CSCE 451/851 Programming Assignment 5

Designing a Virtual Memory Manager

1 Overview of Project

In this programming assignment you will be implementing a virtual memory manager. This PA actually comes from pages 458-461 in Silberschatz 9th edition[1]. It is replicated in Section 8 below for convenience. Additionally, Chapter 9 of that book is provided in the `PA5.zip` file if you have any questions.

Part 1 In Part 1 you will accomplish the assignment assuming infinite memory space. That is, there is no need for page replacement in Part 1.

Part 2 In Part 2, you will assume that physical memory can now only accommodate 128 frames. Now you will need a page replacement algorithm. You will need to implement both FIFO and LRU page replacement algorithms.

2 Program Specification

Command Line Parameters Your program should accept the following command line parameters:

- 1^{st} parameter: a file representing the "backing store" - a generic term for storage *not* in memory.
- 2^{nd} parameter: a file containing addresses you will access.

And Part 1 will be run as:

```
./part1 BACKING_STORE.bin addresses.txt
```

For Part 2, your program will need to accept an additional parameter:

- 3^{rd} parameter: page replacement algorithm indication, either `fifo` or `lru`.

Part 2 will be run as:

```
./part2 BACKING_STORE.bin <address file> <strategy>
```

where `<strategy>` can be `fifo` (First In First Out) or `lru` (Least Recently Used).

Output Your program for both parts should create a file `correct.txt` consisting of virtual and corresponding physical addresses, and the value at that address. These should be formatted **exactly** as indicated in the provided `correct.txt`. Additionally, at the end of the run your program should output the following (as seen in the provided `correct.txt` file):

- Number of translated addresses
- Page faults
- Page fault rate
- TLB hits
- TLB hit rate

[1] Silberschatz, Abraham, Greg Gagne, and Peter B. Galvin. *Operating system concepts: 9th Edition.* Wiley, 2013.

3 Submission

Do the following:

1. Create a directory called `<UNL username>_pa5` (replace `<UNL username>` with your UNL username).
2. Create subdirectories: `part1` and `part2` under `<UNL username>_pa5`.
3. Add a separate `Makefile` for both problems. The name of the executable produced by the first part and second part should be `part1` and `part2` respectively (this differs slightly from the text in Section 8 below which assumes the binary is named `a.out`). As usual if you don't provide a Makefile we can't/won't compile your code and you won't pass any tests.
4. You should have the following directory structure:

```
<UNL username>_pa5
|-----part1
|      |----Makefile
|      |----part1.c
|      |----<other C files as needed>
|-----part2
       |----Makefile
       |----part2.c
       |----<other C files as needed>
```

5. Zip the directory `<UNL username>_pa5` and submit.

Use web handin to hand in your assignment. Submit a single zip file, `<UNL username>_pa5.zip` (e.g., `jdoe2_pa5.zip`).

4 Evaluation

The output of your program should create a file, `correct.txt`, that matches the indicated "correct" file exactly. For grading, we will execute your program using the command shown in column 1 of the table and diff the output in your generated `correct.txt` against the provided output files as indicated in column 2 of the table. Points will be awarded in the following manner:

Command	File to Match	Points
`./part1 BACKING_STORE.bin addresses.txt`	`correct.txt`	60
Part 1 Total = 60 points		
`./part2 BACKING_STORE.bin addresses1.txt fifo`	`correct1_fifo.txt`	10
`./part2 BACKING_STORE.bin addresses1.txt lru`	`correct1_lru.txt`	10
`./part2 BACKING_STORE.bin addresses2.txt fifo`	`correct2_fifo.txt`	10
`./part2 BACKING_STORE.bin addresses2.txt lru`	`correct2_lru.txt`	10
Part 2 Total = 40 points		
Total: 100 points		

Additionally if the naming convention described in the Submission section above is not followed you lose 10 points. If your code does not compile on the CSE servers you will get 0 points. ***Please*** check that your code both compiles and runs on the CSE servers!!! Don't wait until it works completely on your own machine before testing on the CSE servers. Test on the CSE servers at regular milestones in your development.

5 Description of Provided Files

To help you, the following files and sample program can be found in this directory:

- `BACKING_STORE.bin`: file representing the backing store (i.e., storage not in memory)

- `part1.out`: binary for Part 1 against which you can test your output (only runs on CSE servers)
- `part2.out`: binary for Part 2 against which you can test your output (only runs on CSE servers)
- Files containing addresses you will access:
  - `addresses.txt`
  - `addresses1.txt`
  - `addresses2.txt`
- Files containing the output you must match exactly:
  - `correct.txt`
  - `correct1_fifo.txt`
  - `correct1_lru.txt`
  - `correct2_fifo.txt`
  - `correct2_lru.txt`
- `part1/template.c`: a (hopefully helpful pseudocode) template to get you started. You don't *have* to follow the template, but it may help you organize your code and clarify your thinking.

6 Part 1

*Part 1 consists of everything in Section 8 down to the last section called "**Modifications**."*

Notes for Part 1

1. Memory is large enough to accommodate all 256 pages from the `BACKING_STORE.bin`. That is, there is no need for page replacement in Part 1.
2. The solution uses FIFO replacement for TLB updating, so you should too!
3. To assign physical memory frames, start with frame 0 and assign them sequentially. This should make your addresses match those in `correct.txt`.

7 Part 2

*Part 2, described in "**Modifications**" in Section 8, consists of modifying your Part 1 program to implement FIFO and LRU for page-replacement in physical memory.*

Notes for Part 2

1. Assume that physical memory can now only accommodate 128 frames. Now you will need page replacement. You will implement both FIFO and LRU page-replacement strategies.

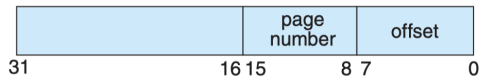
---

8 Designing a Virtual Memory Manager (Replicated from Textbook pages 458-461)

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. Your program will read from a file containing logical addresses and, using a TLB as well as 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. The goal behind this project is to simulate the steps involved in translating logical to physical addresses.

Specifics

Your program will read a file containing several 32-bit integer numbers that represent logical addresses. However, you need only be concerned with 16-bit addresses, so you must mask the rightmost 16 bits of each logical address. These 16 bits are divided into (1) an 8-bit page number and (2) 8-bit page offset. Hence, the addresses are structured as shown in the figure:



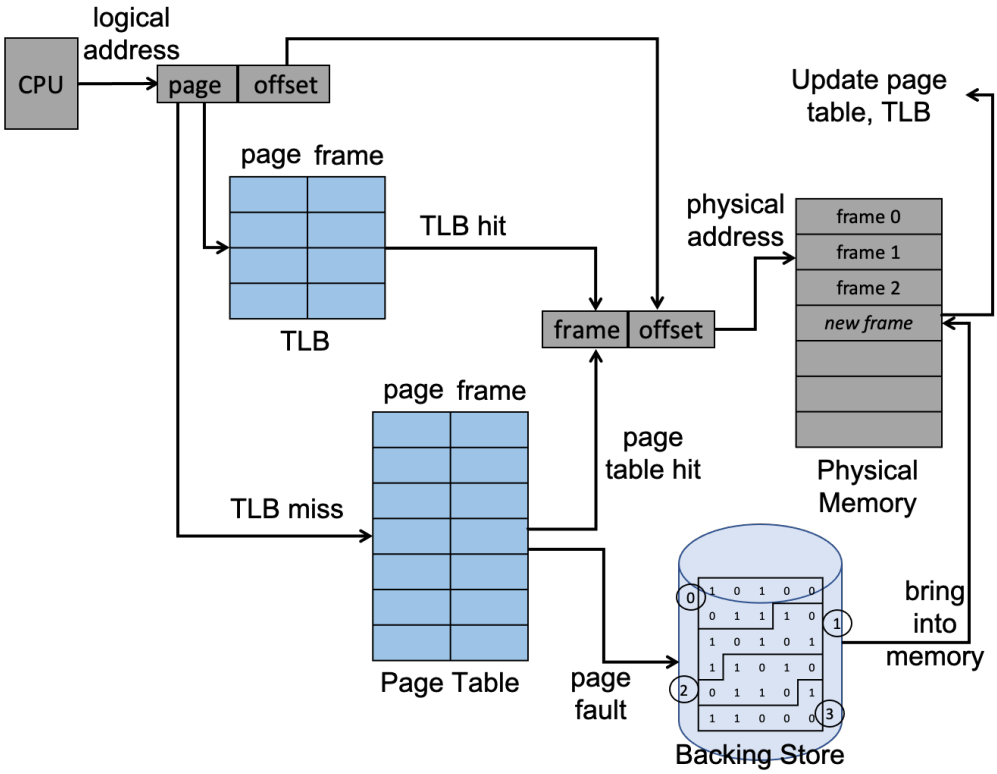
Other specifics include the following:

- 2^8 entries in the page table
- Page size of 2^8 bytes
- 16 entries in the TLB
- Frame size of 2^8 bytes
- 256 frames
- Physical memory of 65,536 bytes (256 frames $\times$ 256-byte frame size)

Additionally, your program need only be concerned with reading logical addresses and translating them to their corresponding physical addresses. You do not need to support writing to the logical address space.

Address Translation

Your program will translate logical to physical addresses using a TLB and page table as outlined in Section 8.5 of the textbook. 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 appears in the figure **(slightly improved from what's in the book)**:



Handling Page Faults

Your program will implement demand paging as described in Section 9.2 of the textbook. The backing store is represented by the file `BACKING_STORE.bin`, a binary file of size 65,536 bytes. When a page fault occurs, you 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.

You will need to treat `BACKING_STORE.bin` as a random-access file so that you can randomly seek to certain positions of the file for reading. We suggest using the standard C library functions for performing I/O, including `fopen()`, `fread()`, `fseek()`, and `fclose()`.

The size of physical memory is the same as the size of the virtual address space — 65,536 bytes — so you do not need to be concerned about page replacements during a page fault. Later, we describe a modification to this project using a smaller amount of physical memory; at that point, a page-replacement strategy will be required.

Test File

We provide the file `addresses.txt`, which contains integer values representing logical addresses ranging from 0-65535 (the size of the virtual address space). Your 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.

How to Begin

First, write a simple program that extracts the page number and offset (based on Figure) from the following integer numbers:

1, 256, 32768, 32769, 128, 65534, 33153

Perhaps the easiest way to do this is by using the operators for bit-masking and bit-shifting. Once you can correctly establish the page number and offset from an integer number, you are ready to begin.

Initially, we suggest that you bypass the TLB and use only a page table. You can integrate the TLB once your page table is working properly. Remember, address translation can work without a TLB; the TLB just makes it faster. When you are ready to implement the TLB, recall that it has only 16 entries, so you will need to use a replacement strategy when you update a full TLB. You may use either a FIFO or an LRU policy for updating your TLB.

How to Run Your Program

Your program should run as follows:

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

Your program will read in the file `addresses.txt`, which contains 1,000 logical addresses ranging from 0-65535. Your 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. (Recall that in the C language, the `char` data type occupies a byte of storage, so we suggest using `char` values.)

Your 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 at the translated physical address.

We also provide the file `correct.txt`, which contains the correct output values for the file `addresses.txt`. You should use this file to determine if your program is correctly translating logical to physical addresses.

Statistics

After completion, your 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.

Modifications

This project assumes 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. A suggested modification is to use a smaller physical address space. We recommend using 128 page frames rather than 256. This change will require modifying your program so that it keeps track of free page frames as well as implementing a page-replacement policy using either `FIFO` or `LRU` (Section 9.4 of the textbook).