CS367 Project #3: Paged Memory System
Due: Sunday, December 9th at 11:59PM

This is to be an individual effort. No partners.

Before you Start: Make sure you can do Recitation 12 completely. The concepts from the entire recitation will be used when you design your solution for this project. This project requires a good understanding of how the Virtual Memory System and Memory Caching works.

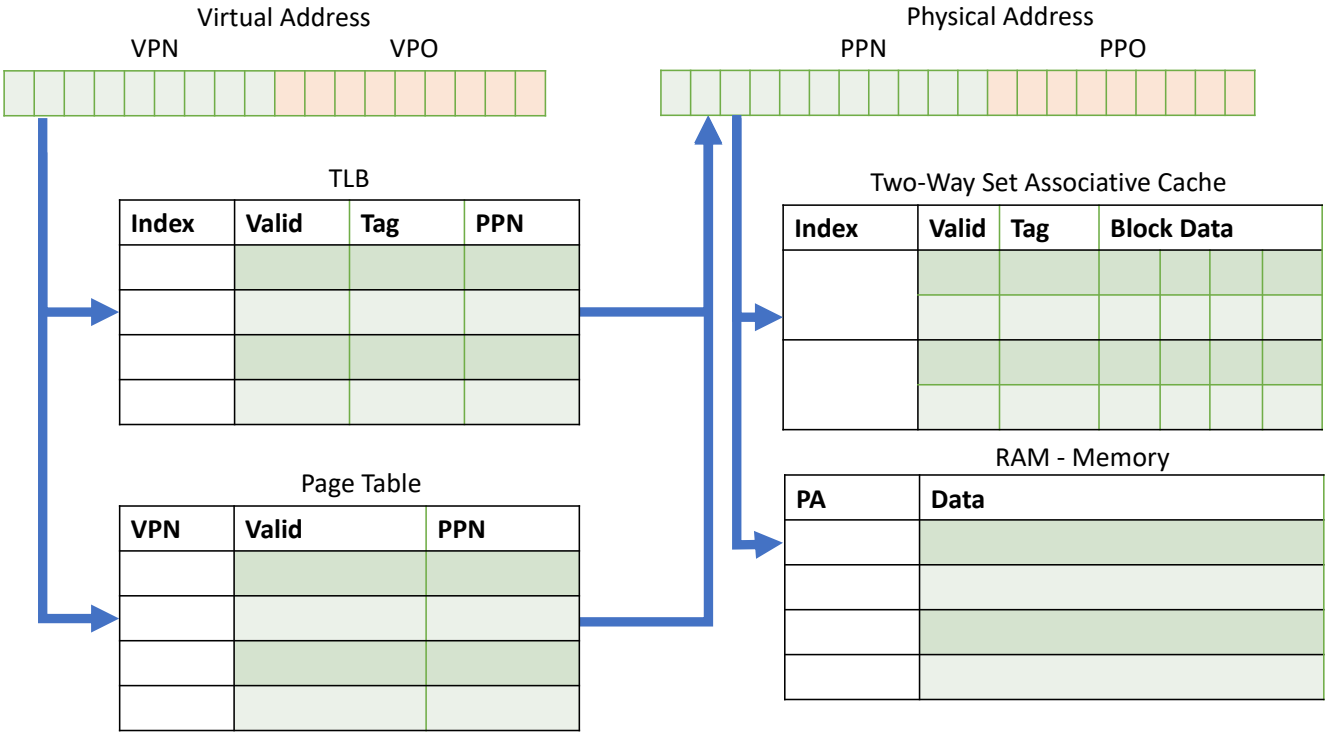
Overview

For this assignment, you are going to use C to implement a Virtual Address to Physical Address memory mapping using bit-level operators for the given scheme:

- Virtual Address 18 bits (9 bit VPN, 9 bit VPO)
- Physical Address 20 bits (11 bit PPN, 9 bit PPO)

You will be building a paged memory management system with three components, CPU, Cache, and Memory, where for a given virtual address, you will output the associated byte in the system memory.

Visual Representation (Simplified)



Details of the Implementation

The project handout consists of a tar archive that contains a directory and a series of files. Of these, you will be modifying **caching.c** and you have the option of creating a header file to use with it.

Your objective is to modify **caching.c** to provide an implementation for the TLB, a Page Table, and Memory Caching, along with Virtual Address to Physical Address translation.

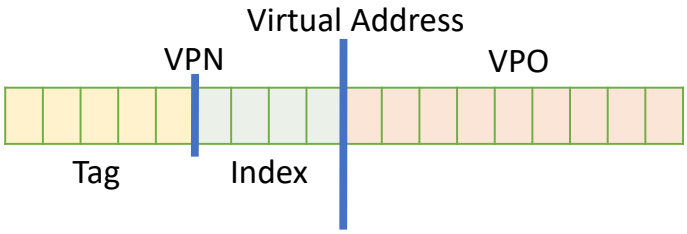
The main memory will be provided for you in a supporting archive (**libframesupport.a**), which is described below. Since the virtual addresses and physical addresses are shorter than 32 bits in length, you will use the standard 32-bit `int` data type in C to represent them. **You must use an integer (and bit-level operators) to do you work.** Use masks and shifting to get these parts of the addresses you need to use at a given time.

The program input will be a series of virtual addresses. The input may be made directly when running the program by entering addresses in decimal at the prompt. You may also create testing files that consist of one address on each line, with -1 as the last address. When the program encounters a negative number, it will quit. This testing file may be read into the program using the Unix redirect in operator (demonstrated in the sample output section). Each line of input represents a virtual address that a program wants to read memory from.

For a given virtual memory address, **you will first compute the physical address** by first checking the TLB. If the Physical Page Number (PPN) is in the TLB, then you can get it and then return the Physical Address (PA). Otherwise, if the data is unavailable, you will try to get the PPN from the page table using the VPN. When accessing the Page Table, you will need to check to see if the page has been loaded in to RAM and has an entry in the page table. If so, you can then update the TLB with the information and return the PA. If not, you will use a supporting library function to load the page into memory, after which you can update both the page table and the TLB with the PPN before returning the PA.

You must use the provided logging functions (described here and in the code) to document where you are getting your PPN (TLB, Page Table, or Page Fault Handler) and what the physical address is.

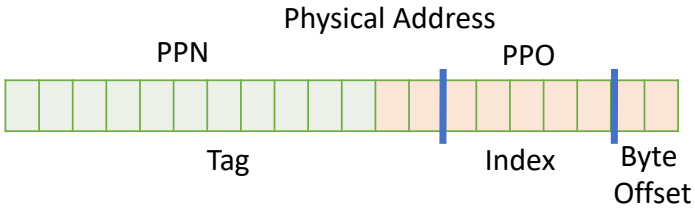
- **TLB**: You will be implementing a TLB to cache memory addresses. Implement it as a direct-mapped cache, which will initially be empty (i.e. all entries have valid set to 0), but you will update and use this data structure based on the memory addresses provided. **For the TLB, you will use the first 5 bits of the VPN as the tag and the remaining 4 bits for the index.** The data the TLB will hold are the most recently used PPNs (11-bits).



- **Page Table:** You will be implementing a simple page table that is indexed by the VPN and consists of a **valid bit** and the **PPN** entry for the VPN index. If the data is not in the TLB, you will search the page table for the value. If you find the value in the page table, make sure to update your TLB with the information! If you do not find the value in the page table, then you will need to call the library function, **load_frame(vpn)**, which will load the frame from memory and return the PPN where it is loaded. You will then update the page table AND the TLB with the information so the next time you search for it, it will be in both places. For this project, once you have loaded a frame, it will stay in memory. You will not have to remove anything from your 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 memory cache for the data. If it is not there, you will need to get it directly from main memory. Just as computing the address, you **must** use the logging functions to document where your data came from (Cache or Memory).

- **Cache** - The **first 13 bits of the physical address will be used for the tag. The next 5 bits will be for the index and the last 2 bits is the offset.** Your cache will hold 4 bytes of data; use the last two bits of the physical address to get the correct byte to return.



- Just like the TLB, this cache should initially have everything marked as invalid. The data will be stored little endian, meaning if the data in the cache is 0x12345678 and the offset bits 1, you will be returning 0x56.
- **This will be a two-way set associative cache**, meaning each set has two possible entries. The added issue here is that when you put something into the cache, you have two possible location choices. So that we are all doing the same thing, our replacement algorithm is:
    - **If both entries are invalid, use the first one**
    - **If one entry is valid and one is invalid, use the invalid one**
    - **If both entries are valid, use the oldest one**
- **Oldest entry:** To figure out which entry is the oldest one, you should use a timestamp like approach, where you always replace the entry with the lowest timestamp.

Main Memory: The supporting library implements main memory via a function **get_word(addr)** which returns a 4-byte integer at address (addr &~0x3). For example, if you request the byte at address 0x101, you will get the 4 bytes starting at 0x100 (0x100, 0x101, 0x102, and 0x103). Be sure to update the cache if you have to get data from memory.

Starting the Assignment

The starting tar file (project3.tar) has 5 files in it:

- **memory_system.h** : This is the starting header file. It currently only contains some constant definitions you should use. **Do not modify this code.**

- **memory_system.c** : This is the driver program implementation. **Do not modify this code**.

- **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.

- **libframesupport.a** : This is a static library that contains several functions needed to make the entire system work. You will be using **get_word** and **load_frame** functions.

- **Makefile** : This is a makefile that will make and clean your program. When you make it, you will get an executable called **mem**

You may add a header file if you like for your caching.c implementation.

Implementation Notes

- **You must use an integer (and bit-level operators) to do your work.** Use masks and shifting to get the parts of the address you need to use at a given time.
- **You may implement the TLB, Page Table, and Cache however you want** as long as it behaves correctly as a cache given the specification above.

Submitting and Grading

Submit your **caching.c** and any needed additional header files (not included in the handout) on **blackboard** as a tar or zip file. Be sure this file contains everything you need -- **incomplete submissions cannot be graded**. All submissions will be **tested and graded on Zeus!** Make sure you test your code on Zeus before submitting.

If you have any remaining tokens, you may use them.

Your grade will be determined as follows:

- **80 points** - Correctness. To be completely correct, you have to both generate the correct addresses and data AND get the information to do the generation **from the correct location** (ie. TLB vs page table vs page fault handler and cache vs memory). As with prior assignments, if we cannot compile the code or we can't get output, your score will be very low.
- **20 points** - Specifications. Following specifications, use of C, comments, bit-level operators...

Use of the Logging Functions

The files **memory_system.c** and **memory_system.h** provide the functions for creating the output log file. You must use these functions at the appropriate times to get the right output for your program. PA, VA, and Data refer to the Physical Address, Virtual Address, and Data found. Use whatever variables your function defines for these arguments. The first argument to the log_entry function is a pre-defined constant.

Log that you found the PPN in the TLB directly.

log_entry(ADDRESS_FROM_TLB, PA);

Log that the PPN was not in the TLB, but you did find it in the Page Table.

log_entry(ADDRESS_FROM_PAGETABLE, PA);

Log that the PPN was not in the TLB or the Page Table, so you had to handle the Page Fault

log_entry(ADDRESS_FROM_PAGE_FAULT_HANDLER, PA);

Log that the Data was found in the Cache

log_entry(DATA_FROM_CACHE, Data);

Log that the Data was not found in the Cache, so you had to read Main Memory

log_entry(DATA_FROM_MEMORY, Data);

Log that the input was an illegal virtual address

log_entry(ILLEGALVIRTUAL, VA);

Sample Input

This is a sample input file, called test0. To run with a test file, use the input redirection (<) operator. To end a test run, use a negative address, such as -1.

kandrea@zeus-1:~/p3/ref_project3$ cat test0
126
127
17001
1001
17001
17001
-1

Sample Output

When you run your program, you may have it output anything you feel is necessary to the screen, however, your program will also automatically generate and update a file called **project3_logfile** when you call any **log_event** function, which is what we use to grade your program output.

This is the output after the reference program runs test0.

kandrea@zeus-1:~/p3/ref_project3$ ./mem < test0

...

kandrea@zeus-1:~/p3/ref_project3$ cat project3_logfile

```
Virtual Address: 0x7e
        Physical Address: 0x7847e from Page Fault Handler
        Data: 0xae from memory
Virtual Address: 0x7f
        Physical Address: 0x7847f from TLB
        Data: 0x52 from cache
Virtual Address: 0x4269
        Physical Address: 0x96e69 from Page Fault Handler
        Data: 0xd3 from memory
Virtual Address: 0x3e9
        Physical Address: 0x1e5e9 from Page Fault Handler
        Data: 0x54 from memory
Virtual Address: 0x4269
        Physical Address: 0x96e69 from Page Table
        Data: 0xd3 from cache
Virtual Address: 0x4269
        Physical Address: 0x96e69 from TLB
        Data: 0xd3 from cache
```

This output shows that the first VA has no entry in the TLB or the Page Table (cold miss) and had to call **load_frame(vpn)** to start request that the system load a new page into memory. This function returns the ppn, which your code will then add to both the Page Table and the TLB. When it looked for the data at that address, it also got a cold miss in the cache, so had to pull it from main memory and then added the block to the cache.

The second address is in the same page, so it used the TLB, and it is also in the same cache block. The third and fourth addresses are both in different pages, so they had to be handled as page faults when first read. These addresses also have the same TLB set index, so they overwrite each other in the TLB! This is why the second 0x4269 read was not in the TLB, but the third one was.

Your program will be graded based on the output project3_logfile.