CPSC 351, Operating Systems Concepts

Final Programming Project

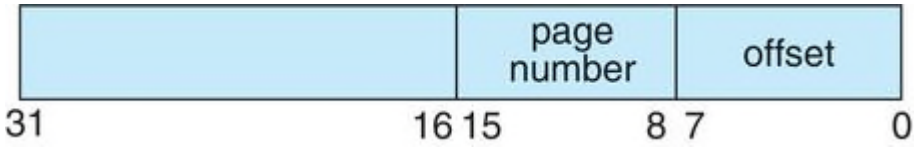
This project is part of your final exam and is to be done in teams of 3 students. Teams must be pre-established and coordinated with your instructor. One project shall be submitted through TITANium for the entire team. See TITANium for due dates. The project as described herein is inspired by Chapter 10's Programming Project, but is not identical. Several subtle changes have been made throughout, so pay particular attention to the problem definition describe herein.

Designing a Virtual Memory Manager

You are to design, write, compile, execute, and submit a well-formed C++[1] virtual memory manager program. 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 shall read from standard input[2] containing logical addresses and, using a translation lookaside buffer (TLB) and a page table (PT), shall translate each logical address to its corresponding physical address and output the value of the byte stored at the translated physical address. Your learning goal is to use simulation to understand the steps involved in translating logical to physical addresses. This shall include resolving page faults using demand paging, managing a TLB, and implementing two page-replacement algorithms.

Specifics

Your program shall read from standard input 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) an 8-bit page offset. Hence, the addresses are structured as shown as:



Other specifics include the following[3]:

- 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.

---

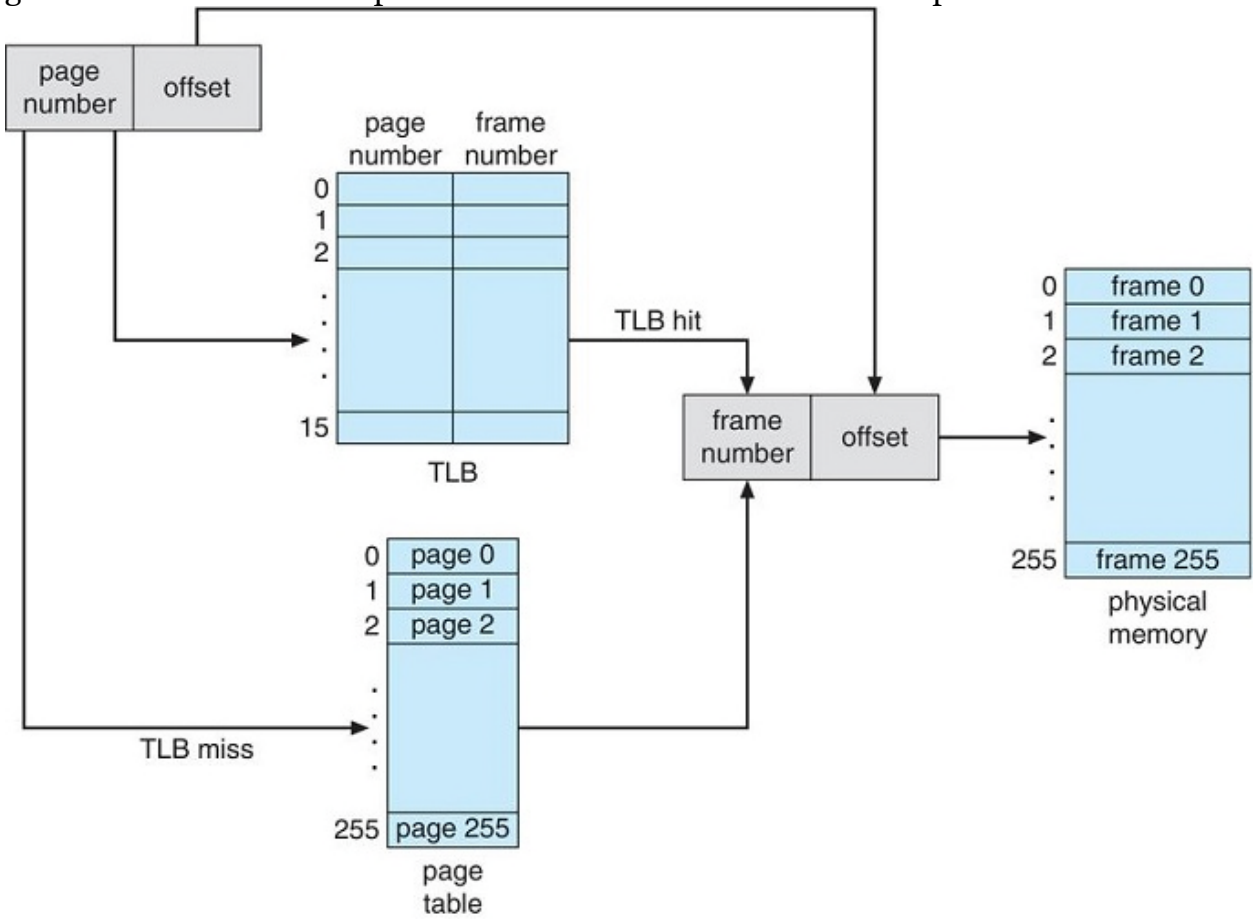
[1] This program must be done in C++, not C. Well-formed means it shall be Object Oriented with clear separation of domain objects, each codified in separate C++ classes. Interface and Implementation shall be separated into header files (.hpp, not .h) and source files (.cpp). Standard practice of having all function definitions (bodies) reside in source files shall be observed - no function definitions shall reside in header files. Each class shall be implemented its own pair of header/source files. It is expected you will have many header/source file pairs. If two classes are very, very closely related (a rare event), then they may reside in the same header/source file pair. Well-formed also means it complies with no errors and no warnings, as well as executes without abnormal termination. Good coding practices and well commented code are also expected. This is meant to be a formal delivery representative of a 300-level computer science college course and will be evaluated as such.

[2] Read the address.txt file from standard input (sometimes called the console input, or simply "cin"). See https://www.tutorialspoint.com/unix/unix-io-redirections.htm if you need a redirection refresher.

[3] The final version of your program submitted for credit shall implement the Page Replacement section described towards the end of this description. This will alter some of the values in this section. It's recommended that you get your program working without page replacement first, then add page replacement to your working system.

Address Translation

Your program shall translate logical to physical addresses using a TLB and page table as outlined in Section 9.3. 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 is:



Handling Page Faults

Your program shall implement demand paging as described in Section 10.2. 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 shall read in a 256-byte frame from the file `BACKING_STORE` and store it in an available 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 shall be resolved by either the TLB or the page table. You shall treat `BACKING_STORE.bin` as a random-access file[4] so that you can randomly seek to certain positions of the file for reading. Use the standard C++ library functions[5] for performing I/O, including `std::fstream::open()`, `std::fstream::read()`, and `std::fstream::seekg()`.

Initially, the size of physical memory is the same as the size of the virtual address space—65,536 bytes—so you do not (yet) need to be concerned about page replacements during a page fault. Later,

[4] Good Object Oriented Design and Programming suggests the BackingStore should be its own class, and this behavior encapsulated within. The BACKING_STORE.bin file should be opened by the BackingStore constructor and closed in the BackingStore destructor.

[5] Use only C++ headers and functions. Do not use C headers and functions.

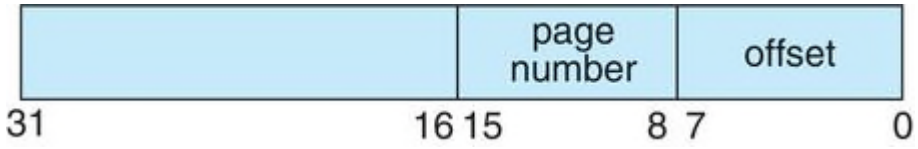
we describe a modification to this project using a smaller amount of physical memory; at that point, a page-replacement strategy shall be required[3].

Test File

We provide the file `addresses.txt`, which contains integer values representing logical addresses ranging from 0 *to* 65535 (the size of the virtual address space). You shall redirect `addresses.txt` to standard input at the command line when running your program. Your program shall read from standard input each logical address and translate it to its corresponding physical address, and output the value of the unsigned byte in hexadecimal at the physical address.

How to Begin

1. Read this problem statement again looking for major concepts and domain objects. Sketch out what you think your classes will be in a UML Class Diagram.
2. Create header files for your classes and populate. Classes should contain member and non-member function prototypes, instance attributes, and good comments. Do not yet implement functions. Header files convey to the user (client) of the class what the object is, what operations can be performed, and any limitations and side effects. So your comments should too.
3. Create main.cpp and function `main`, include your header files and get that much to compile clean.
4. Create source files for your header files and populate with function definition stubs. Verify each source file compiles clean.
5. Populate function `main` and related functions by writing a simple program that extracts the page number and offset based on:



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 (binary operators &, |, and ^) and bit-shifting (operators << and >>). Once you can correctly establish the page number and offset from an integer number, you are ready to continue.

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 sixteen entries, so you shall need to use a replacement strategy when you update a full TLB. You shall implement both a FIFO and an LRU policy for updating your TLB, but only one shall be used at a time selectable either at build time (easier) or run time (preferred).

How to Compile and Run Your Program

Compile your program using the following:

```
<compiler> <options> -o <executable> <sources>
```

where:

- `<compiler>` is one of `g++` or `clang++`
- `<options>` is all of `-g3 -O0 -std=c++17 -pedantic -Wall -Wold-style-cast -Wextra -Woverloaded-virtual -I./`
- `<executable>` is the name of the executable file being created
- `<sources>` is a whitespace delimited list of C++ source files

For example:

```
g++  -g3 -O0 -std=c++17 -pedantic -Wall -Wold-style-cast -Wextra
     -Woverloaded-virtual -I./ -o VMManager   main.cpp TLB.cpp
      LRU_Replacement.cpp PageTable.cpp
```

Your program shall be run as follows:

```
./VMManager <addresses.txt 2>&1 | tee output.txt
```

Your program shall read in the file `addresses.txt`, which contains 1,000 logical addresses ranging from 0 to 65535, by redirecting `addresses.txt` to standard input as shown above. Your program is to translate each logical address to a physical address and determine the contents of the unsigned byte stored at the correct physical address. (Recall that in the C++ language, the `unsigned char` data type occupies a byte of storage, so we suggest using `unsigned char` values. The data type `char` will not work correctly.

Your program is to output the following unsigned hexadecimal 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 unsigned signed byte value stored in physical memory 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.

Page Replacement[3]

Thus far, this project has assumed 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. This phase of the project now assumes using a smaller physical address space with 128 frames rather than 256. This change shall require modifying your program so that it keeps track of free frames as well as implementing two page-replacement policies using FIFO and LRU (Section 10.4) to resolve page faults when there is no free memory. You shall implement both a FIFO and an LRU policy for page-replacement, but only one shall be used at a time selectable either at build time (easier) or run time (preferred).