<Spring 2015: Operating Systems>

Machine Problem 3:
Virtual Memory Manager
(DEADLINE: 6/10 23:59)

Disclaimer: This assignment is adopted from a project developed by Mangesh Yadav, Ajinkya, and Isan Sahoo at San Jose State University

- **Goals:**

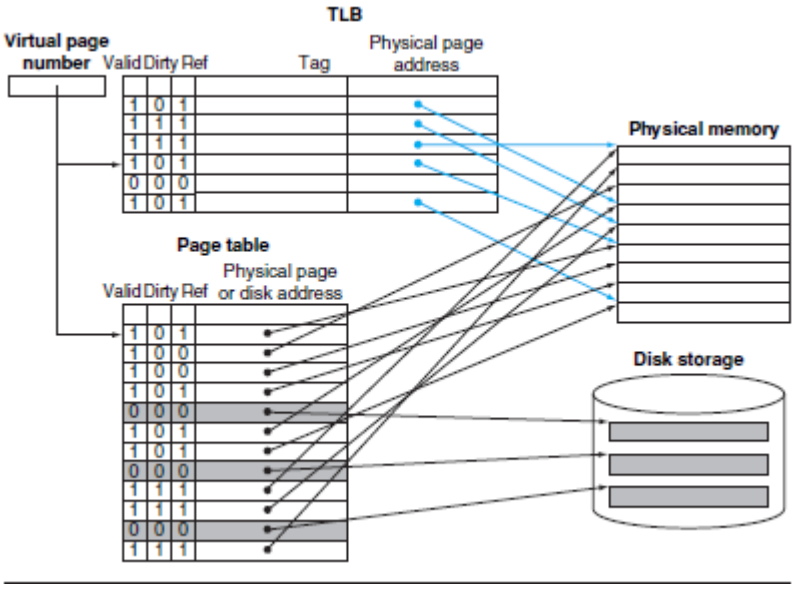
1. To dust off your C/C++ programming skills;
2. To understand virtual memory management in OS;
3. To implement the translation of logical addresses to physical addresses and replacement.

- **Introduction:**

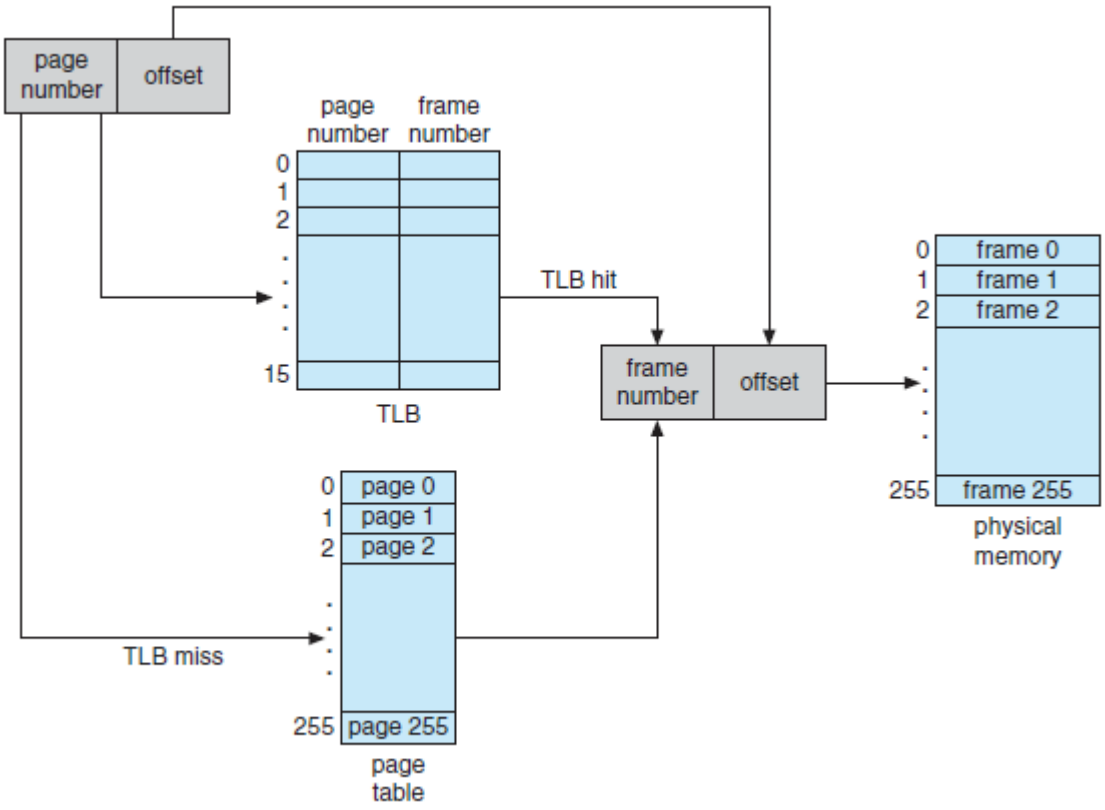
This MP consists of writing a program that translates logical to physical addresses for a virtual address space of size 2^{16} = 65,536 bytes. This program will read from a file containing logical addresses and, using a TLB as well as a page table, will translate each logical address to its corresponding physical address and output the value of the byte stored at the translated physical address. The goal behind this MP is to simulate the steps involved in translating logical to physical addresses and to get a better understanding of virtual memory management concepts
Size of Logical Address: 32 bit
Size of Physical Address: 16 bit
Frame size of 2^8 bytes
256 frames
Physical memory of 65,536 bytes (256 frames × 256-byte frame size)
Size of TLB: 16 entries in the TLB
Size of Page table: 2^8 entries in the page table
Page size of 2^8 bytes
Replacement scheme used: **FIFO/RANDOM (Requirement), LRU(Extra credit!!)**
The goal behind this MP is to simulate the steps involved in translating logical to physical addresses and to get a better understanding of virtual memory management concepts.
Additionally, we will also try to demonstrate the miss rate performance for following three replacement schemes in the TLB – FIFO, LRU, Random

- **Background:**

Virtual memory is a technique that allows the execution of processes that are not completely in memory. Only the part of the program that is needed is fetched fr The main memory can act as a cache for the secondary storage. One major advantage of this scheme is that programs can be larger than physical memory. Further, virtual memory abstracts main memory into an extremely large, uniform array of storage, separating logical memory as viewed by the user from physical memory. This technique frees programmers from the concerns of memory-storage limitations. Virtual memory also allows processes to share files easily and to implement shared memory. In addition, it provides an efficient mechanism for process creation.
A virtual memory block is called a ***page***, and the corresponding physical memory block it represents is called a frame. A virtual memory miss is called a **page fault**. With virtual memory, the processor produces a **virtual address**, which is translated by a combination of hardware and software to a *physical address*, which in turn can be used to access main memory.

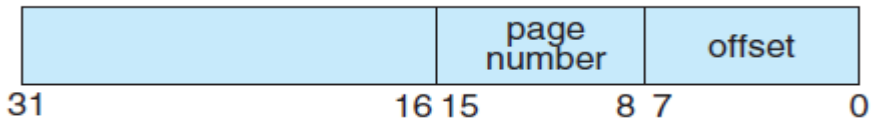


This is the process we are attempting to simulate though our code. Thus the schematic representation for our code would be like this -



- **Details:**

Our program will read a file containing several 32-bit integer numbers that represent logical addresses. However, we need only be concerned with 16-bit addresses, so we 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 –



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

1. Address Translation

Our program will translate logical to physical addresses using a TLB and page table. 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.

2. Handling Page Faults

Your program will implement demand paging. The backing store is represented by the file **mp3/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.
We will need to treat **mp3/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 MP using a smaller amount of physical memory; at that point, a page-replacement strategy will be required.

3. Test File

We provide the file **mp3/addresses.txt**, which contains integer values representing logical addresses ranging from 0 − 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.

4. How to Run Your Program

Our program will read in the file **mp3/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. (In the C language, the char data type occupies a byte of storage, so we suggest using char values.)

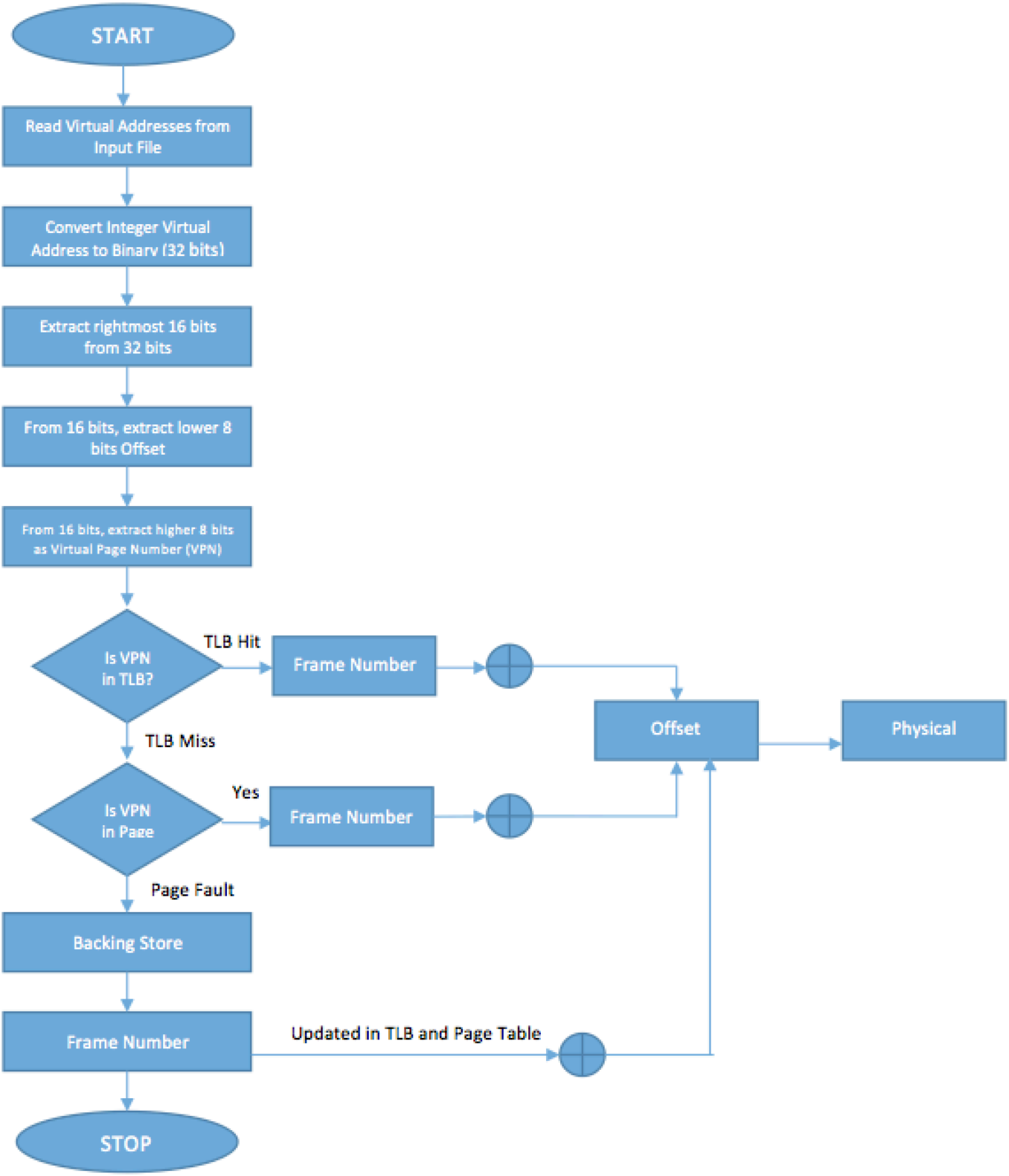
Our program is to output the following values:
1) The logical address being translated (the integer value being read from **mp3/addresses.txt**).
2) The corresponding physical address (what our program translates the logical address to).

3) The signed byte value stored at the translated physical address. (This will be a char)
We also provide the file **mp3/correct.txt**, which contains the correct output values for the file **mp3/addresses.txt**. You should use this file to determine if our program is correctly translating logical to physical addresses.

5. 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 **mp3/addresses.txt** were generated randomly and do not reflect any memory access locality, do not expect to have a high TLB hit rate.

- **Psudo-Code/ Algorithms/ Flow Chart:**



Submission Instructions

1. After compiling, we should see one executable (e.g., shell). **Attach a README file** describing the name of the executable, special compiling instructions, or anything else special you want to let us know. (specify associated source codes if applicable) The README file should be in **plain text** format
2. **A report in PDF format**. The report should briefly describe your implemented solution and provide all screenshots of result after executing each requirement described here. It need to be written in **ENGLISH**. Write down **0) your working environment and assignment details to each team member**. **1)implementation details about each requirement, 2)the screenshots of the results after executing them, and 3) summary of how the operating system supports the virtual memory management based on your implementation**.
3. Your source codes, README, and report compressed to a single **yourname_student#.tar.gz** file. The code should be well commented and it should be easy to see the correspondence between what's in the code and what's in the report. **1) Create .c or .cpp source files for the shell implementation, and add additional relevant files if needed** and **2) Make a directory whose name should be exactly your student#** and **put all source files and relevant files into the directory**