COMP 3500 Introduction to Operating Systems
Project 6 – Virtual Memory Manager

Points Possible: 100
Submission via Canvas
This is an individual assignment; no collaboration among students. Students should NOT share any project code with any other student. Collaborations among students in any form will be treated as a serious violation of the University's academic integrity code.

Objectives:

- To design a simple virtual memory system
- To Implement a C or C++ program that services TLB faults.
- To tune your TLB and memory replacement algorithms.
- Use GDB to debug your C or C++ program

1. Overview

This project aims at providing you with an opportunity to design and implement a standalone virtual memory manager, where there is a software-managed TLB. It is your responsibility to implement the code to manage TLB of your virtual memory system. Your program is responsible to (1) load a file containing a list of logical addresses, (2) translate logical addresses into physical addresses for a virtual address space of size 2^{16} = 65,536 bytes, and (3) output the value of the byte stored at the translated physical address.

2. Information about the Simulated Virtual Memory

2.1 16-bit Logical Addresses

Important! Your program must read a file containing a list of 32-bit integer numbers, which represent 32-bit logical addresses. Please note that your program only deals with 16-bit addresses. Thus, you need to mask the rightmost 16 bits of each logical address loaded from the file.

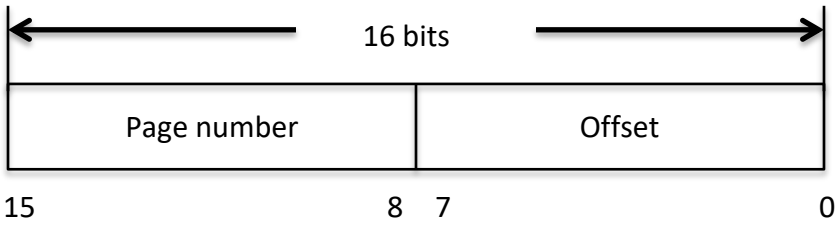

Figure 1. 16-bit Logical Address Structure

Each 16-bit logical address is divided into two parts, namely, (1) an 8-bit page number

and (2) an 8-bit page offset. The logical address structure is illustrated in Figure 1.

2.2 System Parameters of the Virtual Memory

The page table size is 2^8; the TLB contains 16 entries. The page size is 2^8 bytes, which is the same as the frame size. There are a total of 256 frames in the physical memory, meaning that the total physical memory capability is 65,536 bytes (i.e., 256 frames * 256 bytes/frame). The system parameters of the simulated virtual memory is summarized below.

- Page table size: 2^8
- Number of TLB entries: 16
- Page size: 2^8 bytes
- Frame size: 2^8 bytes
- Number of frames: 256
- Physical memory size: 65,536 bytes

2.3 Your mission is to:

- Read a file containing a list of logical addresses
- Translate logical addresses into physical addresses
- Output the value of the byte stored at the translated physical address.

3. Address Translation

3.1 Paging

The operating system provides a virtual memory system in a way to use physical memory as a cache of virtual pages. As we mentioned in our lecture, with paging in place, all the pages in a process's virtual address space are NOT required to be in physical memory. Rather, a process can have its pages either on a disk or in memory. A **page fault** occurs in case that the process issues an access to a page on the disk. The operating system has to retrieve the page from the disk and bring it into the memory. Note that pages with valid TLB entries reside in physical memory. Thus, a reference to a page on disk results in a TLB fault. When a TLB fault occurs, the hardware generates a TLB exception, trapping to the operating system. The operating system then checks its own page table to locate the virtual page requested. If the page is in memory without being mapped by the TLB, then we merely need to update the TLB. However, if the page is on disk, the operating system must read the page from the disk, update the page table, and update the TLB. Please refer to Section 4 on page 3 on how to handle page faults.

3.2 Performing Address Translations

Your simulated virtual memory system is expected to translate logical addresses into physical addresses using TLB and a page table. Given a logical address, your program takes the following three steps to perform address translation.

- **Step 1:** The page number is extracted from the logical address.
- **Step 2:** Access the TLB using the extracted page number.
  - If there is a TLB-hit, the frame number of the page is obtained from the TLB.
  - Otherwise, follow **step 3** to access the page table.
- **Step 3:** Access the page table
  - The frame number of the page is obtained from the page table if the page has been loaded into the main memory;
  - Otherwise, a page fault occurs.

Figure 2 below depicts the address-translation process, where the TLB and page table are two key data structures.

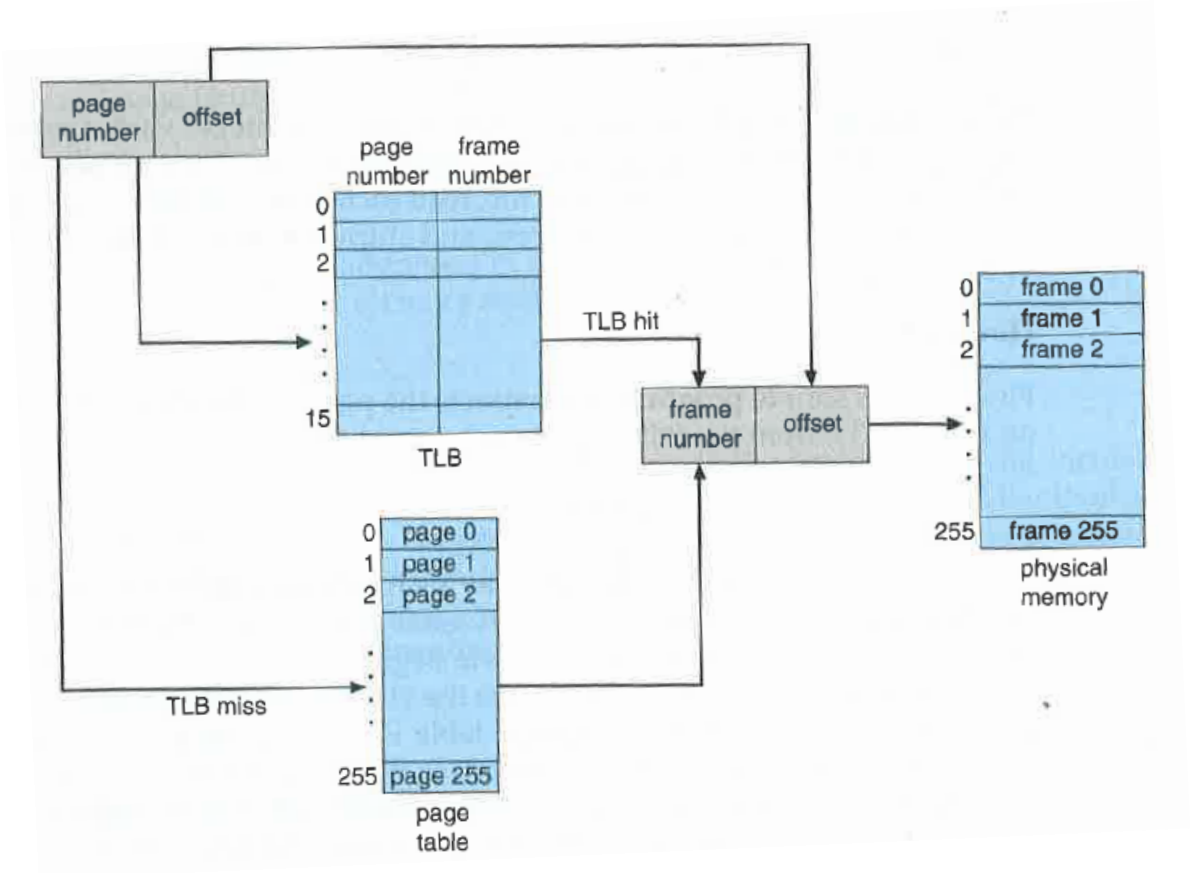

Figure 2. The address-translation process.

4. Handling Page Faults

4.1. A Question about TLB and Page Fault

You do not need to submit your answers to the following question, but you must make

an effort to answer it. The question is designed to direct your attention to the design of your simulated virtual memory system. This question aims to improve your understanding of virtual memory.

Assuming that a user program just tried to access a piece of data at (virtual) address X, describe the conditions under which each of the following can arise. If the situation cannot happen, answer "impossible" and explain why it cannot occur.

- TLB miss, page fault
- TLB miss, no page fault
- TLB hit, page fault
- TLB hit, no page fault

4.2. How to handle a page fault?

You must implement demand paging in the simulated virtual memory system. The backing store is simulated by a file called "`BACKING_STORE`", which is available on the Canvas system. `BACKING_STORE` is a binary file of 65,536 bytes. When a page fault occurs, your virtual memory system will perform the following four steps:

- **Step 1:** read a 256-byte page from the file `BACKING_STORE` and
- **Step 2:** store the loaded page frame in the physical memory.
- **Step 3:** Update the page table
- **Step 4:** Update the TLB

An Example For example, if a logical address with page number 15 triggers a page fault, your virtual memory system will read in page 15 from the file `BACKING_STORE`. Then, the loaded page frame is placed in the physical memory. After the page frame is fetched from the disk, the page table and the TLB will be updated accordingly. Subsequent access of page 15 will be referenced by accessing either the TLB or the page table.

4.3. Accessing `BACKING_STORE`

One of your group members needs to implement functions to move a page from the simulated disk to physical memory. Note that `BACKING_STORE` is a simulated hard drive in your system.

It is worth mentioning that the file `BACKING_STORE` is accessed as a random-access file, where your virtual memory system can randomly seek to certain positions to the file for reading pages from the simulated disk. You are suggested to use the standard C library functions for performing file reads, including fopen(), fread(), fseek(), and fclose().

Important! In one of our lectures related to this project, we will show you how to use the standard C library functions to access `BACKING_STORE`.

4.4. No Page Replacement

Your system does not need to deal with the page replacement issue, because the physical memory size equals to the virtual address space. In other words, during the page fault process, no page will be invicted from the physical memory.

5. How to emark on this project?

5.1. First Step

The first step is understanding how TLB and page faults occur. You should expect to spend a substantial amount of time reading and learning the concepts of paging and TLB. Therefore, you must study TLB faults and page faults in details.

5.2. Address Key Design Issues

You should address the following key issues before moving on.

- What will your page tables look like?
- What should you put in each PTE (page table entry)?
- In what order can TLB faults and page faults occur? For example, can a page fault occur without causing a TLB fault?

When you write test programs, think about verifying that your TLB replacement algorithms are working correctly (i.e., "If I run this program it should generate exactly n TLB faults with your algorithm; does it?").

Review each other's designs. Think about how you might have implemented the different parts and compare ideas. As your system begins to stabilize, begin to work on the performance tuning. Continue to test and fix bugs. Take turns testing and tuning. Be sure to keep a careful log of your performance experiments so you can trade it back and forth.

5.3. Page Fault vs. TLB Fault Handling

The time required to implement page fault handling is longer than the time required to implement TLB fault handling. You should divide the virtual memory implementation into several small and well-defined modules so that you can both work on it as soon as one of you has completed the TLB implementation.

When you have completed your initial implementation of the TLB and virtual memory, you can partition the tasks as follows:

- **Task 1:** work performance tuning. Please refer to Section 7.2 on page 7 for details on performance evaluation.

- **Task 2:** Write a performance evaluation report.
- **Task 3:** Work on separate compilation and prepare a makefile. Please refer to Section 8 on page 7.

6. Implementation Details

6.1. Extracting Page Numbers and Offset

Write a function is to extract page number and an offset from the following addresses:

```
1, 256, 32768, 23769, 128, 65534, 33153
```

A straightforward way of extracting page number and offset is to use bit-masking and bit-shifting operators. After you fully implement and test this function, you can integrate this function into your simulated virtual memory system.

6.2. Implementing the Page Table

Your system must handle page faults using the TLB and the page table. Before implement the TLB, you should focus on the implementation of a page table. Please make sure your system has a functional page table prior to the development of the TLB. This strategy is feasible, because your system can perform without a TLB, the goal of which is to improve the performance of a virtual memory system.

6.3. Implementing the TLB along with a Replacement Strategy

It is strongly suggested that you implement the TLB and paging module separately and then integrate them together. Assuming that you have successfully implemented the page table in your system, now we can start the TLB implementation. Recall that the TLB has a total of 16 entries, which means you must implement a replacement strategy when your system is about to update a full TLB. The easiest strategy to handle TLB faults is FIFO (i.e., First In and First Out). You must also implement the LRU (i.e., Least Recently Used) strategy to optimize the TLB hit rate. Your system must guarantee that the TLB state is initialized properly.

6.4. What you don't have to implement?

Please keep in mind that this project will not be build on top of OS/161. You don't have to implement a handful of components of an operating system. In particular, you can ignore the following items while implementing your system.

- You do NOT have to implement a replacement strategy for the page table.
- You do NOT need to take care of context switch, because there is no process manager in your system.
- You do NOT have to worry about the kernel.
- You do NOT have to read OS/161 source code.

7. Testing and Performance Evaluation

7.1. Testing File

A testing file named `InputFile.txt` will be provided to you to test your simulated virtual memory system. This testing file is available on Canvas system. The file contains a list of integers representing logical addresses ranging from 0 to 65,535, which is the virtual address space. You program must (1) read all the logical addresses stored in this file, (2) translate the addresses into corresponding physical addresses, and (3) output the value of the singed byte at the physical address.

A small test input file called `testinput.txt` can be downloaded from Canvas to test your implementation.

7.2. Performance Tuning

Now we pay our attention to the issue of performance evaluation for your virtual memory system. Please use the file `vm_sim_output.txt` as your "lab notebook" for this section. You will undoubtedly want to implement some additional software counters. As a start, we suggest:

- The number of page faults.
- The number of TLB faults.

You should add the necessary infrastructure to maintain these statistics as well as any other statistics that may be of help in the process of tuning your system. Once the page handling functions and added instrumentation have been completed, you position yourself to tune your virtual memory system.

Experiment with different TLB replacement strategies (i.e., FIFO and LRU) and system parameters in an attempt to improve your performance. Please predict what will happen as you change TLB replacement algorithms and parameters. In doing so, you can compare the measured to the predicted results. A different result from what you expect might indicate a bug in your understanding, a bug in your implementation, or a bug in your testing. In this case, you need to figure out where the error is.

You should add other test programs to your collection as you evaluate the performance. The purpose of doing so is to demonstratively show that your system perform well for various system parameters.

Provide a complete summary of the performance analysis you conduct. The summary should include a thorough explanation of the performance benefits of your optimizations.

8. Programming Requirements

8.1. Programming Environment

Write your simulated virtual memory system in either C or C++. Compile and run your system using the `gcc` or `g++` compiler on a Linux box (either in Shop 3, computer labs in Shelby, your home Linux machine, a Linux box on a virtual machine, or using an emulator like Cygwin).

8.2. Function-Oriented Approach

You are *strongly suggested* to use an structure-oriented (a.k.a., function-oriented) approach for this project. In other words, you will need to write function definitions and use those functions; you can't just throw everything in the `main()` function. A well-done implementation will produce a number of robust functions, many of which may be useful for future programs in this project and beyond.

Remember good design practices include:

- A function should do one thing, and do it well
- Functions should NOT be highly coupled

8.3. File Names

You will lose points if you do not use the specific program file name, or do not have a comment block on **EVERY** program you hand in.

8.4 Usability Concerns and Error-Checking

You should appropriately prompt your user and assume that they only have basic knowledge of the system. You should provide enough error-checking that a moderately informed user will not crash your system. This should be discovered through your unit-testing. Your prompts should still inform the user of what is expected of them, even if you have error-checking in place.

9. Separate Compilation:

You must use separate compilation and create a makefile for this project.

What is Make?

Make is a program that looks for a file called "makefile" or "Makefile", within the makefile are variables and things called dependencies. There are many things you can do with makefiles, if all you've ever done with makefiles is compile C or C++ then you are missing out. Pretty much anything that needs to be compiled (postscript, java, Fortran), can utilize makefiles.

Format of Makefiles -- Variables

First, lets talk about the simpler of the two ideas in makefiles, variables. Variable definitions are in the following format:

```
VARNAME = Value
```

So let's say I want to use a variable to set what compiler I'm going to use. This is helpful b/c you may want to switch from cc to gcc or to g++. We would have the following line in our makefile

```
CC = gcc
```

This assigns the variable CC to the string "gcc". To expand variables, use the following form:

```
${VARNAME}
```

So to expand our CC variable we would say:

```
${CC}
```

Format of Makefiles -- Dependencies

Dependencies are the heart of makefiles. Without them nothing would work. Dependencies have the following form:

```
dependecy1: dependencyA dependencyB ... dependencyN
    command for dependency1
```

Check out the following links for more information on makefiles:
http://oucsace.cs.ohiou.edu/~bhumphre/makefile.html

10. Sample Usage and Output File

10.1. Sample Usage

Your simulated virtual memory system should run as follows:

```
./vm_sim InputFile.txt
```

Your program should provide the following user interface:

```
Welcome to Your-Group-Name's VM Simulator Version 1.0

Number of logical pages: 256
Page size: 256 bytes
Page table size: 256
TLB size: 16 entries
Number of physical frames: 256
Physical memory size: 65,536 bytes

Display Physical Addresses? [yes or no] Yes
Choose TLB Replacement Strategy [1: FIFO, 2: LRU] 1
Virtual address: 28720; Physical address: 48; Value: 0
Virtual address: 24976; Physical address: 400; Value: 0
Virtual address: 24112; Physical address: 560; Value: 0
Virtual address: 49454; Physical address: 814; Value: 48
……

Page fault rate:  48%
TLB hit rate: 7.5%

Check the results in the outputfile: vm_sim_output.txt
```

10.2. Output File Format

The format of your output file vm_sim_output.txt is given below:

```
Virtual address: 28720; Physical address: 48; Value: 0
Virtual address: 24976; Physical address: 400; Value: 0
Virtual address: 24112; Physical address: 560; Value: 0
Virtual address: 49454; Physical address: 814; Value: 48
……

Page fault rate:  48%
TLB hit rate: 7.5%
```

11. Project Report

Write a project report that explains how you design and implement your virtual memory manager. Your report must address all of the questions. Your report is worth 20 points.

Important! Your project report should provide sample input and output from your virtual memory manager.

11.1 (10 points) Solution Description

You must describe your solutions with good sample input and output to show users what your memory manager is doing.

(1) How did you guarantee that each logical address is translated to the correct physical address?
(2) How did implement the page table, physical memory, and TLB?
(3) Does your program realistically and accurately simulate a virtual memory system?
(4) Did you use the Java operators for bit-masking and bit-shifting?
(5) When a TLB miss occurs, how do you decide which entry to replace?

11.2 (5 points) Generality and Performance Criteria

(1) How general is your solution?
(2) How easy would it be to change parameters such as the size of the TLB?
(3) Does your program only load pages from the backing store when they are needed?
(4) Does your solution allow the physical address space to be smaller than the virtual address space?

11.3 (5 points) Miscellaneous factors

(1) Is your code elegant?
(2) How innovative is your solution? Did you try any ideas not suggested here (e.g. a choice of replacement policies for the TLB)? Innovative ideas that go beyond the requirements could receive extra credit.
(3) Did you document all outside sources?

12. Deliverables

12.1 Final Submission

Your final submission should include:

- Your project report (see also Section 11).
- A copy of the complete source code of your virtual memory manager.

Important! You must submit a single compressed file (see Section 12.2) as a .tar.gz file, which includes both a project report and source code.

12.2 A Single Compressed File

Now, submit your tarred and compressed file named `proejct6.tgz` through Canvas. You must submit your single compressed file through Canvas (no e-mail submission is accepted.

12.3 What happens if you can't complete the project?

If you are unable to complete this project for some reasons, please describe in your final design document the work that remains to be finished. It is important to present an honest assessment of any incomplete components.

13. Grading Criteria

The approximate marks allocation will be:

1) Project Report: 20%
2) Implementation (i.e., Source Code): 70%
3) Clarity and attention to details: 10%

14. Late Submission Penalty

- Ten percent (10%) penalty per day for late submission. For example, an assignment submitted after the deadline but up to 1 day (24 hours) late can achieve a maximum of 90% of points allocated for the assignment. An assignment submitted after the deadline but up to 2 days (48 hours) late can achieve a maximum of 80% of points allocated for the assignment.
- Assignment submitted more than 3 days (72 hours) after the deadline will not be graded.

15. Rebuttal period

- You will be given a period a week (7 days) to read and respond to the comments and grades of your homework or project assignment. The TA may use this opportunity to address any concern and question you have. The TA also may ask for additional information from you regarding your homework or project.