CSCI 4061: Introduction to Operating Systems

Project 3 - Virtual-Memory Manager **Due: Thursday March 28, 2024**

Ground Rules. You may choose to complete this project in a group of up to three students. Each group should turn in **one** copy with the names of all group members on it. The code must be originally written by your group. No code from outside the course texts and slides may be used—your code cannot be copied or derived from the Web, from past offerings, other students, programmer friends, etc. All submissions must compile and run on any CSE Labs machine located in KH 4-250. A zip file containing all materials for your submission should be submitted through Canvas by 11:59pm on Thursday March 28, 2024.
Note: Do not publicize this assignment or your answer to the Internet, e.g., public GitHub repo.

Objectives: The main focus of this project is to implement (1) an algorithm that translates virtual addresses to physical addresses using two-level page tables (PTs) and (2) a Translation Lookaside Buffer (TLB) that implements the "First In First Out" (FIFO) policy for page replacement. The two parts should be completed by implementing the functions (with "TODO") in `/src/vmemory.c`.

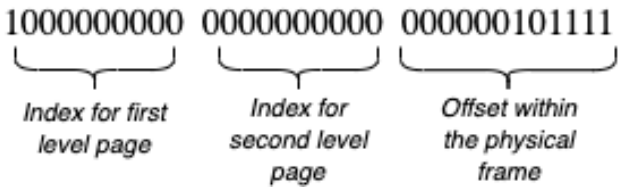
1 Translate Virtual Addresses to Physical Addresses

1.1 The Page Table Structure

For this project, we shall assume 32-bit physical and virtual addresses. Hence there are 2^{32} possible distinct memory locations. You will use a two-level page table, only accessible through the first-level page table's base pointer `cr3`. This pointer is already set up for you in `vmemory.c` using hidden code from `page_table.o`. The relevant specifications of the page table are as follows:

- The 10 most significant bits (MSB) of a 32-bit virtual address are used to index into the first-level page table, which has exactly 1024 page table entries (PTEs). Each entry contains a pointer to the base of a second-level page table.
- The next 10 bits of the virtual address are used to index into a second-level page table, which again has 1024 PTEs. Each PTE contains a physical frame number.
- The 12 least significant bits (LSB) determine the offset within a given page/frame, with possible addresses explicitly ranging from 0x000 to 0xfff.

See below for an example of how a virtual address should be decomposed in order to index into the page table. The address should be split into 3 pieces as follows:



1.2 Translating the Virtual Address

Using the `cr3` page-table pointer available in `vmemory.c`, your task is to translate each virtual address provided in the file `/bin/virtual.txt` to the respective physical address. Each virtual address is stored in hexadecimal format, and you should print the translated physical addresses in this same format. The translation should use the two-level PT as described in 1.1. A virtual address is valid if it has a physical frame number in the second-level PTs. That is, it is valid if the corresponding second-level page table is non-NULL and the corresponding physical frame number does not equal -1. Use the physical frame number from the second-level PT and the offset (12 LSB of virtual address) to determine the exact physical address in the case that it is valid. An invalid entry for a virtual address is called a page fault and should return -1. Specifically, for this part you should implement the following functions in `vmemory.c`:

1. `int translate_virtual_address(unsigned int v_addr)`
2. `void print_physical_address(int frame, int offset)`

Notice that the virtual address translation in `translate_virtual_address` takes in an `unsigned int` as a parameter to account for the entire range of 32 bits. However, the return type is an `int` to ensure you can return `-1`. Do not worry about -1 overlapping with some valid physical address; we will not test with any valid physical addresses equal to -1 when written as a signed integer. Additionally, treat any physical addresses in the page table equal to -1 as invalid addresses.
In `print_physical_address`, the printed physical address should be in hexadecimal format with exactly 8 digits in lower case, zero (if applicable) padded to the left, and preceded by "0x", such as in `0x009ea628`. When the frame parameter is `-1`, just print `-1`. The inputs are the physical frame number and offset parameters that would make up a full physical address.

2 Implementation of the TLB

In the second part, you should implement a TLB. While, in practice, the TLB is typically in CPU, we shall emulate it as a buffer in the main memory in this project. Your TLB should contain up to 8 frequently used virtual address entries, to avoid page table lookup latency. The TLB should be filled whenever there is a TLB miss but no page fault in the PT lookup. When the TLB is full, the page that was first added in should be removed, following the FIFO policy. One way to implement the TLB is as a 2D int array that implements the queue data structure. The first entry in the TLB should contain the 20 MSB of the virtual address page number and the second entry should contain the physical frame base address. The offset is not needed to be stored in the TLB, as you are simply caching the page address translations. Specifically, implement the following functions in `vmemory.c`:

3. `int get_tlb_enty(int n)`
4. `void populate_tlb(int v_addr, int p_addr)`
5. `float get_hit_ratio()`
6. `void print_tlb()`

Feel free to declare extra global variables to aid in implementing these functions and the TLB in general. Some requirements for the above four functions:

- `get_tlb_enty` should take in an `int` parameter containing the 20 MSB of the virtual address. It should return the physical frame base address mapped in the TLB to that particular virtual address. A TLB miss should return `-1`.
- `populate_tlb` should take in the 20 MSB of a virtual address, i.e., the virtual page number, and the 20 MSB of a physical address, i.e., the physical frame number, as parameters and populate the TLB with this new pair. Make sure to properly follow the FIFO cache replacement policy, or the LFU policy if you are implementing the extra credit.
- `get_hit_ratio` should provide the hit ratio of the TLB, which is the ratio of the number of times the TLB served a page request without invoking the PT to the number of total addresses translated, so far. The result of this call should change as the TLB runs; make sure you are tracking the relevant statistics in your other TLB functions.
- `print_tlb` should write the state of all of the 8 TLB entries to the output file `tlb_out.txt`. Each line in the output file should consist of two columns delimited by a single space containing the virtual page number and the physical frame number respectively, in hexadecimal. (20 MSB should give you 5 digits; the 5 digits should be preceded by `0x`. Pad 0s to the left if you would get fewer digits, as in `print_physical_address`). Initial TLB entries should be set to `-1`. Each call to `print_tlb()` should print a blank line to separate the TLB entries for the current call from the last and should *append* its contents to `tlb_out.txt`.

Note that `get_hit_ratio` and `print_tlb` can be called anytime during the program's execution, and must return the correct result for the exact point at which they are called. You can assume that neither function will be called inside of one of the 6 that you should implement, however.

3 The Main Program

After completing your address translation and TLB implementation functions, you should complete the main program in main.c. Your program should read in the file `virtual.txt`, translate all of the virtual addresses in it to physical addresses, and print the physical addresses, one on each line. You should update your TLB as you go by calling your TLB functions at appropriate points. Furthermore, you should write the contents of the TLB to `tlb_out.txt` both before and after you have translated all of the virtual addresses. Finally, you should print the hit ratio of your TLB to the screen. Feel free to do extra calls/testing of any of your TLB functions in the main before submitting, but only print the required information in your final submission. Also, make sure your code is general. We have provided you a `virtual.txt` file with which to test your code, but we will use a larger file for grading. Your main function must be able to handle a `virtual.txt` of arbitrary size!

4 Extra Credit: LFU

Implement a Least Frequently Used (LFU) page replacement policy along with the FIFO policy for your TLB. The LFU policy should be used instead of FIFO if a command line argument, `-lfu`, is

specified. This can be done by adding extra functions and/or cases to `vmemory.c` and `main.c` that are invoked if `-lfu` is specified. By default, without entering any command line arguments, your TLB should run the FIFO policy. Also compare and contrast the performance (hit ratio) of your TLB with the LFU vs. the FIFO policy in your README. Briefly explain the results you saw and give a possible reason(s) why this might be the case. Note that `print_tlb()` should be a general function that is able to print the TLB maintained by either FIFO or LFU, and that `populate_tlb` should populate based on the replacement policy currently in use.

5 Deliverables

Students should upload to Canvas a zip file containing their C code, a Makefile, and a README that includes the group member names, what each member contributed, any known bugs, test cases used (we will also use our own), whether the extra credit has been attempted, any assumptions outside of this document, and any special instructions for running the code.

6 Grading Rubric

- 5% For correct README contents
- 5% Code quality such as using descriptive variable names and comments
- 5% Correct invocation of function calls within `main.c`; the code does not print any extra characters other than the ones required (we will use `diff` to test the results). To get more info on `diff` use `man diff` in terminal
- 40% Correct address translations
  - 15% `translate_virtual_address` computes the correct physical address whenever the physical frame number is in the PT
  - 10% `translate_virtual_address` returns `-1` without error whenever the physical frame number is absent from the PT
  - 15% `print_physical_address` prints the physical address in the correct format
- 45% Correct implementation of FIFO TLB
  - 5% Uses a reasonable representation for the TLB in memory that stores up to 8 virtual address mappings
  - 5% `get_tlb_entry` returns the correct physical frame number if it is present in the TLB
  - 5% `get_tlb_entry` returns `-1` if the virtual address is not mapped in the TLB
  - 10% `populate_tlb` correctly implements the FIFO replacement policy if no `-lfu` flag
  - 10% `get_hit_ratio` returns the correct hit ratio (includes setting needed variables in other functions)
  - 10% `print_tlb` correctly *appends* the TLB state to `tlb_out.txt`
- 5% Extra credit for correctly implementing the LFU policy and comparing its performance to FIFO

7 Example Translation

Consider the virtual address `0x72ae2247` from `virtual.txt`. Its binary transformation is `0111001010 1011100010 001001000111`. Its 10 MSB (0111001010) in decimal is `458`. We thus look for the base pointer (i.e., the base of the corresponding second-level page table) with index `458` in the first-level page table, and check its contents to find a number that in decimal is `738`. The entry in the second level page table with index `738` is `0x71728`. This is the physical frame base address. The offset of the virtual address is `0x247`. Thus, the final physical address is `0x71728000 + 0x247 = 0x71728247`. In particular, if `virtual.txt` were to only contain `0x72ae2247`, your program should output as follows, where **bold** denotes user input:

./vmanager
0x71728247
Hit rate of the TLB is 0.000000

Note that the hit rate is zero since the TLB was cold at the start. For larger `virtual.txt`, the hit rate should in general be nonzero.

8 Testing & Execution

Run `make` to compile the program and then run the program using **./vmanager**. Pass the argument `-lfu` to run the program with the LFU replacement policy for TLB. On successful execution `tlb_out.txt` will be created. Subsequent runs of **./vmanager** will append the content of TLB to this file. Run `make clean` to remove the executable and delete the output file `tlb_out.txt`. All the `.txt` files are inside `src/bin/` folder.

Expected output of `tlb_out.txt` for the given `virtual.txt`:

```
-1 -1
-1 -1
-1 -1
-1 -1
-1 -1
-1 -1
-1 -1
-1 -1

0x72ae2 0x71728
0x4f65d 0x77c6f
0x2fcd8 0x763a1
0xaf199 0x78ac5
0x19c78 0x73bf9
0x6f25e 0x770e0
0xe9bb1 0x7be7e
0xfffff 0xffffffff
```

Expected output on the `stdout` for the given `virtual.txt`:

```
0x7084c0fe
0x77c6f4dc
0x763a11b8
0x78ac522d
0x73bf9df7
0x770e02a5
0x7be7e7bf
-1
0x770e02a5
0x71728247
Hit rate of the cache is 0.100000
```

Hints & other requirements Don't modify `vmemory.h`; instead implement helper functions via the static qualifier. Other than error checking and required prints, avoid any other prints in vmemory.c. Do not modify anything in the `bin` directory except `virtual.txt` for testing purposes, and `tlb_out.txt` during execution of your program. After using make, run the executable with `./vmanager` to run the program with FIFO or with an optional parameter to run the LFU policy (as `./vmanager -lfu`). A missing specification can be implemented by any reasonable assumption, though you should specify in your README which assumptions you made.