CSCI 3150 Introduction to Operating Systems:

Assignment Three

Deadline: 18:00:00 p.m., Mon, April 17th
Total Marks: 100
Topics: Virtual Memory, Address Translation and Swapping

Mandatory (100 marks, 2 problems)

1. **Address Translation** (40 marks)

Suppose a machine uses the **word addressing (A physical address of memory on a machine uniquely identify one word in the memory)** and **two-level paging**. The word length of the machine is 32 bits. The page size is 32 bytes, the page directory has 4 entries, and the page table (PT) has 8 entries. You need to answer the following questions.

a) What is the size of the virtual address space? How many bits does a virtual address have? How many bits should be reserved for the page directory index, page-table index and the offset respectively?

b) For virtual address space defined by the two-level page table as shown in Figure 1, given the following virtual addresses: **22, 68, 150, 245.**
Which of the above addresses would be mapped? If the virtual address is mapped, what is its corresponding physical address?

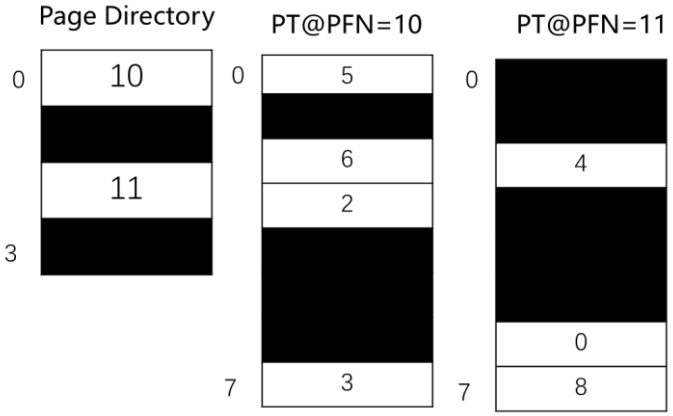

Figure 1

c) A program has three segments that should be loaded and mapped. The base and bound expression for these three segments is as the followings:

Index	Segment	Base	Size
0	Code	16	24
1	Data	48	32
2	Stack	240	16

We assumed that the code segment is loaded at physical memory address range $[0,24)$, the data segment is loaded at physical memory address range [24,56), and the stack segment is loaded at physical memory address range [56,72). You are required to **fill the page tables (PT)** in Figure 2 to map the physical address of these segments to the corresponding virtual address. If

some entries in the page table are not used, you can just leave them empty.

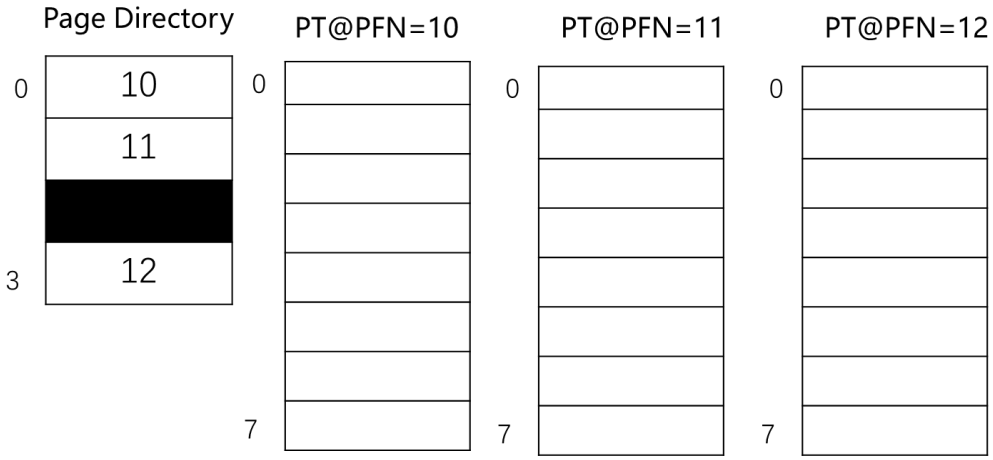

Figure 2

2. **Swapping** (60 marks)

In this problem, you would be guided to implement two commonly used page swapping policies, **FIFO and LRU** algorithm **simply using the single linked list**. **Using array to mark the recency is not allowed.**

We provide a framework for you to implement these two algorithms. The followings are some descriptions about the framework.

The node in the single linked list is defined in node.h file as follows:

```
typedef struct node{
        int page_no;
        struct node * next;
} node_t;
```

The *node_t* data type defines the node in the single linked list. *Page_no* refers to page number in the reference row, and *next* is the pointer pointing to the next node in the single linked list. The header of FIFO cache and LRU cache is defined in main.c as *head_fifo* and *head_lru*.

In this framework, you are required to implement functions *FIFO_cache_put* and *LRU_cache_put* in fifo.c and lru.c. The declaration of these two functions are as follows:

```
int FIFO_cache_put(int Newpageno, node_t **head_fifo, int cache_capacity);
int LRU_cache_put(int Newpageno, node_t **head_lru, int cache_capacity);
```

These two functions are intended for assessing each page in the reference row when using FIFO and LRU cache. Parameter *Newpageno* is the page number intended to be accessed. Parameter *head_fifo* and *head_lru* are the pointers pointing at the header of FIFO and LRU cache. Parameter *cache_capacity* is the capacity of FIFO and LRU cache. The return value of these two functions are the

indicator of whether we miss the page in the cache, 1 for missing and 0 for not missing.

main.c would report the status of cache after assessing each page in the reference row and calculate the total number of missing pages. It would output the result in the <out_file> and also print on the command.

Two testcases are provided in *testcase* folder for you to validate your implementation. We show case1 for your better understandings of this problem.

```
3
7
1 2 3 1 4 1 3
```

The first row represents the capacity of FIFO cache and LRU cache. The second row represents the quantity of the pages that are required to be assessed in the reference row. The last row represents the page numbers in the reference row.

The output of the program involves the the status of cache after assessing each page in the reference row and the total number of missing pages, like the followings for case1:

```
After assessing 1, the elements in FIFO cache:1 the elements in LRU cache:1
After assessing 2, the elements in FIFO cache:2 1 the elements in LRU cache:2 1
After assessing 3, the elements in FIFO cache:3 2 1 the elements in LRU cache:3 2 1
After assessing 1, the elements in FIFO cache:3 2 1 the elements in LRU cache:1 3 2
After assessing 4, the elements in FIFO cache:4 3 2 the elements in LRU cache:4 1 3
After assessing 1, the elements in FIFO cache:1 4 3 the elements in LRU cache:1 4 3
After assessing 3, the elements in FIFO cache:1 4 3 the elements in LRU cache:3 1 4
Total times of missing in FIFO:5
Total times of missing in LRU:4
```

You are required to write down the outputs of program for case2 using your implemented code. Meanwhile, your implementation would be verified by some hidden testcases. You could refer to the tutorial slides to get some hints about how to implement this.

Bonus (20 marks, 1 problem)

This part is the bonus part. You would dive into the linux kernel to see how the segmentation and paging works.

In the tutorial, we explore an interesting example of terminating the indefinite loop in *linux-0.11* by address translation with *Bochs x86* Emulator . Based on this example, we would imitate the similar steps under the condition of multi processes. The updated code is as follows:

```
#include <stdio.h>
#include <unistd.h>
int i = 0x12345678;
int main(void)
{
	int pid = fork();
	if(pid == 0)
		printf("I am child process.The logical/virtical address of i is 0x%08x", &i);
	else
		printf("I am father process.The logical/virtical address of i is 0x%08x", &i);
	fflush(stdout);
	while(i);
	if(pid == 0)
		printf("The child process exits!");
	else
		printf("The father process exits!");
	return 0;
}
```

In this problem, you are required to **find the physical address of variable 'i' and set the value of 'i' to be zero in either father process or child process** with *Bochs x86* Emulator. A large difference from the tutorial is that you would find some interesting phenomena after introducing the **multi processes**.

Please answer the following questions (The code already exists with filename as '**homework.c**' and you could see it by 'ls' command in the *Bochs x86* Emulator shell. All the addresses should be written in the form of hexadecimal (like 0x00fa6000) in the following questions) :

a) After compiling 'homework.c' and executing 'homework', what are the outputs in the *Bochs x86* Emulator shell? Please explain about the outputs.

b) IA-32 uses two-level paging. When the *Bochs x86 Debugger* pauses at 'while(i)' at the first time, what is the logical address and physical of variable 'i'?
(hints: Use Ctrl+C and c to pause and continue the debugging until you firstly see 'cmp dword ptr' assembly language. Use *sreg*, *creg* and *xp* command to find the segment descriptors and starting address of page table)

c) After finding the physical address of variable 'i', using ***setpmem*** to set the value of 'i' to be zero and continuing the program, what are the phenomena you find in the *Bochs x86* Emulator shell? Please explain about this phenomena and give another possible outputs in the *Bochs x86* Emulator shell. **(Give a snapshot of the *Bochs x86* Emulator shell containing the phenomena)**

For this problem, you need to setup the *bochs x86* and *linux-0.11* enviroment using virtual machine. You could refer to '***Enviorment_Setup_Instructions.pdf***' in the attachment to establish the *bochs x86* + *linux-0.11* enviroment and know the usage of *bochs x86*. Also, we have a zoom recording about the similar case in the tutorial, you could see this to get some ideas. (Zoom link: https://cuhk.zoom.us/rec/share/YiSXwJu47YbBeE5K7aqwNRm0SzZ1RCo7oPHwwgY297Icq9GWOfUFxGMHNywxIhUh.ymrIjNPCrpbr8aYm?startTime=1679493977000

Password: NECKA2c?)

Submissions:

- **Plagiarism, incorrect file name, and lack of comments will result in various degrees of deductions in points.**
- The mandatory part is required to be completed for you. For the bonus part, you would get extra scores if you do it.
- In problem1, you need to submit your answers about the questions in pdf file named as '**Assignment_3.pdf**'
- In problem2, you need to submit your answers about the outputs of program for case2 in the same file '**Assignment_3.pdf**'. You also need to submit **fifo.c** and **lru.c** that can be compiled with the "Makefile" for problem2.
- In Bonus part, you need to submit your answers about the questions in the same file '**Assignment_3.pdf**' if you complete it.
- TA LUO, QIN is responsible for this assignment. Questions about the assignment via Piazza are welcomed and preferred. You could also contact him via email: qluo22@cse.cuhk.edu.hk. Requests like writing codes and debugging for you will be rejected according to the regulations.