Introduction

In this project, you will be implementing a virtual memory system simulator. You have been given a simulator which is missing some critical parts. You will be responsible for implementing these parts. This is quite an easy assignment, and comments have been given to guide you along. However this still builds up on fundamental paging concepts. We have posed some conceptual questions for each section of the project, and it is in your best interest to answer them before working on that section of the project. If you have trouble answering these questions, step back and review virtual memory concepts before coding.
This assignment has six main steps

1. Splitting the virtual address into VPN and page offset
2. Translating the virtual address into the physical address
3. Handling page faults
4. Finding an optimal page to remove
5. Adding a TLB to improve translation speeds
6. Computing the average access time (AAT) of the virtual memory system

Each part starts with some theoretical questions. You **do not** have to submit answers to these questions, as they are only there to help guide you along. If you are able to answer the questions without much trouble, you will have an easy time with the code. If you have trouble with the questions, then we **strongly suggest** that you take the time to read about the material from the book, and class notes.

You will be responsible for writing some critical functions in the simulator. There are excellent comments to guide you along, and help you accomplish each challenge. You should start with answering the conceptual questions given in this PDF, and then proceed to working on the code in the `student-src` folder of the project. All the functions you have to write can be found here. The `simulator-src` folder may be helpful for looking at struct definitions and globals, however you will not need to modify any code there.

Step 1 - Split the Address

In most modern operating systems user programs access memory using virtual addresses. The hardware and the operating system work together to turn the virtual address into a physical address, which can be used to address into physical memory. The first step of this process is to translate to split the virtual address into two parts: the higher order bits, known as the **VPN** and the lower order bits known as the **page offset**.

Conceptual Questions

Consider a machine in which virtual addresses are made up of 16 bits. Each page on this machine is 256 (2^8) bytes big:

1. How many bits should the page offset be?
2. How many bits should the VPN be?
3. What are the VPN and offset of the address 0xDEAD?
4. What are the VPN and offset of the address 0xBEEF?

Implementation

Look at the file `address_split.c`. You should find two functions, used by the simulator to split the address, there. These are called:

- `uint64_t get_offset(uint64_t virtual_address)` - This functions returns the page offset of the passed in virtual address.
- `uint64_t get_vpn(uint64_t virtual_address)` - This functions returns the vpn of the passed in virtual address.

Fix the **TODO** sections of these functions, so that they return the proper offset and vpn. Here's a **hint:** The global variables in `global.h` of the `simulator-src` folder may be helpful.

Step 2 - Address Translation

Now that we can split the address, we can transform the virtual address into a physical address. This requires a page table. Recollect that the page table is a per process data structure that stores virtual page number to page frame number (aka physical frame number) mappings. In the simulator, every process has a page table represented as an array of structs of the below type:

```
typedef struct page_table_entry_t {
    uint64_t pfn; /* Physical Frame Number */
    uint8_t valid; /* Valid 'bit' */
    uint8_t dirty; /* Dirty 'bit' */
    uint64_t frequency; /* Counter for keeping track of accesses */
} pte_t;
```

Notice that the VPN does not appear as part of the page table entry. You will index into the array of the page table entries using the VPN.

Conceptual Questions

Most of the address translation deals with reading values from the page table. The table below is similar to a `pte_t` array that is used in the simulator, although we have reduced the size of the VPN and the PFN (Physical Frame Number) to simplify the table below. Assume that the page size is the same page size you determined in the previous part. Assume that any VPNs that do not appear in the table are **invalid**.

Table 1: Page Table

VPN	PFN	Valid	Dirty
0xDE	0xBE	YES	NO
0xF0	0xD0	YES	YES
0xBE	0x42	YES	NO
0x42	0x43	NO	NO
—	—	NO	NO

1. What physical address is virtual address 0xDEAD mapped to?
2. What physical address is virtual address 0xF00D mapped to?
3. Is the virtual address 0x42ED valid?
4. What will happen when 0xF00D has to be kicked out from the page frame it is using.

Implementation

After finishing the questions in the previous section, you should have a pretty good idea of how the page table is used when translating the virtual address into a physical address. Open up the file `page_lookup.c`. In this file, you have to write the `pagetable_lookup` function. You will need to check if the page table entry is valid. If it is not valid, increment the page fault statistics, and call the `pagefault_handler` function. You can find the header of the page fault function in the `page_fault.c` file of the `student-src` folder. Look at the `stats.h` file of the `simulator-src` folder to find more about the statistics struct. Make sure to increment the frequency of accesses for accessed page table entry in this function. More information has been provided in the form of comments in the source file.

- You do not need to worry about checking the TLB first. This is done for you by the simulator.
- **Note that in the questions, you were asked to find the physical address. In the function, you simply need to find the Physical Frame Number - these are slightly different values!**

Step 3 - Efficient Page Replacement

Recollect that when the CPU encounters an invalid VPN to PFN mapping in the page table the OS allocates a new frame for the page by either finding an empty frame or evicting a page from a frame that is in use. In this section, you will be implementing an efficient page replacement algorithm. Look at the `find_free_frame` function in the `page_fault.c` file of the source.

Conceptual Questions

Answer the following questions about page replacement, given the following page table. Assume that any entry not listed is **VALID** and has a higher access count.

Table 2: Page Table

VPN	PFN	Valid	Dirty	Access Count
0xDE	0xBE	YES	NO	20
0xF0	0xD0	YES	YES	14
0xBE	0x42	YES	NO	1
0xBC	0x43	YES	NO	1
0x42	0xAB	YES	NO	12
0x21	0x23	NO	NO	34
—	—	YES	NO	YES

1. What is the first page that should be used when we need to allocate a new page?
2. Assuming that we are now using the page we selected from the previous question and no other pages have been marked as "used" since then, what is the next page we will select?
3. Again, assuming we are using the pages selected in the previous two questions and that no pages have been marked as used since then, which page is selected?

Implementation

You need to go through the reverse lookup table (aka the frame table) an find any free frames. (**HINT:** A frame that is invalid is free). If you do not find any free frames, then you must find the **Least Frequently Used** frame, and evict the page it holds. Please be careful when evicting a page. You will need to make sure that you update the page table of the victim frame to reflect that the mapping it holds is no longer valid, and you will also need to increment statistics for write backs if the page you are evicting is dirty. Use the `rlt` variable for this purpose. You can find more information about the `rlt` by looking at the `reverselookup.h`

source file in the `simulator-src` folder. **Make sure to return the page frame number that the page fault handler should use to resolve the page fault at the end of this function**.

Note: Since we are simulating accesses and there is no data involved, we do not understand the significance of writing data back to disk. In reality, not having enough physical memory can result in many page faults, which in turn can result in multiple writes to disk. Writing to disk is much slower and will slow down your system a lot.

Step 4 - Handling Page Faults

Now that our simulator has the ability to find an efficient replacement frame, let us examine how a page fault is actually resolved. You must update the mappings from VPN to PFN in the page table as well as the mappings in the reverse lookup table in order to resolve the page fault. Before moving on to the implementation section, consider answering the below questions.

Conceptual Questions

1. If a page fault occurs and there is an invalid frame in the physical memory, should you choose to evict a frame? When would such a situation occur?
2. What is thrashing? How would a virtual memory system try to subvert this phenomenon?

Implementation

Look in the `page_fault.c` file to view the partially implemented page fault handler. You must update the reverse lookup table to reflect that the frame you chose in the `find_free_frame` function is valid. You must also update the reverse mapping to VPN in the `rlt`, and finally set the `rlt` to point to the page faulting process. This can be done by making the rlt's `task_struct` field equal to the `current_process`. Ensure that you mark the page as dirty in the page table (if it is a write read), which can be accessed using **current_pagetable**. **Make sure to return the page frame number that in which you have put this new page**.

Step 5 - Adding a TLB

By now, you might have noticed that accessing memory is very slow. Virtual memory does not help to curtail this problem, and instead adds to it! In our implementation, we must go to memory twice - once to translate the virtual address to the physical address and again to access the memory at the translated location. Virtual memory may even cause us to load in a page from the disk! These costs are unacceptable for smaller programs that do not take advantage of the virtual memory system.

We cannot eliminate the actual memory access, but we can diminish the page table lookup by adding a small piece of hardware that keeps a small buffer of past VPN to PFN translations. If we locate a VPN in this buffer, we can bypass the page table lookup in memory entirely - a great boon! We call this buffer the Translation Lookaside Buffer. It will provide us an alternative means of performing the lookup during VPN to PFN translation.

The TLB uses an interesting eviction algorithm, called the 'clock sweep algorithm'. The idea here is that you mark an entry as "used" whenever it is accessed or used. When finding an entry to evict, you first go through ('sweep') all the entries and set the used ones as unused (i.e. If an entry has its used bit set to 1, you set it to 0), and return the first entry you encounter which is already unused (i.e. which already had its used bit set to 0). Should you reach the end of the list (Array in this case), you wrap around and start from the top. If all the entries were used (i.e. All of them had the used bit set to 1), you return the first entry in

the list that was checked (In your case this will always be entry number 0).

Note: Usually the clock sweep algorithm remembers the last position it left off on, however you **DO NOT** have to implement that feature.

Fun Fact: Linux developers sometimes refer to the Clock-Sweep Algorithm as the "Second Chance Algorithm" because it gives each page that has been used recently a second chance at not being evicted.

Conceptual Questions

The structure of the TLB is remarkably similar to that of the page table. The biggest difference is the fact that unlike in the page table, where we use the VPN as an index to an array, the VPN is simply another entry in the TLB. Note that the TLB is relatively small, and cannot hold many entries. Use the TLB shown below to answer the questions. This TLB is only capable of holding four entries. Assume that any entry not explicitly present in the page table is **invalid**.

Table 3: TLB

VPN	PFN	Valid	Dirty	Used
0xDE	0xBE	YES	NO	YES
0xF0	0xD0	YES	YES	YES
0x42	0xAB	YES	NO	YES
0x21	0x23	YES	NO	NO

1. What address does the virtual address 0xDEAD translate to? Is this translation found in the TLB or in the Page Table?
2. What address does the virtual address 0xBE21 translate to? Is this translation found in the TLB or the page table?
3. When we lookup the address 0xBC87, we miss the TLB. This requires us to evict an entry from the TLB. Which entry would we pick to evict, assuming we use a standard Clock-Sweep Algorithm?

Implementation

Open the `tlb_lookup.c` file and follow the comments described there. You may want to note that you have access to the TLB entries via the `tlb` array, which is an array of type `tlbe_t` structs and whose length is 2^{tlb_size} entries. Keep in mind that since there is **no relationship** between the index in the TLB and the content stored there, you will have to check **every valid entry** in the TLB before deciding that you are unable to find an entry. The structure of each entry is given below:

```
typedef struct {
    uint64_t vpn; /* Virtual Page Number */
    uint64_t pfn; /* Physical Frame Number */
    uint8_t valid; /* Valid 'bit' */
    uint8_t dirty; /* Dirty 'bit' */
    uint8_t used; /* Used (recently accessed) 'bit' */
} tlbe_t;
```

If you find a valid mapping from VPN to PFN in the TLB, you must set the used bit of the TLB to **1**. You must also update the frequency count of the corresponding page table entry. I understand that this is not efficient, however since this is a simulation, this must be done. And then finally, return the **Entire physical address** (that means the PFN and the offset concatenated together).

If you miss in the TLB, then the `page_lookup` function is called to get the VPN to PFN mapping. You must also update the TLB. First try finding an unused TLB entry. If you find one, you can just update the contents (valid, vpn, pfn, dirty and used bits) of that location, and you are done. However if you do not find an unused entry, run the clock sweep algorithm described above and choose a victim. Make sure to mark the victim vpn dirty in the current pagetable if the victim tlb entry was dirty.

Note: This function returns the entire physical address, not just the page frame number.

Step 6 - Computing the EMAT

In the final section of this project, you will be computing some statistics. Below are the entries in the `stats_t` struct:

1. accesses - The total number of accesses to the memory system
2. reads - The total number of accesses that were reads (type == READ)
3. writes - The total number of accesses that were writes (type == WRITE)
4. translation_faults - Accesses that resulted in a TLB miss
5. page_faults - Accesses that resulted in a page fault
6. writes_to_disk - How many times did you have to write data to disk
7. reads_from_disk - Number of times you had to read data from disk
8. AAT - The average access time of the memory system
9. TLB_READ_TIME - The time taken to access the TLB (Set by simulator)
10. MEMORY_READ_TIME - The time taken to access memory (Set by simulator)
11. DISK_READ_TIME - The time taken to read a page from disk (Set by simulator)
12. DISK_WRITE_TIME - The time taken to write a page to disk (Set by simulator)

Implementation

Complete the `compute_stats.c` file, and complete the function given there. You will need to think how often you access each of memory, disk and the tlb, and then come up with a formula for the average access time. This has been left as an exercise for you.

HOW TO RUN YOUR CODE

We have provided you a 'Makefile' that you can use to compile your code. **Modify the Makefile at your own risk**

Use these commands to run the simulator driver:

```
$:  make
$:  ./vm-sim < <input trace name> to run a trace program from the traces directory.
```

To see the simulator options, run:

```
$:  ./vm-sim -h
```

To run the script that will run all the tests, do the following:

```
$:  bash run_script.sh
```

Finally: To pipe the output for all the test cases to a text file, do the following:

```
$:  bash run_script.sh >> <Filename>
```

To **verify** that you are matching the TA generated output, you can use the diff command. Here is an example:

```
$:  diff <Filename> solution.log
```

NOTE: TO GET FULL CREDIT YOU MUST COMPLETELY MATCH THE TA GENERATED OUTPUTS. THESE WILL BE MADE AVAILABLE ON T-SQUARE VERY SHORTLY.

WHAT TO SUBMIT

To generate a submittable version of your project, type in `make submit` into the terminal. This will generate a tarball of your source code.

You need to submit the following files:

- The tarball you just generated

Don't forget to sign up for a demo. We will announce when these are available. Failure to demo will result in a score of 0.