i].frame_id);
            return TLB[i].frame_id;
        }
    }
    return -1;
}

int search_in_page_table(int page_id)
{
    int frame_id;
    if (page_table[page_id].valid == 1)
    {
        frame_id = page_table[page_id].frame_id;
    }
    // page fault
    else
    {
        page_fault++;
        frame_id = search_unused_frame();

        // page replacement
        if (frame_id == -1)
        {
            int frame_to_be_replace = get_buttom_id(frame_stack_tail);
            for (int i = 0; i < PAGE_TABLE_ENTRY_NUM; i++)
            {
                if (page_table[i].frame_id == frame_to_be_replace)
                {
                    page_table[i].valid = 0;
                }
            }
            frame_id = frame_to_be_replace;
        }

        fseek(fp_store, page_id * PAGE_SIZE, SEEK_SET);
        fread(memory[frame_id].data, sizeof(char), FRAME_SIZE, fp_store);

        page_table[page_id].frame_id = frame_id;
        page_table[page_id].valid = 1;
    }

    move_to_top(frame_stack_head, frame_stack_tail, frame_id);

    // update TLB
    int TLB_id = search_unused_TLB();
    if (TLB_id == -1)
    {
        TLB_id = get_buttom_id(TLB_stack_tail);
    }
```

```
    else
    {
        TLB[TLB_id].valid = 1;
    }

    TLB[TLB_id].page_id = page_id;
    TLB[TLB_id].frame_id = frame_id;
    move_to_top(TLB_stack_head, TLB_stack_tail, TLB_id);

    return frame_id;
}

// return index of an unused frame, return -1 if no such frame
int search_unused_frame()
{
    int id = remove_node(free_frame_list);
    return id;
}

// return index of an unused TLB entry, return -1 if no such TLB entry
int search_unused_TLB()
{
    int id = remove_node(free_TLB_list);
    return id;
}
```

6.2 data_structure.h

```
#define PAGE_TABLE_ENTRY_NUM 256
#define PAGE_SIZE 256
#define TLB_ENTRY_NUM 32
#define FRAME_SIZE 256
#define FRAME_NUM 256

typedef struct TLB_ITEM
{
    int valid; // valid bit
    int frame_id;
    int page_id;
} tlb_item;

typedef struct PAGE
{
    int valid; // valid bit
    int frame_id;
} page;

typedef struct FRAME
{
    char data[FRAME_SIZE];
} frame;

/****************************** stack begin
*********************************/

typedef struct STACK_NODE
{
```

```
    int id;
    struct STACK_NODE *prior;
    struct STACK_NODE *next;
} stack_node;

//move the node with id to the top of the stack (means it is the latest id been visited)
void move_to_top(stack_node *head, stack_node *tail, int id);

//get the bottom node's id (means it's the last recently visited id)
int get_buttom_id(stack_node *tail);

//release the memory
void clean_stack(stack_node *head);

/****************************** stack end *********************************/

/****************************** list begin ********************************/

typedef struct LIST_NODE
{
    int id;
    struct LIST_NODE *next;
} list_node;

//insert a new free node with id
void insert_node(list_node *head, int id);

//remove a free node (means it is selected to be used)
int remove_node(list_node *head);

//release the memory
void clean_list(list_node *head);

/****************************** list end **********************************/

```

6.3 data_structure.c

```
#include "data_structure.h"
#include <stdio.h>
#include <stdlib.h>

/****************************** stack begin ********************************/

void move_to_top(stack_node *head, stack_node *tail, int id)
{
    stack_node *tmp = head->next;
    while (tmp != tail)
    {
        // the node with id already exists
        if (tmp->id == id)
        {
            tmp->prior->next = tmp->next;
            tmp->next->prior = tmp->prior;
            head->next->prior = tmp;
```

```
            tmp->next = head->next;
            head->next = tmp;
            tmp->prior = head;
            return;
        }
        tmp = tmp->next;
    }

    // the node with id doesn't exists, create a new node
    tmp = (stack_node *)malloc(sizeof(stack_node *));
    tmp->id = id;
    head->next->prior = tmp;
    tmp->next = head->next;
    head->next = tmp;
    tmp->prior = head;
    return;
}

int get_buttom_id(stack_node *tail)
{
    return tail->prior->id;
}

void clean_stack(stack_node *head)
{
    while (head)
    {
        stack_node *tmp = head;
        head = head->next;
        free(tmp);
    }
}

/****************************** stack end ********************************/

/****************************** list begin ********************************/

void insert_node(list_node *head, int id)
{
    list_node *tmp = (list_node *)malloc(sizeof(list_node *));
    tmp->id = id;
    tmp->next = head->next;
    head->next = tmp;
}

int remove_node(list_node *head)
{
    if (head->next == NULL)
        return -1;

    int id = head->next->id;
    list_node *tmp = head->next;
    head->next = tmp->next;
    free(tmp);

    return id;
}

```

```c
void clean_list(list_node *head)
{
    while (head)
    {
        list_node *tmp = head;
        head = head->next;
        free(tmp);
    }
}

/****************************** list end ********************************/

```