CS367 Project #3: Paged Memory System
Due: Monday, May 6th at 11:59PM

This is to be an individual effort. No partners.

Before you Start: This project requires a good understanding of how Caching and Virtual Memory System works. As the first step, you should review caching and virtual memory lectures carefully. Also, make sure you can do Recitation 11 completely. Once released, Recitation 12 will also prove greatly beneficial for this project. The concepts from these two recitations will be important when you design your solution for this project.

Overview

For this assignment, you are going to use C to implement a Virtual Address to Physical Address memory mapping system **using bit-level operators**.

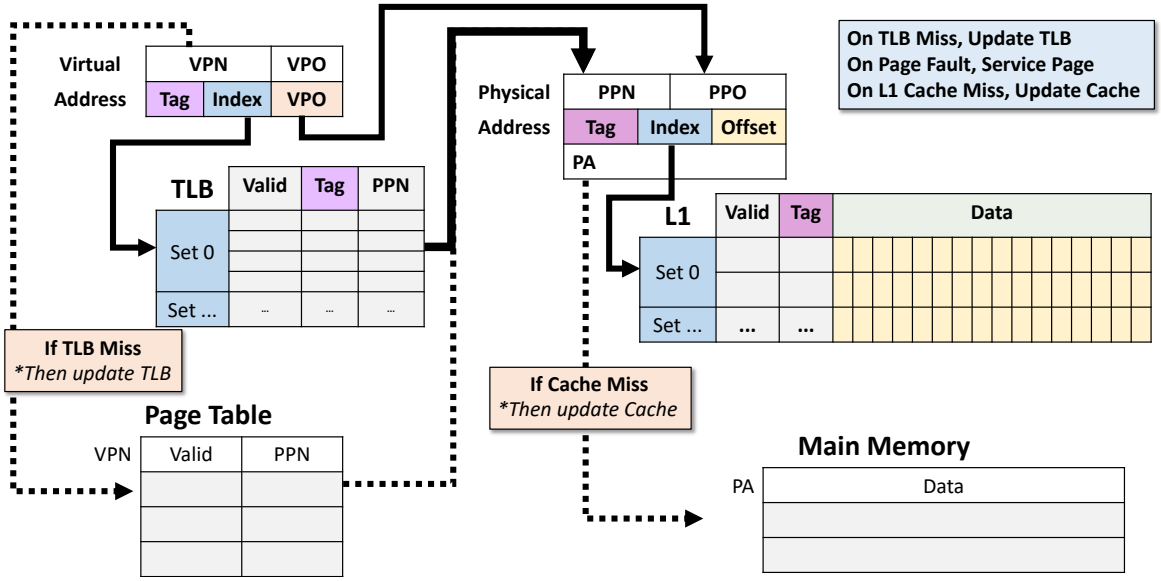
You will be building a paged memory management system with three components: TLB, Page Table, and L1 Cache. Your TLB will be 4-way Set Associative and your L1 Cache will be 2-way Set Associative. Other details of the TLB, L1 Cache, and Page Table organization are given on the following pages.

You will be designing and implementing these three components in two required functions (see caching.c) and any other functions you wish to create to support them. For your part of this program, you will be given a virtual address and will output the associated byte in that virtual address.
You will be able to design and implement your components using any data structures you like.

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

- Virtual address – 24 bits (18 bit VPN, 6 bit VPO)
- Physical address – 16 bits (10 bit PPN, 6 bit PPO)

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 submitting it for grade. You do have the option of creating a header file to use with it. You may **not modify** any other provided file and you will only submit **caching.c** and any header file you may create for 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 (**libsupport.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 int (and bit-level operators) to encode and use each address.** Use masks and shifting to get the parts of the address you need to use at a given time.

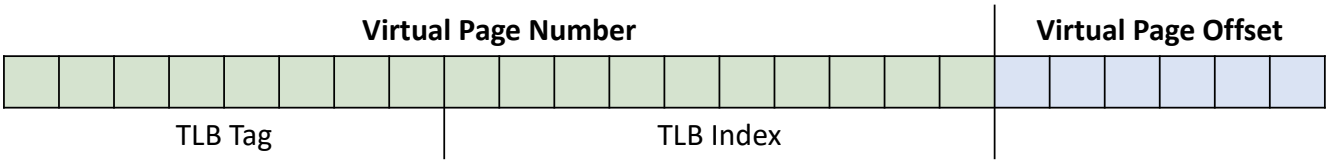
This program, like with Project 1, uses a prompt system for your input. Your program, **mem**, is made using the other files given in the handout. The program input will be a series of virtual addresses that are entered at the prompts. When the program encounters a negative number, it will quit. The input may also be put into script files to allow you to run using a file for your inputs. These testing scripts must consist of one virtual address on each line, with -1 as the last address. 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. The first step will be to check your TLB to get the PPN to compute and return the Physical Address (PA). If you have a TLB miss, you will try to get the PPN from the page table using the VPN, then update the TLB and return the PA. If it is not in the page table, you will use a supporting library function to load the page into memory, after which you can update the page table and the TLB, before returning the PA.

Once you have the PA, you can then check the L1 Cache to fetch the byte requested. If the data is not in cache, then you can use a provided function to get the data from RAM, which can also be used to update your Cache block using the given replacement policy.

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.

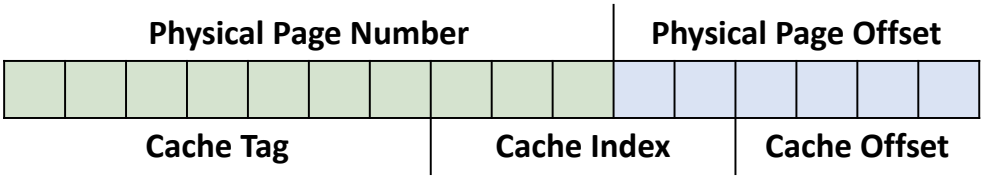
- **TLB**: You will be implementing a TLB to cache PPNs. Implement it as a **4-way set associative cache**, which will initially be empty (i.e., all entries have valid set to 0), but you will update and use this data structure based on the memory addresses provided. **For the TLB, you will use the first 8 bits of the VPN as the tag and the remaining 10 bits for the index.** The data the TLB will hold are the most recently used PPNs (10-bits).
  (How many total entries do you have in the TLB? You should be able to figure out!)



- **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, **handle_page_fault(vpn)**, which will load a physical page to 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 page into main memory, it will stay there. 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 7 bits of the physical address will be used for the tag. The next 5 bits will be for the index and the last 4 bits is the offset.** Your cache will hold 16 bytes of data; use the last four 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 in Main Memory as **little endian**, meaning if 4-bytes of data in the cache at the address has the value 0x12345678 and the offset bits value equals 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. (How many total lines do you have in the L1 cache? You can figure out!)

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. This will only get one 4-byte set of data from memory. Your cache entry can hold 16 bytes of data, so make sure to fill your cache properly when using this API function.

Cache Replacement Algorithm (L1 and TLB): For this project, we will be using the **Least Recently Used (LRU)** cache replacement algorithm. Here are the steps for updating an entry:

1. **If any of the entries are invalid, use the first invalid one.**
2. **If all of the entries are valid, use the one that is least recently accessed.**

- **Least Recently Accessed Entry:** To figure out which entry is the least recently accessed one, you should use a timestamp like approach, where you always replace the entry with the lowest timestamp. This also means you need to update the timestamp **on each access (hits and when replacing the entry after a miss),** to keep track of the time of last access to an entry. This timestamp may be any format you feel appropriate. This is a part of your design.

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.

- **libsupport.a** : This is a static library that contains several functions needed to make the entire system work. You will be using **get_word** and **handle_page_fault** 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 (but then you must submit it in your tar/zip file).

Implementation Notes

- **You must use an int (and bit-level operators) to encode and use each address.** 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. 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 ONLY, 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 L1 cache accesses as intermediate steps, and complete your project.

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 as **caching.tar** or **caching.zip**. No other naming formats are needed for this. Be sure this file contains everything you need -- **incomplete submissions cannot be graded**.

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

All submissions will be **compiled**, **tested, and graded on Zeus!** Make sure you test your code on Zeus before submitting. 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.

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.

You **may use up to one late token** on this project. The final date this can be submitted is May 7th, which is the final date you can submit an assignment by University policy. We cannot extend this further for any reasons. If have no late tokens, the 25% ceiling penalty will be applied (see syllabus)

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** (i.e., TLB vs page table vs page fault handler and L1 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.

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. The log that these functions create, **project_3_logfile**, is what is used for grading. These **log_entry** calls generate this output file that we use for grading.

Below is a list of sample calls to the function using different **codes**. It is your responsibility to figure out when each function should be used, considering your own program design and the rules for address translation.

For example, if you find the PPN in the TLB, you will call **log_entry(ADDRESS_FROM_TLB, PA);** where PA is the physical address you generated from the PPN and PPO. If you then have a Cache Miss and get the final byte from Memory, your program would also call **log_entry(DATA_FROM_MEMORY, Data);** to log that you got your data from RAM.

Listing of Logging Functions and Codes

Address Translation: Log where you found the PPN using one of these three functions.

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(PAGEFAULT, PA);

Data Fetch: Log where you found the Data using one of these two functions.

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);

Administrative: Log at the beginning of your **get_physical_address** function if the input is invalid.

Log that the input was an illegal virtual address

log_entry(ILLEGALVIRTUAL, VA);

Sample Input

This is a sample input file, called script0. To run with a test file, use the input redirection (<) operator. To end a test run, use a negative address, such as -1.

```
zeus-1:~/p3$ cat script0
0
65536
131072
196608
0
262144
0
65536
-1
zeus-1:~/p3$ ./mem < script0
...
```

Sample Output

When you run your program, you may have it print 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. This logfile is what we use to grade your program output.

This is the output after the reference program runs script0.

```
zeus-1:~/p3$ cat project3_logfile
Virtual Address: 0x0
       Physical Address: 0xf080 from Page Fault Handler
       Data: 0x78 from memory
Virtual Address: 0x10000
       Physical Address: 0xa780 from Page Fault Handler
       Data: 0xee from memory
Virtual Address: 0x20000
       Physical Address: 0x8c40 from Page Fault Handler
       Data: 0x30 from memory
Virtual Address: 0x30000
       Physical Address: 0x8880 from Page Fault Handler
       Data: 0x82 from memory
Virtual Address: 0x0
       Physical Address: 0xf080 from TLB
       Data: 0x78 from cache
Virtual Address: 0x40000
       Physical Address: 0xac80 from Page Fault Handler
       Data: 0x1a from memory
Virtual Address: 0x0
       Physical Address: 0xf080 from TLB
       Data: 0x78 from cache
Virtual Address: 0x10000
       Physical Address: 0xa780 from Page Table
       Data: 0xee 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 **handle_page_fault(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 first four addresses are in different pages and reference different areas of memory, so there are no hits for them. The fifth address is the same as the first one, so you can see it was found in the TLB and its data was in Cache. The next address, however, gives us a problem. All of the addresses used in this script, as it turns out, go to TLB Set 0. Since it is a 4-way set associative, the first four can all be in the TLB at the same time, however, with this next address, it will overwrite the entry that was used least recently. Looking at the script, we accessed 0x0, 0x10000, 0x20000, 0x30000, 0x0, then 0x40000. So the least recently used one will be 0x10000, and that entry was replaced by 0x40000. When the script tries 0x0 again, it's still in the TLB, even though it's the oldest (Make sure to understand the difference between being the oldest and being the least-recently-used entry). But 0x10000 is not as it was evicted for being the least recently used.