CS 367 Project #4: Paged Memory System

Due: Friday, May 4, 11:59 PM

This is to be an individual effort. No partners.

For this assignment, you are going to use C to implement a virtual to physical address mapping using bit-level operators. Specifically, you will be building a paged memory management system with three components (CPU with TLB, Cache, Memory). For a given virtual address, your program will output the associated byte in that virtual address.

The format of your virtual and physical addresses will be as follows:

- Virtual address – 15 bits (8 bit VPN, 7 bit VPO)
- Physical address – 17 bits (10 bit PPN, 7 bit PPO)

The main memory and page table contents for the process will be provided for you in a supporting archive file which is described below. Since the virtual address and physical address are fewer than 32 bits in length, you will use standard 32 bit integers to represent them. **You must use 32-bit integer variables and bit-level operators to do your work.** Use masks and shifting to get to the parts of the addresses you need to use at a given time.

For a given virtual memory address, **you will first compute the physical address** by first checking the TLB, and if the data (PPN) is unavailable in TLB, by obtaining it from the page table. Once you have the physical address, you will access cache, and if necessary main memory to read the data. In this project, the disk subsystem is not considered, so assume that each virtual page is allocated to a certain physical page in the main memory at all times.

The page table will be given to you. You **must** use the provided logging functions (described here and in the code) to document where you are getting your information (e.g., TLB vs page table) and what the computed physical address is.

- TLB – you will be implementing a TLB to cache virtual-to-physical page mappings. Implement it as a direct-mapped cache which will be initially be empty (i.e., all entries marked valid = 0). You will update and use this data structure based on the memory addresses provided**. For the TLB, you will use 5 bits of the VPN as the tag (TLBT) and the remaining 3 bits for the index (TLBI).** Remember that the data that the TLB holds are the recently used PPNs.
- Page table – the supporting library has a function `get_page_table_entry(VPN)` which will return the associated PPN from its internal data. Make sure to update the TLB if you have to get the virtual-to-physical page mapping information from the page table.

Once you have computed the physical address, you will use this address to get your data. To do this, you will first check the cache for the data. If it is not there, you will get it directly from main memory. Just as with computing the address, you **must** use the logging functions to document what is going on.

- Cache **–The first 9 bits of the physical address with be used for the tag (CT). The next 6 bits will be used for the index (CI)**. Your cache will hold 4 bytes of data – use the last two bits of the physical address (CO) to get the correct byte to return. Just as with the TLB, this cache

should initially have everything marked as invalid. The data will be stored in little-endian format, for example, if the data in the cache is 0x12345678 and the offset (CO) bits are 01, you would be returning 0x56.

Your cache will be 2-way associative, meaning each set has two possible entries. The added issue here is that when some data is put in a cache set, you have two possible location choices (two lines). Use a single "valid/invalid" bit for each line.

You will use the following 'replacement' algorithm in case you need to evict a cache line:

- If both entries are invalid/unused, replace the first one
- If one entry is valid and the other invalid/unused, replace the invalid/unused one
- If both entries are valid, replace the 'oldest' one – the one that has been in the cache the longest. To determine this, you will need to keep additional information around. You can, for example, store a timestamp like value and always replace the entry with the lowest timestamp value (the "oldest" one).

- Main memory – the supporting library implements the main memory access via a function `get_word(addr)`, which returns a 4-byte integer (the "block") at the address given by `addr & ~0x3`. For example, if you request the byte at address 0x1, you will get the 4 bytes starting at 0x0 (0x0, 0x1, 0x2, 0x3). Make sure to update the cache if you have to get data from the main memory.

The program input will be a series of virtual addresses (the program exits when it receives a negative integer as the next virtual address to translate).

Starting the Assignment

The starting tar file (project4_handout.tar) contains five files:

- memory_system.h – starting header file. You may add data structure definitions here. It currently only contains some constant definitions you can (and should) use.
- memory_system.c – driver program implementation. It reads in the input and calls your functions to translate the virtual address to a physical one and then get the relevant byte. You do not need to modify the code in this file.
- caching.c – **This is the file you will edit.** It already contains some stubs used by the driver to do the computations. You may write other functions as well. **Put all of the code for your assignment in this single file**.
- libsupport.a – static library file contains functions `get_word()`, `get_page_table_entry()` and `print_page_table()`. This last function was added in case you may find it useful.
- Makefile

As mentioned above, you will make logging calls to record what is happening with your implementation. Those logging calls are made by the `log_entry()` functions invoked within `caching.c`. Each `log_entry()` call will update the logging file named `project4_logfile` in your project directory. You should make the necessary `log_entry()` calls within `caching.c` to update the log file. – **The resulting log file** `project4_logfile` **will be examined to determine if your implementation is correct.** As with prior assignments, a reference implementation is provided to you – you can access it at `/home/cs367/bin/mem`. The reference implementation generates a log file named `project4_logfile_expected` in your project directory. Your log files should be identical to those generated by the reference implementation.

Implementation Notes

- **You must use 32-bit integer variables and bit-level operators to do your work.** Use masks and shifting to get to the parts of the address you need to use at a given time.
- **You can implement TLB and cache in any way you want** as long as they behave correctly as specified. Of course, use good C programming style and document your code so that the grader can see clearly your design decisions.
- **It may be a good idea to develop your program incrementally.** That is, first you can make sure that every virtual address is successfully translated to the physical address by using the page table, and the data at that physical address is read directly from the main memory. Once those steps are implemented and debugged properly, then you can add TLB and cache accesses as intermediate steps.

Submitting & Grading

Submit `caching.c` on Blackboard. It must still build with the rest of the code using the given `Makefile`. ONLY CHANGE AND SUBMIT THIS FILE (`caching.c`).

Make sure to put your name and G# as a commented line in the beginning of your program. Also, in the beginning of your program list the known problems with your implementation in a commented section.

No hard copy submission is required in this project. Questions about the specification should be directed to the CS 367 Piazza forum. However, recall that debugging your program is essentially your responsibility; so please do not post long code segments to Piazza.

Your program will be compiled and graded on zeus. If your program does not compile on zeus, we cannot grade it. If your program compiles but does not run or generate output on zeus, we cannot grade it.

Your score will be determined as follows.

- 80 points – Correctness: This includes aspects such as generating correct address and data, and getting the data from the correct location (i.e., TLB vs. page table and cache vs. memory). To get credit, you must make logging calls to record what is happening with your implementation so that the grader can follow all the steps in the address translation process.

- 20 points – Code & comments: Be sure to document your design clearly in your code comments. This score will be based on (subjective) reading of your source code by the grader. The grader will also evaluate your C programming style.

  No late work is allowed after 48 hours; each day late uses up one of your existing tokens. If you submit at a time beyond what is allowed by your existing late tokens, your project will be subject to 25% penalty for each extra day.