Project 8--Designing a Virtual Memory Manager

陈思远 518021910567

1. 实验内容和目标

1. 设计并实现一个可以将逻辑地址转化为内存中的物理地址的模拟器。
2. 使用TLB加快地址转化的速度，使用FIFO或LRU替换策略更新已满的TLB。
3. 处理缺页异常，使用FIFO或LRU替换策略更新已满的物理内存。

2. Virtual Memory Manager

2.1 要求与实现

本次project要求设计并实现一个虚拟内存的管理器。首先读取一个写有逻辑地址的文件，然后从逻辑地址中解析出页号(page number)和偏移量(offset)，再根据页号从TLB或页表(page table)中寻找对应的帧号(frame number)，若未找到，则处理缺页异常，从backing store中把缺失的页调入内存中，并得到对应的帧号。最后即可根据帧号和偏移量得到物理地址，并从物理地址中读出对应的值。

$$\text{physical address} = \text{frame number} \times \text{frame size} + \text{offset}$$

主函数如下。

```
int main(int argc,char*argv[]) {
    if(argc!=2)
        return -1;

    FILE *in;
    in=fopen(argv[1],"r");
    out=fopen("myoutput.txt","w");
    store=fopen("BACKING_STORE.bin","r");
    char addr[10];
    while(fgets(addr,9,in)!=NULL){
        total++;
        int logi_addr=atoi(addr);   //逻辑地址
        int offset=0b11111111&logi_addr;    //用mask解析偏移量
        int page_number=(logi_addr>>8)&0b11111111;  //用mask和移位解析页号
        int frame_number=get_framenumber(page_number);  //根据页号得到帧号
        int phy_addr=frame_number*FrameSize+offset; //根据帧号计算物理地址
        char value=getValue(phy_addr);  //根据物理地址访问内存
        print_res(logi_addr,phy_addr,value);    //打印结果
    }
    printf("page fault rate=%.2f%%\n",(float)pagefault/total*100);
    printf("TLB hit rate=%.2f%%\n",(float)TLBhit/total*100);
    fclose(in);
    fclose(out);
    fclose(store);

    return 0;
}
```

2.1.1 数据结构

```
#define TLBsize 16
#define PageTablesize 256
#define PageNum 256
#define FrameNum 128
#define FrameSize 256

//TLB[i][0]: valid/invalid bit; TLB[i][1]: page number; TLB[i][2]: frame number
int TLB[TLBsize][3]={0};

//physical memory
char phy_memory[FrameNum*FrameSize];

/*page table*/
/*i:page number; pagetable[i][0]:valid/invalid bit; pagetable[i][1]: frame
number*/
int pagetable[PageTablesize][2]={0};

//backing store file
FILE* store;
//output file
FILE* out;

int available_frame=0;
int available_TLB=0;

/*statics*/
int TLBhit=0;
int pagefault=0;
int total=0;
```

2.1.2 地址转换（Address Translation）

调用`get_framenumber()`函数可以得到页号对应的帧号。在这个函数中，首先搜索TLB，寻找有无该页号。若有，可以直接返回帧号；若无，则在page table中以页号作为索引查询帧号，若valid位为1，表示该帧号有效，可以在物理内存中找到，若valid位为0，则说明该页并不在内存中，发生缺页异常。

```
int get_framenumber(int pagenumber){
    int framenumber=search_TLB(pagenumber);
    if(framenumber!=-1){    // TLB hit!
        return framenumber;
    }
    if(pagetable[pagenumber][0]==1){     //page is valid!
```

```
        update_TLB(pagenumber,pagetable[pagenumber][1]);
        return pagetable[pagenumber][1];
    }else{              //page fault!
        pagefault++;
        pagetable[pagenumber][0]=1;
        pagetable[pagenumber][1]=load_new_page(pagenumber);
        update_TLB(pagenumber,pagetable[pagenumber][1]);
        return pagetable[pagenumber][1];
    }
}
```

2.1.3 处理缺页异常（Handle Page Faults）

如果出现了缺页异常，就要从backing store中把缺失的页调入内存，这里采用读文件的方式读入缺失的页，暂存入buffer中。接着，就要寻找物理内存中空闲的帧，用来存放调入的页的内容。这里调用了一个`find_empty_frame()`的函数，返回帧号。

```
int load_new_page(int pagenumber){
    int off=pagenumber*FrameSize;
    fseek(store,off,SEEK_SET);
    char buffer[FrameSize];
    fread(buffer, sizeof(char),FrameSize,store);

    int framenum=find_empty_frame();

    for(int i=0;i<FrameSize;i++){
        phy_memory[framenum*FrameSize+i]=buffer[i];
    }
    return framenum;
}
```

在`find_empty_frame()`函数中，首先寻找有无未被使用的帧。因为内存中的帧采用按照访问时间从前到后顺序存放的方式，且内存未满时每次调入新的页，帧的计数都会自增，所以未被使用的帧号就是当前帧的计数。若内存已满，则需要寻找一个victim，用新调入的帧替换掉victim，我采用FIFO的替换策略，先进入的先被替换。替换的具体操作就是在页表中把对应的valid位置为0，表示该帧不在内存中。

```
int find_empty_frame() {
    if(available_frame<FrameNum) {
        return available_frame++;
    }
    int victim=(available_frame++)%FrameNum;
    for(int i=0;i<PageNum;i++){
        if(pagetable[i][0]&&pagetable[i][1]==victim){
            pagetable[i][0]=0;
            break;
        }
    }
    return victim;
}
```

2.1.4 TLB

TLB用于加快页号到帧号的转化过程。如果可以在TLB中找到该页号，就可以直接返回帧号，不必再到page table里找，这里采用`search_TLB()`函数实现这个功能。

```
int search_TLB(int page_num){
    for(int i=0;i<TLBsize;i++){
        if(TLB[i][0]==1 && page_num==TLB[i][1]){   //TLB hit
            TLBhit++;
            return TLB[i][2];//return frame number
        }
    }
    return -1;  //TLB miss
}
```

如果发生了TLB miss，就需要去页表中找对应的帧号，找到之后，由于局部性原理，需要把刚使用过的页号和帧号加入TLB，方便以后的查找。如果TLB未满，直接选择下一个空位加入即可；若TLB已满，需要寻找victim，这里依然采用FIFO的策略。具体代码如下。

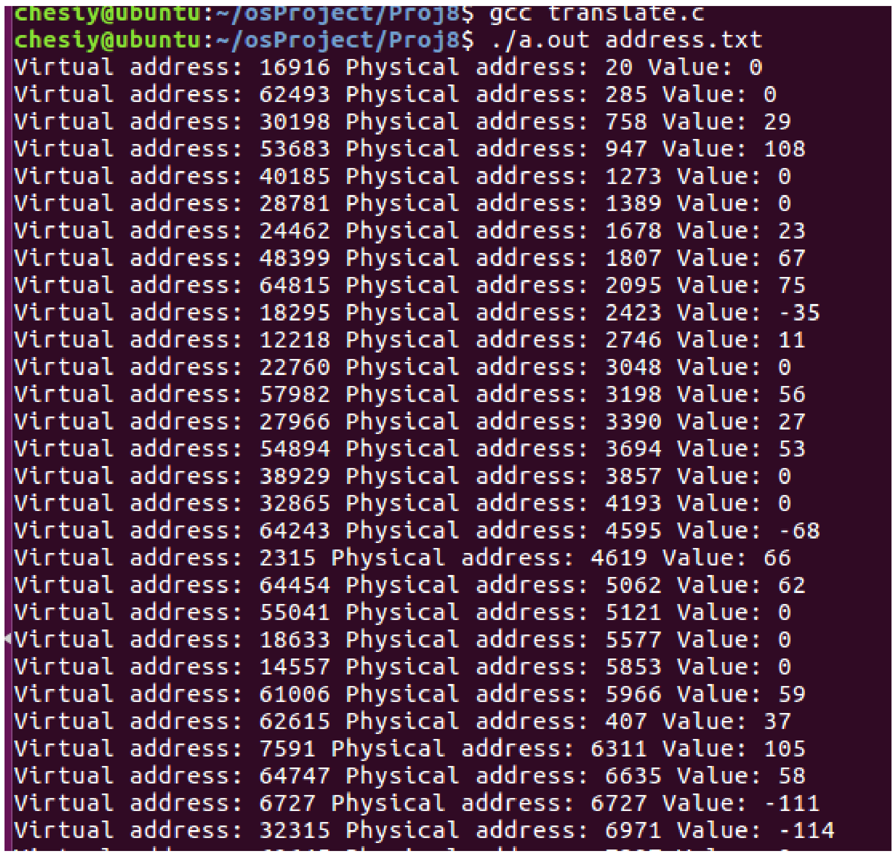
```
void update_TLB(int page_num,int frame_num){
    int victim=available_TLB%TLBsize;
    available_TLB=(available_TLB+1)%TLBsize;
    TLB[victim][0]=1;
    TLB[victim][1]=page_num;
    TLB[victim][2]=frame_num;
}
```

2.1.5 访问内存和打印结果

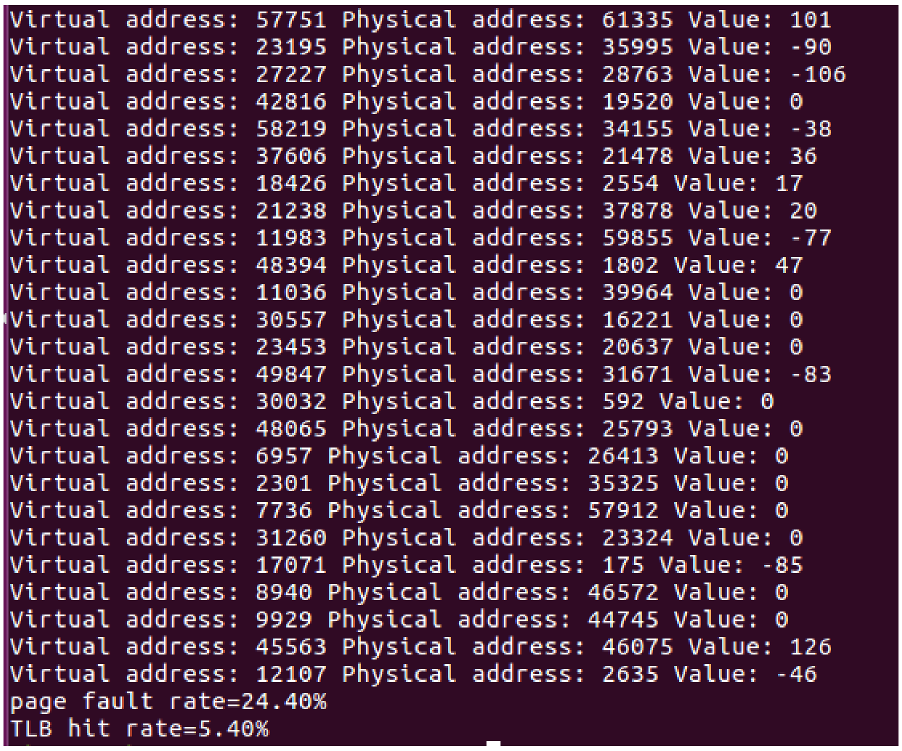
```
char getValue(int phy_addr){
    return phy_memory[phy_addr];
}
void print_res(int vir,int phy,char val){
    fprintf(out,"Virtual address: %d Physical address: %d Value:
%d\n",vir,phy,val);
    printf("Virtual address: %d Physical address: %d Value: %d\n",vir,phy,val);
}
```

2.2 结果

首先，我测试了frame number=256（不需要替换页）的情况。部分截图如下。page fault的比率为24.40%，TLB hit的比率为5.40%。

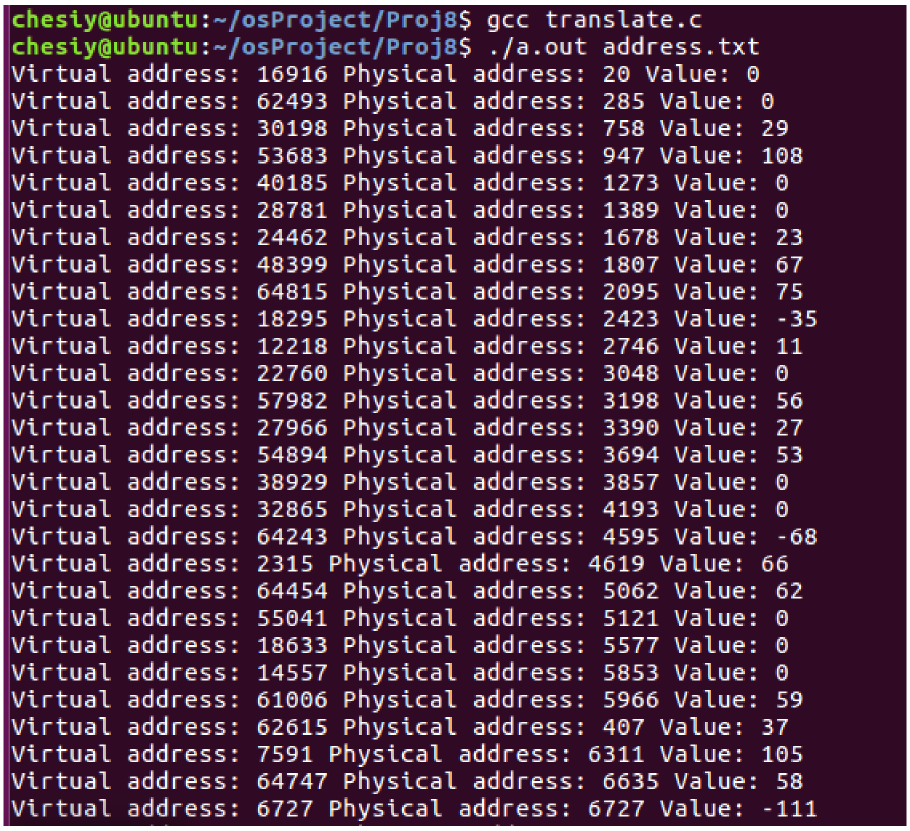


```
chesiy@ubuntu:~/osProject/Proj8$ gcc translate.c
chesiy@ubuntu:~/osProject/Proj8$ ./a.out address.txt
Virtual address: 16916 Physical address: 20 Value: 0
Virtual address: 62493 Physical address: 285 Value: 0
Virtual address: 30198 Physical address: 758 Value: 29
Virtual address: 53683 Physical address: 947 Value: 108
Virtual address: 40185 Physical address: 1273 Value: 0
Virtual address: 28781 Physical address: 1389 Value: 0
Virtual address: 24462 Physical address: 1678 Value: 23
Virtual address: 48399 Physical address: 1807 Value: 67
Virtual address: 64815 Physical address: 2095 Value: 75
Virtual address: 18295 Physical address: 2423 Value: -35
Virtual address: 12218 Physical address: 2746 Value: 11
Virtual address: 22760 Physical address: 3048 Value: 0
Virtual address: 57982 Physical address: 3198 Value: 56
Virtual address: 27966 Physical address: 3390 Value: 27
Virtual address: 54894 Physical address: 3694 Value: 53
Virtual address: 38929 Physical address: 3857 Value: 0
Virtual address: 32865 Physical address: 4193 Value: 0
Virtual address: 64243 Physical address: 4595 Value: -68
Virtual address: 2315 Physical address: 4619 Value: 66
Virtual address: 64454 Physical address: 5062 Value: 62
Virtual address: 55041 Physical address: 5121 Value: 0
Virtual address: 18633 Physical address: 5577 Value: 0
Virtual address: 14557 Physical address: 5853 Value: 0
Virtual address: 61006 Physical address: 5966 Value: 59
Virtual address: 62615 Physical address: 407 Value: 37
Virtual address: 7591 Physical address: 6311 Value: 105
Virtual address: 64747 Physical address: 6635 Value: 58
Virtual address: 6727 Physical address: 6727 Value: -111
Virtual address: 32315 Physical address: 6971 Value: -114
```

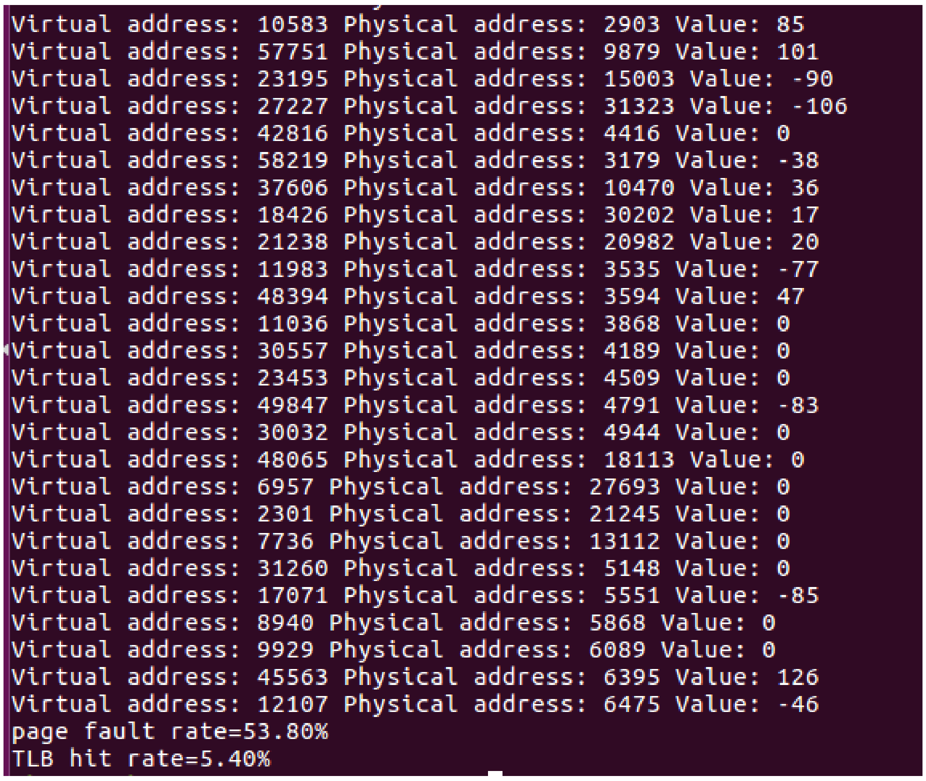


```
Virtual address: 57751 Physical address: 61335 Value: 101
Virtual address: 23195 Physical address: 35995 Value: -90
Virtual address: 27227 Physical address: 28763 Value: -106
Virtual address: 42816 Physical address: 19520 Value: 0
Virtual address: 58219 Physical address: 34155 Value: -38
Virtual address: 37606 Physical address: 21478 Value: 36
Virtual address: 18426 Physical address: 2554 Value: 17
Virtual address: 21238 Physical address: 37878 Value: 20
Virtual address: 11983 Physical address: 59855 Value: -77
Virtual address: 48394 Physical address: 1802 Value: 47
Virtual address: 11036 Physical address: 39964 Value: 0
Virtual address: 30557 Physical address: 16221 Value: 0
Virtual address: 23453 Physical address: 20637 Value: 0
Virtual address: 49847 Physical address: 31671 Value: -83
Virtual address: 30032 Physical address: 592 Value: 0
Virtual address: 48065 Physical address: 25793 Value: 0
Virtual address: 6957 Physical address: 26413 Value: 0
Virtual address: 2301 Physical address: 35325 Value: 0
Virtual address: 7736 Physical address: 57912 Value: 0
Virtual address: 31260 Physical address: 23324 Value: 0
Virtual address: 17071 Physical address: 175 Value: -85
Virtual address: 8940 Physical address: 46572 Value: 0
Virtual address: 9929 Physical address: 44745 Value: 0
Virtual address: 45563 Physical address: 46075 Value: 126
Virtual address: 12107 Physical address: 2635 Value: -46
page fault rate=24.40%
TLB hit rate=5.40%
```

接着，我测试了frame number=128（需要替换页）的情况。部分截图如下。page fault的比率为53.80%，TLB hit的比率为5.40%。可以看到，物理内存减小后，发生缺页错误的可能性显著提升。



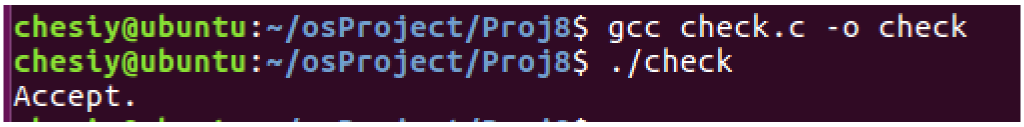
```
chesiy@ubuntu:~/osProject/Proj8$ gcc translate.c
chesiy@ubuntu:~/osProject/Proj8$ ./a.out address.txt
Virtual address: 16916 Physical address: 20 Value: 0
Virtual address: 62493 Physical address: 285 Value: 0
Virtual address: 30198 Physical address: 758 Value: 29
Virtual address: 53683 Physical address: 947 Value: 108
Virtual address: 40185 Physical address: 1273 Value: 0
Virtual address: 28781 Physical address: 1389 Value: 0
Virtual address: 24462 Physical address: 1678 Value: 23
Virtual address: 48399 Physical address: 1807 Value: 67
Virtual address: 64815 Physical address: 2095 Value: 75
Virtual address: 18295 Physical address: 2423 Value: -35
Virtual address: 12218 Physical address: 2746 Value: 11
Virtual address: 22760 Physical address: 3048 Value: 0
Virtual address: 57982 Physical address: 3198 Value: 56
Virtual address: 27966 Physical address: 3390 Value: 27
Virtual address: 54894 Physical address: 3694 Value: 53
Virtual address: 38929 Physical address: 3857 Value: 0
Virtual address: 32865 Physical address: 4193 Value: 0
Virtual address: 64243 Physical address: 4595 Value: -68
Virtual address: 2315 Physical address: 4619 Value: 66
Virtual address: 64454 Physical address: 5062 Value: 62
Virtual address: 55041 Physical address: 5121 Value: 0
Virtual address: 18633 Physical address: 5577 Value: 0
Virtual address: 14557 Physical address: 5853 Value: 0
Virtual address: 61006 Physical address: 5966 Value: 59
Virtual address: 62615 Physical address: 407 Value: 37
Virtual address: 7591 Physical address: 6311 Value: 105
Virtual address: 64747 Physical address: 6635 Value: 58
Virtual address: 6727 Physical address: 6727 Value: -111
```



```
Virtual address: 10583 Physical address: 2903 Value: 85
Virtual address: 57751 Physical address: 9879 Value: 101
Virtual address: 23195 Physical address: 15003 Value: -90
Virtual address: 27227 Physical address: 31323 Value: -106
Virtual address: 42816 Physical address: 4416 Value: 0
Virtual address: 58219 Physical address: 3179 Value: -38
Virtual address: 37606 Physical address: 10470 Value: 36
Virtual address: 18426 Physical address: 30202 Value: 17
Virtual address: 21238 Physical address: 20982 Value: 20
Virtual address: 11983 Physical address: 3535 Value: -77
Virtual address: 48394 Physical address: 3594 Value: 47
Virtual address: 11036 Physical address: 3868 Value: 0
Virtual address: 30557 Physical address: 4189 Value: 0
Virtual address: 23453 Physical address: 4509 Value: 0
Virtual address: 49847 Physical address: 4791 Value: -83
Virtual address: 30032 Physical address: 4944 Value: 0
Virtual address: 48065 Physical address: 18113 Value: 0
Virtual address: 6957 Physical address: 27693 Value: 0
Virtual address: 2301 Physical address: 21245 Value: 0
Virtual address: 7736 Physical address: 13112 Value: 0
Virtual address: 31260 Physical address: 5148 Value: 0
Virtual address: 17071 Physical address: 5551 Value: -85
Virtual address: 8940 Physical address: 5868 Value: 0
Virtual address: 9929 Physical address: 6089 Value: 0
Virtual address: 45563 Physical address: 6395 Value: 126
Virtual address: 12107 Physical address: 6475 Value: -46
page fault rate=53.80%
TLB hit rate=5.40%
```

最后，我编写了一份测试代码，检查输出的结果与`correct.txt`的一致性，结果是完全一致的。

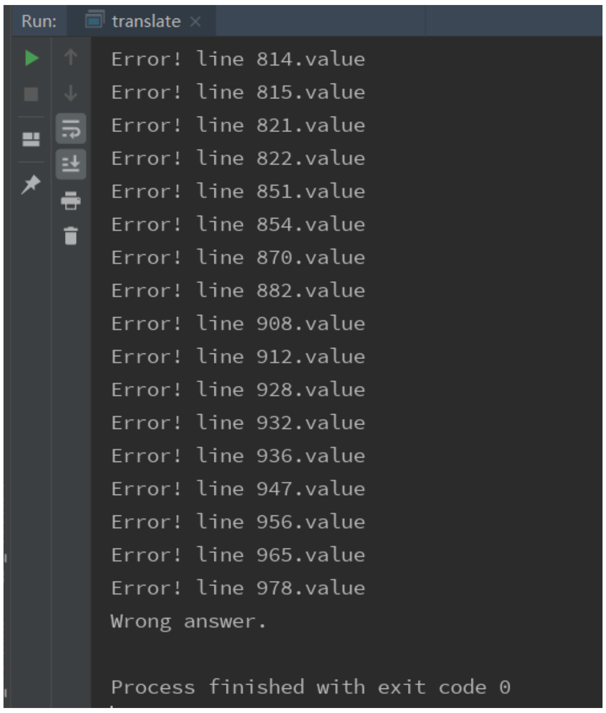
```c
# include <stdio.h>
# include <string.h>
int main() {
    FILE *out = fopen("myoutput.txt", "r");
    FILE *ans = fopen("correct.txt", "r");
    int accept = 1;
    int ans_vir_addr,out_vir_addr;
    int ans_phy_addr,out_phy_addr;
    int ans_val, out_val, cnt = 0;
    while (~fscanf(ans, "Virtual address: %d Physical address: %d Value: %d\n",
                   &ans_vir_addr,&ans_phy_addr,&ans_val)) {
        if (fscanf(out, "Virtual address: %d Physical address: %d Value: %d\n",
                   &out_vir_addr,&out_phy_addr,&out_val) == EOF) {
            printf("Wrong length!\n");
            accept = 0;
            break;
        }
        if(ans_vir_addr!=out_vir_addr){
            printf("Error! line %d. virtual address\n", cnt);
            accept=0;
        }
        if(ans_phy_addr!=out_phy_addr){
            printf("Error! line %d. physical address\n", cnt);
            accept=0;
        }
        if (ans_val != out_val){
            printf("Error! line %d. value\n", cnt);
            accept = 0;
        }
        cnt++;
    }
    if (accept == 0) printf("Wrong answer.\n");
    else printf("Accept.\n");
    fclose(out);
    fclose(ans);
    return 0;
}
```



```
chesiy@ubuntu:~/osProject/Proj8$ gcc check.c -o check
chesiy@ubuntu:~/osProject/Proj8$ ./check
Accept.
```

3. 总结与思考

在本次project中，我完成了一个可以将逻辑地址转化为内存中的物理地址的模拟器。在这个过程中，我复习了虚拟内存管理的内容，在实践中理顺了一个逻辑地址怎样一步一步地转化为物理地址的过程，一些原本抽象的概念也更加具体清晰了。在编写程序的过程中，我遇到了一个比较棘手的问题。一开始我在clion上写程序，用的是Mingw编译器。无论怎么改代码逻辑，输出总有一小部分是错误的（逻辑地址到物理地址的转化是正确的，但从内存中取出来的值是错误的，用测试代码跑的结果如下）。



```
Error! line 814.value
Error! line 815.value
Error! line 821.value
Error! line 822.value
Error! line 851.value
Error! line 854.value
Error! line 870.value
Error! line 882.value
Error! line 908.value
Error! line 912.value
Error! line 928.value
Error! line 932.value
Error! line 936.value
Error! line 947.value
Error! line 956.value
Error! line 965.value
Error! line 978.value
Wrong answer.

Process finished with exit code 0
```

我在尝试了许多办法无果之后，用同样的代码在虚拟机里用gcc编译后运行了一遍，居然得到了正确答案。我这才意识到是编译器的问题。同样的代码，不同的编译器编译出来的结果是不同的。仔细观察我的代码后，我意识到问题可能出在从char到int的强制类型转化，不同的编译器可能补位的方式不同。这给了我一个教训，以后写代码一定要更加规范，必要的时候用`uint32_t`，`uint8_t`等比较安全的类型，这样才能避免编译器带来的问题。