Project8: **Designing a Virtual Memory Manager**

Yanjie Ze 519021910706

1 Introduction

This project consists of writing a program that translates logical to physical addresses for a virtual address space of size $2^{16} = 65,536 bytes$. Our program will read from a file containing logical addresses and, using a TLB and a page table, will translate each logical address to its corresponding physical address and output the value of the byte stored at the translated physical address.

Our learning goal is to use simulation to understand the steps involved in translating logical to physical addresses. This will include:

- resolving page faults using **demand paging**
- managing a **TLB**
- implementing a **page-replacement algorithm**

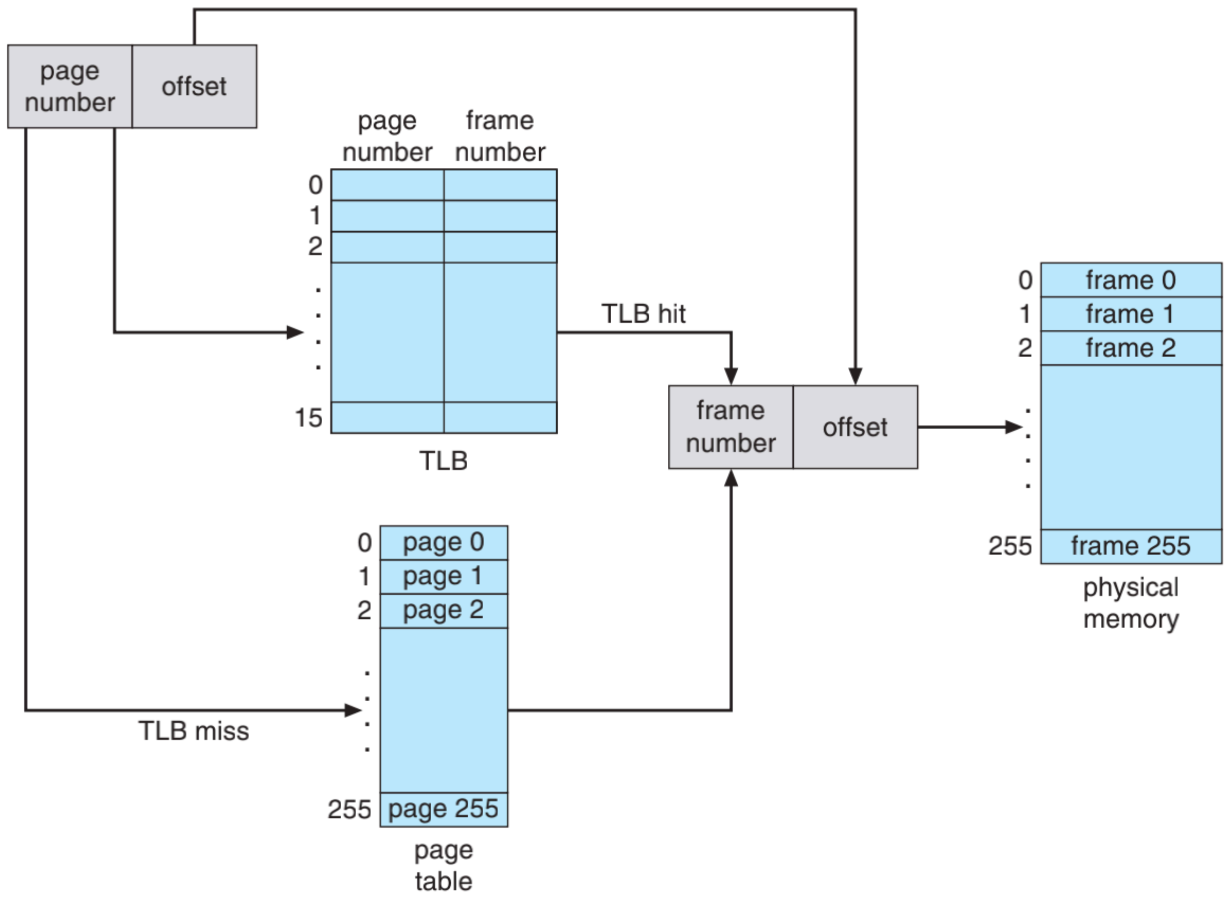
1.1 **Address Translation**

Our program will translate logical to physical addresses using a TLB and page table as outlined in Section 9.3.

First, the page number is extracted from the logical address, and the TLB is consulted.

- In the case of a TLB hit, the frame number is obtained from the TLB.
- In the case of a TLB miss, the page table must be consulted.

In the latter case, either the frame number is obtained from the page table, or a page fault occurs. A visual representation of the address translation process is:



1.2 Handling Page Faults

Our program will implement **demand paging** as described in Section 10.2.

The backing store is represented by the file **BACKING_STORE.bin**, a binary file of size 65,536 bytes.

When a page fault occurs, we will read in a 256-byte page from the file BACKING_STORE and store it in an available page frame in physical memory.

For example, if a logical address with page number 15 resulted in a page fault, your program would read in page 15 from BACKING_STORE (remember that pages begin at 0 and are 256 bytes in size) and store it in a page frame in physical memory. Once this frame is stored (and **the page table and TLB are updated**), subsequent accesses to page 15 will be resolved by either the TLB or the page table.

We will need to treat BACKING STORE.bin as a random-access file so that we can randomly seek to certain positions of the file for reading. We use the standard C library functions for performing I/O, including:

- fopen()
- fread()
- fseek()
- fclose()

1.3 Test File

We use the file **addresses.txt**, which contains integer values representing logical addresses ranging from 0 to 65535 (the size of the virtual address space). Our program will open this file, read each logical address and translate it to its corresponding physical address, and output the value of the signed byte at the physical address.

address.txt

```
16916
62493
30198
53683
40185
28781
24462
48399
 .....
 9929
45563
12107
```

1.4 Elementary Requirements

Our program should run as follows:

```
./a.out addresses.txt
```

Our program will read in the file addresses.txt, which contains 1,000 logical addresses ranging from 0 to 65535. Our program is to translate each logical address to a physical address and determine the contents of the signed byte stored at the correct physical address.

Our program is to output the following values:

1. The logical address being translated (the integer value being read from addresses.txt).
2. The corresponding physical address (what your program translates the logical address to).
3. The signed byte value stored in physical memory at the translated physical address.

1.5 Page Replacement

Thus far, this project has assumed that physical memory is the same size as the virtual address space. In practice, physical memory is typically much smaller than a virtual address space. This phase of the project now assumes using a smaller physical address space with 128 page frames rather than 256.

Thus, we should modify our program so that it keeps track of free page frames as well as implementing a page-replacement policy using either FIFO or LRU to resolve page faults when there is no free memory.

1.6 Statistics

After completion, our program is to report the following statistics:

1. **Page-fault rate** —The percentage of address references that resulted in page faults.
2. **TLB hit rate** —The percentage of address references that were resolved in the TLB.

Since the logical addresses in addresses.txt were generated randomly and do not reflect any memory access locality, do not expect to have a high TLB hit rate.

2 Implementation Details and Methods

2.1 Basic Data Structure

We define three new data structures, to make the creation of variables more convenient:

- `tlb_item` , represent the element of TLB. It consists of three elements:
    - **used_time** , the time it was used last time
    - **frame_id**
    - **page_id**

```
typedef struct TLB_ITEM
{
    int used_time;
    int frame_id;
    int page_id;
} tlb_item;
```

- `page_table_item` , represent the element of Page Table. It consists of:
    - **valid** , the valid bit
    - **frame_id**

```
typedef struct PAGE_TABLE_ITEM
{
    int valid;
    int frame_id;
} page_table_item;
```

- `memory_item` , represent the element of Memory. It consists of:
    - **used_time** , the time it was used last time
    - **data**

```
typedef struct MEMORY_ITEM
{
    int used_time;
    char data[FRAME_SIZE];
} memory_item;
```

By these definitions, we define **page_table**, **memory**, **TLB**.

```
#define PAGE_SIZE 256
#define FRAME_NUM 64
#define FRAME_SIZE 256
#define TLB_ENTRY_NUM 16
#define PAGE_TABLE_SIZE 256

page_table_item page_table[PAGE_TABLE_SIZE];
tlb_item TLB[TLB_ENTRY_NUM];
memory_item memory[FRAME_NUM];
```

2.2 Initialize Variables

First, we need to initialize **page_table**, **memory**, **TLB**, and give them original values.

Initialization of page table needs to set valid to 0 and set frame_id to -1.

```
/* initialize page table */
void page_table_init()
{
    for(int i=0; i<PAGE_TABLE_SIZE; i++)
    {
        page_table[i].valid = 0;// set to not valid
        page_table[i].frame_id = -1; // set to None
    }
}
```

Initialization of TLB needs to set **used_time** to -1 and others to -1 too.

```
/* initialize TLB */
void tlb_init()
{
    for(int i=0; i<TLB_ENTRY_NUM; i++)
    {
        TLB[i].used_time = -1;
        TLB[i].frame_id = -1;
        TLB[i].page_id = -1;
    }
}
```

Initialization of Memory:

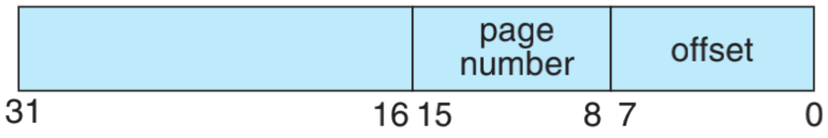
```
/* initialize memory*/
void memory_init()
{
    for(int i=0; i<FRAME_NUM; i++)
    {
        memory[i].used_time = -1;
    }
}
```

Then we include them in a function **initialize()** :

```
/* initialize page table, TLB, memory */
void initialize()
{
    page_table_init();
    tlb_init();
    memory_init();
    printf("Initialize Finish.\n");
}
```

2.3 Get and Parse Addresses

The address should be parsed like this:



Thus, we provide two functions to parse the address, by utilizing the right shift and the left shift.

get_page()

```
/* get the page num, given address*/
int get_page(int address)
{
    address = address>>8;
    return address;
}
```

get_offset()

```
/*get the offset, given address*/
int get_offset(int address)
{
    return address - (get_page(address)<<8);
}
```

Then, we read addresses from the file given, **addresses.txt** and parse one address at a time.

```
 addresses = fopen(argv[1], "r");
.....
  fscanf(addresses, "%u", &address);
.....
```

2.4 LRU Replacement in TLB

When the TLB does not hit, we will do replacement in TLB. In this project, we use **LRU(least recently used)** algorithm.

Before implementing it, we introduce this algorithm:

LRU replacement associates with each page the time of that page's last use.

When a page must be replaced, LRU chooses the page that has not been used for **the longest period of time**.

Based on this, when we do page replacement, we first find out the page that has not been used for the longest time, which means the time of the page is **the smallest**.

Then we update the new information in the old element.

```
/* LRU replacement for TLB*/
```

```
void TLB_LRU_Replacement(int page_id, int frame_id, int time)
{
    int min_time=time;
    int min_idx = 0;
    // find the least recently used
    for(int i=0; i<TLB_ENTRY_NUM; i++)
    {
        if(TLB[i].used_time<min_time)
        {
            min_time = TLB[i].used_time;
            min_idx = i;
        }
    }
    TLB[min_idx].frame_id = frame_id;
    TLB[min_idx].page_id = page_id;
    TLB[min_idx].used_time = time;
}
```

2.5 LRU Replacement in Memory

We also need to perform LRU replacement in memory, since the size of memory may be smaller than the number of pages.

What's more, after we find out the least recently used frame and do a page replacement, we also need to update the page table and TLB.

Thus, the whole implementation has the following procedures:

- find out the least recently used frame in memory
- find out the corresponding pages in page table, set valid=-1
- seek the new data and store it in the new frame
- return the frame id

When seeking data, we can use two functions: `fseek()` and `fread()` .

The codes below show the whole implementation.

```
/* LRU Replacement for Memory
```

```
    Update new data
    return the frame id */
int memory_LRU_Replacement(int page_id, int time)
{
    int min_time=time;
    int min_idx = 0;
    // find the least recently used
    for(int i=0; i<FRAME_NUM; i++)
    {
        if(memory[i].used_time<min_time)
        {
            min_time = memory[i].used_time;
            min_idx = i;
        }
    }
    memory[min_idx].used_time = time;

    //find the old page id, and set invalid
    for(int i=0; i<PAGE_TABLE_SIZE; i++)
    {
        if(page_table[i].frame_id==min_idx)
        {
            page_table[i].valid = -1;
        }
    }

    // seek data
    fseek(backing_store, page_id*PAGE_SIZE, SEEK_SET);
    fread(memory[min_idx].data, sizeof(char), FRAME_SIZE, backing_store);

    return min_idx;
}
```

2.6 Main Part

After preparing many functions, we can fulfill the main part of the program.

First, after reading one address, we decode it.

```
//decode address
        page_id = get_page(address);
        offset = get_offset(address);
```

Second, search the page in **TLB**. If TLB hits, we add the value of **tlb_hit** and update the used time.

```
// search in TLB
        int tlb_find=0;
        for(int i=0; i<TLB_ENTRY_NUM; i++)
        {
            if(page_id==TLB[i].page_id)
            {
                tlb_hit ++;
                tlb_find = 1;
                frame_id = TLB[i].frame_id;
                memory[frame_id].used_time = time;
                TLB[i].used_time = time;
                break;
            }
        }
```

Third, if the page is not found in TLB, we search Page Table.

If we find in Page Table, update the used time and also update TLB by calling `TLB_LRU_Replacement(page_id, frame_id, time).`

If we do not find the page in page table, we need to do page replacement, by calling **`frame_id = memory_LRU_Replacement(page_id, time)`** , to get the allocated new frame id.

```
// not find in TLB ,search in page table
         int page_find=0;
         if(!tlb_find)
         {
             // valid = 1, find page
             if(page_table[page_id].valid==1)
             {
                 page_find = 1;
                 frame_id = page_table[page_id].frame_id;
                 memory[frame_id].used_time = time;
                 TLB_LRU_Replacement(page_id, frame_id, time); // Update TLB

             }
             else // valid = -1, not find, page fault , do demand paging
             {
                 page_fault++;

                 // demand paging
                 frame_id = memory_LRU_Replacement(page_id, time);

                 //update page table
                 page_table[page_id].frame_id = frame_id;
                 page_table[page_id].valid = 1;// set valid

                 //update TLB
                 TLB_LRU_Replacement(page_id, frame_id, time);

             }
         }
```

Forth, we have got the frame id, so we will calculate the physical address and fetch data from the memory.

Also, we output the result in **result_file**, in the format of **Virtual address, Physical address, Value**.

```
 // calculate physical address and get data
        int physical_address = frame_id*FRAME_SIZE + offset;
        int data = memory[frame_id].data[offset];

        // output the result
        fprintf(result_file, "Virtual address: %d Physical address: %d
Value: %d\n", address, physical_address, data);
        fscanf(addresses, "%u", &address);
```

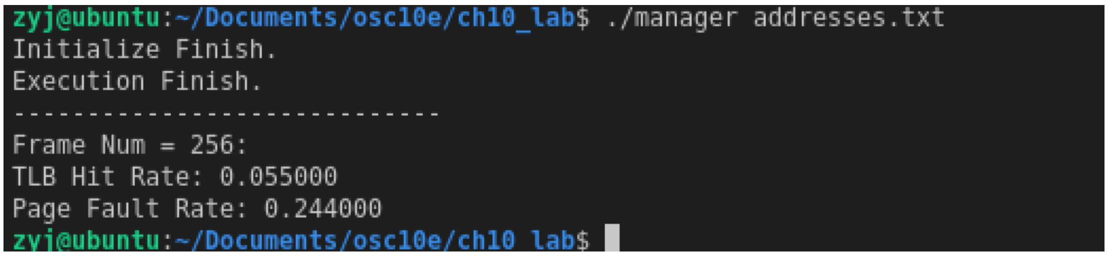
The full codes of the main part are shown here:

```
int main(int argc, char*argv[])
{

    addresses = fopen(argv[1], "r");
    backing_store = fopen("BACKING_STORE.bin", "rb");
    result_file = fopen("result.txt", "w");

    // init
    initialize();

    // addrees parse arg
    int address;
    int page_id;
    int frame_id;
    int offset;

    // count arg
    int count=0;
    int tlb_hit = 0;
    int page_fault = 0;
    int time = 0;

    fscanf(addresses, "%u", &address);
    while(!feof(addresses))
    {
        count ++;
        time ++;

        //decode address
        page_id = get_page(address);
        offset = get_offset(address);


        // search in TLB
        int tlb_find=0;
        for(int i=0; i<TLB_ENTRY_NUM; i++)
        {
            if(page_id==TLB[i].page_id)
            {
                tlb_hit ++;
                tlb_find = 1;
                frame_id = TLB[i].frame_id;
                memory[frame_id].used_time = time;
                TLB[i].used_time = time;
                break;
            }
        }

        // not find in TLB ,search in page table
        int page_find=0;
        if(!tlb_find)
        {
            // valid = 1, find page
            if(page_table[page_id].valid==1)
            {
                page_find = 1;
                frame_id = page_table[page_id].frame_id;
                memory[frame_id].used_time = time;
                TLB_LRU_Replacement(page_id, frame_id, time); // Update TLB

            }
            else // valid = -1, not find, page fault , do demand paging
            {
                page_fault++;

                // demand paging
                frame_id = memory_LRU_Replacement(page_id, time);

                //update page table
                page_table[page_id].frame_id = frame_id;
                page_table[page_id].valid = 1;// set valid

                //update TLB
                TLB_LRU_Replacement(page_id, frame_id, time);

            }
        }
        // calculate physical address and get data
        int physical_address = frame_id*FRAME_SIZE + offset;
        int data = memory[frame_id].data[offset];

        // output the result
        fprintf(result_file, "Virtual address: %d Physical address: %d
Value: %d\n", address, physical_address, data);
        fscanf(addresses, "%u", &address);
    }

    // finish
    fclose(addresses);
    fclose(backing_store);
    fclose(result_file);

    printf("Execution Finish.\n");
    printf("-----------------------------\n");

    //statistics
    double page_fault_rate = page_fault/(double)count;
    double tlb_hit_rate = tlb_hit/(double)count;
    printf("Frame Num = %d:\n", FRAME_NUM);
    printf("TLB Hit Rate: %f\n", tlb_hit_rate);
    printf("Page Fault Rate: %f\n", page_fault_rate);

}
```

3 Program Results

3.1 Compare `result.txt` with `correct.txt`

When **frame_num = 256**, we get the following result:



```
zyj@ubuntu:~/Documents/osc10e/ch10_lab$ ./manager addresses.txt
Initialize Finish.
Execution Finish.
-----------------------------
Frame Num = 256:
TLB Hit Rate: 0.055000
Page Fault Rate: 0.244000
zyj@ubuntu:~/Documents/osc10e/ch10_lab$
```

The output file is `result.txt`.

So whether `result.txt` is the same as `correct.txt`?

We could use bash to compare them:

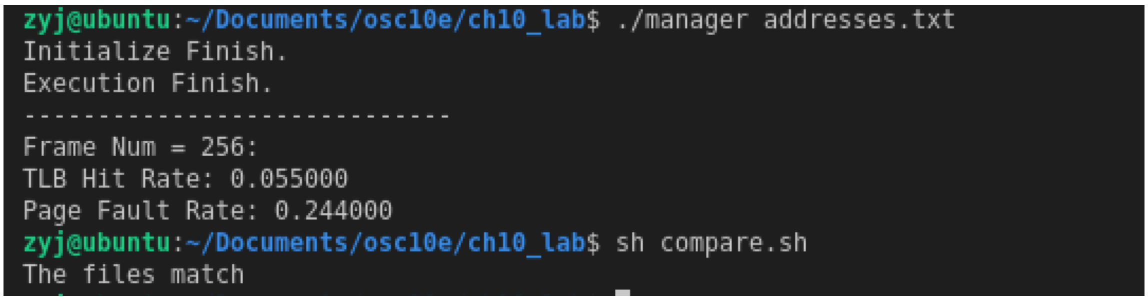
`compare.sh`

```
if cmp -s "correct.txt" "result.txt"
then
   echo "The files match"
else
   echo "The files are different"
fi
```

Enter:

```
sh compare.sh
```

Then we get:



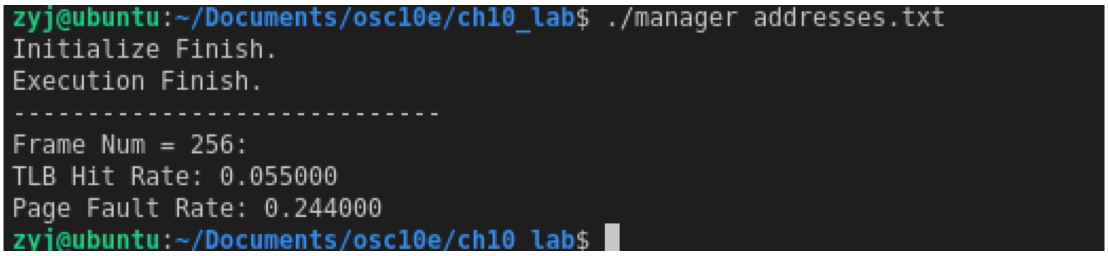
```
zyj@ubuntu:~/Documents/osc10e/ch10_lab$ ./manager addresses.txt
Initialize Finish.
Execution Finish.
----------------------------
Frame Num = 256:
TLB Hit Rate: 0.055000
Page Fault Rate: 0.244000
zyj@ubuntu:~/Documents/osc10e/ch10_lab$ sh compare.sh
The files match
```

Thus, we succeed in getting the correcet results!

3.2 Statistics

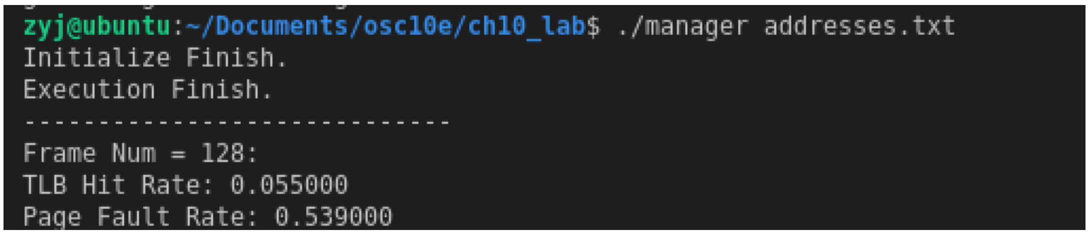
We change the frame num, to see the **TLB hit rate** and **Page Fault Rate**.

When **frame_num=256**:



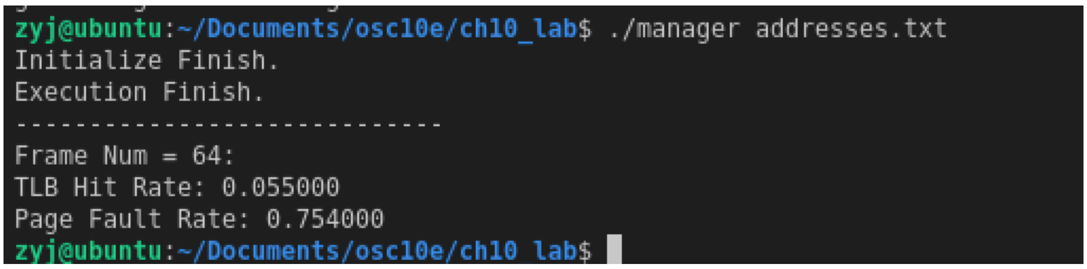
```
zyj@ubuntu:~/Documents/osc10e/ch10_lab$ ./manager addresses.txt
Initialize Finish.
Execution Finish.
----------------------------
Frame Num = 256:
TLB Hit Rate: 0.055000
Page Fault Rate: 0.244000
zyj@ubuntu:~/Documents/osc10e/ch10_lab$
```

When **frame_num=128**:



```
zyj@ubuntu:~/Documents/osc10e/ch10_lab$ ./manager addresses.txt
Initialize Finish.
Execution Finish.
----------------------------
Frame Num = 128:
TLB Hit Rate: 0.055000
Page Fault Rate: 0.539000
```

When **frame_num=64**:



```
zyj@ubuntu:~/Documents/osc10e/ch10_lab$ ./manager addresses.txt
Initialize Finish.
Execution Finish.
-----------------------------
Frame Num = 64:
TLB Hit Rate: 0.055000
Page Fault Rate: 0.754000
zyj@ubuntu:~/Documents/osc10e/ch10_lab$
```

In conclusion, when the frame num gets smaller, the **page fault rate** goes higher, and the TLB hit rate keeps low.

4 Conclusion and Thoughts

In project 8, we finish wrting a simulation of paging, which is related to the management of **Virtual Address** and **Physical Address**.

In the very begining, it's hard for me to get started, since I have no idea how to construct the whole program. The reason lies on that I have not yet understood totally about paging. After reading the material and the books many times and spending a lot of time on programming, I successfully finish the whole project, and do some interesting statistics.

One fatal mistake I have made is that I add an extra `break` in `memory_LRU_Replacement()`:

Correct Version:

```
//find the old page id, and set invalid
    for(int i=0; i<PAGE_TABLE_SIZE; i++)
    {
        if(page_table[i].frame_id==min_idx)
        {
            page_table[i].valid = -1;
        }
    }
```

Wrong Version:

```
//find the old page id, and set invalid
    for(int i=0; i<PAGE_TABLE_SIZE; i++)
    {
        if(page_table[i].frame_id==min_idx)
        {
            page_table[i].valid = -1;
          break
        }
    }
```

One `break` makes the result of my program different from my mates, which makes me check the whole program many times and rethink the logic of program many times. Finally, it's done. This mistake is caused by t**hat there exists more than one pages correspoding to the same frame**. I have not considered this when wrting the wrong codes.

In conclusion, we learn about how paging works in this project, and know deeper about **Virtual Address**. What's more, I understand the importance of the strict logic and practise my porgamming skills further.