CIS 3110: Operating Systems

Assignment 3: Virtual Memory Manager

Due Tuesday, March 17, 2020 @ 11:59pm

1. Objective

The objectives of this assignment is to familiarize you with Virtual Memory and become familiar with OS handling of TLB and page faults. Specifically, you are required to design a simple virtual memory manager[1].

2. Review questions

Before you proceed to perform the experiment described in the lab, you need to solve the following theoretical questions, which will be very helpful to you solving this programming assignment. Write up your answers to the following questions and save them in a file called answers-a3.txt.

Chapter 8, Textbook: 8.20 (page 392)
Q1) Assuming a 1-KB page size, what are the page numbers and offsets for the following address references (provided as decimal numbers):
a. 3085
b. 42095
c. 215201
d. 650000
e. 2000001

Chapter 8, Textbook: 8.23 (page 392)
Q2) Consider a logical address space of 256 pages with a 4-KB page size, mapped onto a physical memory of 64 frames.
a. How many bits are required in the logical address?
b. How many bits are required in the physical address?

Chapter 9, Textbook: 9.8 (page 451)
Q3) Consider the following page reference string:
1, 2, 3, 4, 2, 1, 5, 6, 2, 1, 2, 3, 7, 6, 3, 2, 1, 2, 3, 6.
How many page faults would occur for the following replacement algorithms, assuming one, two, three, four, five, six, and seven frames? Remember that all frames are initially empty, so your first unique pages will cost one fault each.

- LRU replacement

[1]Programming Projects - Designing a Virtual Memory Manager on page 458, Chapter 9, Operating System Concepts, 9th edition

- FIFO replacement
- Optimal replacement

3. Designing a Virtual Memory Manager

In this part, you are required to write 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 it is to simulate the steps involved in translating logical to physical addresses.

Specifics

Your program will take one argument from command line, which is a file name. Your program reads the 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 Figure 1.
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 × 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.

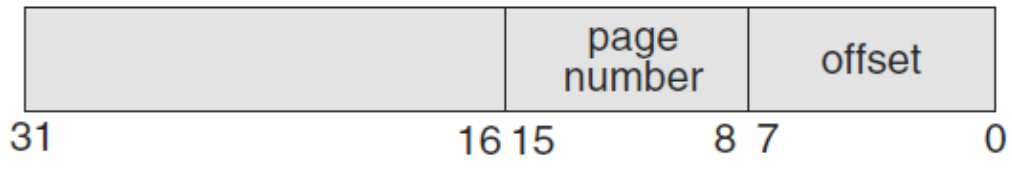

Figure 1 Address structure.

Address Translation

Your program will translate logical to physical addresses using a TLB and page table as outlined in Section 8.5. 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 Figure 2.

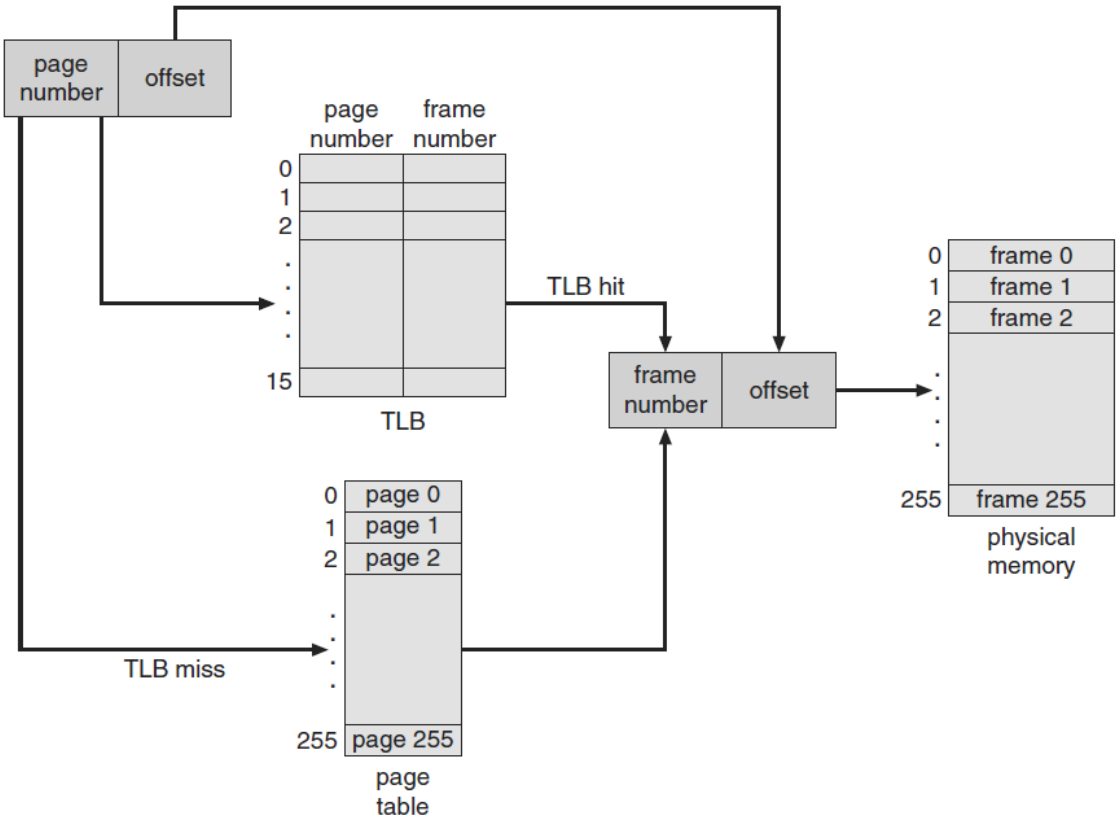

Figure 2 A representation of the address-translation process.

Handling Page Faults

Your program will implement demand paging as described in Section 9.2. The backing store is represented by the file BACKING_STORE.bin, a binary file of size 65,536 bytes. When a page fault occurs, you will read in a 256-byte page from the file BACKING_STORE.bin 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.

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 1) 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 a FIFO policy for updating your TLB**.

How to Run Your Program

Assume that the resulted executable file after compiling your virtual memory manager program is called virmem. Your program should run as follows:

./virmem addresses.txt

Your program will read in the file addresses.txt, which contains 1,000 logical addresses ranging from 0 to 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 the **standard output device (stdout)**:

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.

The output format is shown below

Virtual address: <logical address> Physical address: <physical address> Value: <byte value>
where output fields are separated by a single whitespace.
For example,

Virtual address: 31260 Physical address: 23324 Value: 0

which means that the corresponding physical address for the logical address 31260 is 23324. The value stored at physical memory unit of an address 23324 is 0.

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 to the **standard error output device (stderr)**:

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.

The output format is shown below

Page Fault Rate=< page-fault rate>

TLB Hit Rate=<TLB-hit rate>

Please be advised that your results will be accurate to **three decimal places**.

For example,

Page Fault Rate=0.331

TLB Hit Rate=0.058

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.

Robust programming with error handling

Your program should be robust for bad arguments! If the argument is invalid or bad, your program should print a **usage statement**, and then exit graceful indicating an unsuccessful termination[4]. Further, your program should be robust to handle abnormal situations, for example, reading non-existing files.

There are several parts you must pay attention to about **usage statements** [2]

- The usage message: it always starts with the word "usage", followed by the program name and the names of the arguments. Argument names should be descriptive if possible, telling what the arguments refer to, like "filenameoflogicaladdresses" in the example below. Argument names should not contain spaces! Optional arguments are put between square brackets, like "-l" for the Linux command *ls*. Do not use square brackets for non-optional arguments! Always print to stderr, not to stdout, to indicate that the program has been invoked incorrectly.
- The program name: always use argv[0] to refer to the program name rather than writing it out explicitly. This means that if you rename the program (which is common) you won't have to re-write the code.
- Exiting the program: use the exit function, which is defined in the header file <stdlib.h>. Any non-zero argument to exit (e.g. exit(1)) signals an unsuccessful completion of the program (a zero argument to exit (exit(0)) indicates successful completion of the program, but you rarely need to use exit for this). Or, you can simply use EXIT_FAILURE and EXIT_SUCCESS (which are defined in <stdlib.h>) instead of 1 and 0 as arguments to exit.

For example, the following is a code snippet, which prints a usage statement, and then exits the program by indicating an unsuccessful termination

```
fprintf(stderr, "usage: %s filenameoflogicaladdresses\n", argv[0]);
exit(EXIT_FAILURE);
```

Grading Scheme

Program compiles, runs	2
Code quality (e.g., CODE Style & Documentation)	3
Review questions	30
Address translation testing	40
Statistics correctness testing	16
Stability and reliability testing (e.g., not have memory leaks or memory violations, proper error handling)	9

Submission

- If you have any problems in the development of your programs, contact the teaching assistant (TA) assigned to this course.
- You are encouraged to discuss this project with fellow students. However, you are **not** allowed to share code and your answers to review questions with any student.
- If your TA is unable to run/test your program, you should present a demo arranged by your TA's request.
- Please only submit the source code files plus any files required to build your Virtual Memory Manager as well as any files requested in the assignment, including
  - the complete source code,
  - the Makefile; and
  - answers-a3.txt

Note that when making your Makefile file, the resulted executable file after compiling your virtual memory manager program must be called **virmem**.

How to name your programming assignment projects: For any assignment, zip all the files required to a zip file with name: CIS3110_<assignment_number>_XXX.zip, where <assignment_number> is the assisgnement number you are solving (e.g., a3 for Programming Assignment 3) and XXX is your University of Guelph's email ID (Central Login ID). This naming convention facilitates the tasks of marking for the instructor and course TAs. It also helps you in organizing your course work. Failure to follow the requirements will result in mark reduction.

Note: to zip and unzip files in Unix:

```
$zip -r filename.zip files
$unzip filename.zip
```

Please only submit the .c source code files and Makefile file plus any files requested in an assignment. You are required to develop software in C on VM provided by the textbook.

References:
[1] Operating System Concepts, Silbershatz, Gagne and Galvin, 9th Ed.
[2] Processing command-line arguments.
http://courses.cms.caltech.edu/cs11/material/c/mike/misc/cmdline_args.html
[3] CS 11: How to write usage statements.
http://courses.cms.caltech.edu/cs11/material/general/usage.html
[4] Exit Status.
https://www.gnu.org/software/libc/manual/html_node/Exit-Status.html