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 provided comments to guide you along. 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. Split the Virtual Address
2. Translating the Virtual Address into the Physical Address
3. Handling Page Faults
4. Handling Page Replacement
5. Adding a TLB to Improve Translation Speed
6. Computing the Estimated Memory Access Time (EMAT)

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 in the book and then answer the questions.

Once you are able to answer questions about a given concept, we ask you to implement that concept in code. For each section, the given concept is described briefly, and there will be FIXME sections labeled in the code with greater elaboration as needed. The files you have to fill out for this assignment are located in the `student-src` folder. The simulator code can be found in the `simulator-src` folder. You should not need to look at any of the simulator code. However, please see Appendix A - The Simulator for more details about what is in the simulator. It may be useful if you are having trouble figuring out what to do.

Compiling and Running

To recompile the simulator, we have provided you with a Makefile in the top-level directory. You can type in the command `make` in the terminal to build the project. When the project is finished building, there will be an executable in the top level directory called `vm-sim`. To execute this file, you must provide it with a references file. Reference files are found in `references` folder, and it provides four types of reference files:

- **basic** - A basic reference file that yields no page faults **due to eviction**.
- **eviction** - A reference file that should produce a large number of page faults.
- **tlb** - A reference file that should cause a large number of TLB hits.
- **everything** - A reference file that provides a bit of everything described above.

For example, if we wanted to run the simulator with the basic file, we would do the following:

```
$:  ./vm-sim references/basic
```

There are several other command line options for the simulator that you can play around with, such as changing the memory size, page size, and the tlb size. The default settings are memory size = 16 values, page size = 2 values, and tlb size = 4. You can change these parameters to see how they affect the memory access time. To view these options, type:

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

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 a virtual address into a physical address, which can be used to address into physical memory. The first step of this process is to translate the virtual address into a physical address.

Note that a virtual address consists of two parts: the higher order bits form the Virtual Page Number (VPN), and the lower order bits form the offset.

Conceptual Questions

Consider a machine in which virtual addresses are made up of 16 bits. Each page on this machine has 2^8 addressable locations. Now, answer the following questions:

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 virtual address 0xDEAD?
4. What are the VPN and offset of the virtual address 0xBEEF?

Implementation

Look at the file `page-splitting.h`. You should find two macros that are used by the simulator to split the address. They are called VADDR_PAGENUM and VADDR_OFFSET. These macros take a virtual address and return the page number or offset, respectively. Fix these so that they return the proper VPN and offset. Here's a hint: Use the global variable `page_size` to access the size of a page.

Step 2 - Address Translation

Now that we can split the address, we can transform the virtual address into a physical address. This requires a unique page table per process to store the mapping between virtual page numbers and physical frame numbers. In the simulator, a page table is represented as an array filled with the following structure:

```
typedef struct {
    pfn_t pfn; /* Physical Frame Number */
    unsigned char valid; /* Valid 'bit' */
    unsigned char dirty; /* Dirty 'bit' */
    unsigned char used; /* Used 'bit; */
} 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. In addition, there is a used bit. This is not the same as the valid bit; its purpose will become apparent in Step 4.

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	Used
0xDE	0xBE	YES	NO	NO
0xF0	0xD0	YES	YES	YES
0xBE	0x42	YES	NO	NO
0x42	0x43	NO	NO	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?

Implementation

After finishing the questions in the previous section, you should have a pretty good idea about 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 modify the function `pagetable_lookup` function. Make sure that you are checking if the page table entry is valid! If the page table entry is not valid, then you must increment the global variable `count_pagefaults` and call the `pagefault_handler` function. Keep in mind that the global variable `current_pagetable` is a pointer to an array of `pte_t` entries, which can be indexed by the VPN. Here is a hint:

- 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 - Handling Page Faults

When the CPU encounters an invalid page in the page table, the OS allocates a page frame for the requested page by either locating a free frame or by evicting a physical frame that is already in use. After this occurs, the OS updates the page table to reflect the new mapping, and then continues with the memory access.
However, what causes page faults to occur? When a program is initially started, none of the virtual addresses are valid. When a given address is used for the first time, the OS will allocate a physical frame to store the appropriate information. After this occurs for a while, the OS might run out of physical frames to allocate! In this situation, the page fault handler will have to evict a physical frame. When it does this, it moves the information stored in the frame to the hard disk and allocates the evicted page to the new process.

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 `page-fault.c` to view the partially implemented page fault handler. Follow the comments detailed in the source file. While working on this part, keep in mind that each process has its own page table. The data structure `current_pagetable` refers to the page table of the currently running process (which is the one that is requesting a new page). The page table of the process that owns the victim page can be found in

the victim process control block `victim_pcb`. Think about why it is important to have access to the victim process' page table when evicting one of its pages!

Step 4 - Improving Page Replacement

Up to this point we have not concerned ourselves with what happens when we have to make more pages than we have room for in our physical memory. In reality, the virtual memory system uses the hard drive to make it seem like there is more memory than there really is. Indeed, it is quite slow to load processes in and out of the disk. However, if our page replacement algorithm is performed in an intelligent manner, it will be much more effective than to simply stop the user's process during execution.

The virtual memory system you have constructed so far uses a randomized page replacement algorithm that we have provided. You will be implementing a more intelligent system. We will ask you to implement a page replacement algorithm and observe its effects on the average access time.

The optimal page replacement algorithm is one that can tell the future! Out of all the physical frames in memory, our algorithm should evict the page that will not be used for the longest time from when the page fault happens. Unfortunately, there is no "magical soothsayer" that will allow us to gleam into what is to pass. However, you will be implementing the Clock-Sweep Algorithm, which takes advantage of temporal locality in that it assumes that pages that were recently accessed will be used again in the near future. While this algorithm is not optimal, it is relatively easy to implement and achieves decent performance in practice.

The basic idea of the Clock-Sweep Algorithm is that you first mark a page as "used" whenever it is accessed. When you need to evict a page, you will iterate (Sweep) through the page table, examining each frame's "used" bit. If a page is marked "used", you mark it as "unused". When you encounter a page that is not "used" you will select this page to evict. Should you ever reach the end of memory, you should wrap around and start over at the beginning of the page table array. This means that if all pages are marked as "used", the algorithm will unmark all the pages and choose the first page you examined when you iterated through the page table array. Normally, the clock-sweep algorithm remembers the page it left off on, so that the next time it executes, it continues on from there. Note that for this assignment, you *do not* have to implement this feature. If you do, you get brownie (read: not extra credit) points.

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

Answer the following questions about page replacement, given the following page table. Assume that any entry not listed is **VALID** and **USED**.

Table 2: Page Table

VPN	PFN	Valid	Dirty	Used
0xDE	0xBE	YES	NO	YES
0xF0	0xD0	YES	YES	YES
0xBE	0x42	YES	NO	NO
0xBC	0x43	YES	NO	NO
0x42	0xAB	YES	NO	YES
0x21	0x23	NO	NO	NO
—	—	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

Now that you understand how page replacement is supposed to work, open `page-replacement.c` and change the replacement algorithm to more intelligently evict pages. Your code should be doing two things: first, check to see if there is an uninitialized frame. If so, just use that one (already implemented)! If there are no invalid pages, then perform a Clock-Sweep Algorithm to decide which page you should evict.

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.

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	Last Access
0xDE	0xBE	YES	NO	32
0xF0	0xD0	YES	YES	41
0x42	0xAB	YES	NO	27
0x21	0x23	YES	NO	11

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 LRU Algorithm? What if we used FIFO?

Implementation

Open the `tlb-lookup.c` file and follow the comments describing what needs to be fixed. 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 `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 {
    vpn_t vpn; /* Virtual Page Number */
    pfn_t pfn; /* Physical Frame Number */
    uint8_t valid; /* Valid 'bit' */
    uint8_t dirty; /* Dirty 'bit' */
    uint64_t access; /* Access Timestamp */
} tlbe_t;
```

In your code, you will implement two functions. Our simulator will support two eviction policies in the TLB: FIFO and LRU. Only one of them will be called by the simulator during any given simulation run. In `tlb_lookup_lru()`, you will implement a TLB that uses the LRU eviction scheme. In `tlb_lookup_fifo()`, you will implement a TLB that uses the FIFO eviction scheme. Both modes' code will look nearly identical, but there will be a slight difference in when the access timestamp is changed. You may use the global variable `current_timestamp` in your functions, and you may assume it is accurate. The simulator will automatically call the appropriate `tlb_lookup` function depending on the command line inputs.

To run the simulator using a specific replacement policy for the TLB, run `./vm-sim -e FIFO` for FIFO or `./vm-sim -e LRU` for LRU. The default if none is specified is LRU.

Step 6 - Computing the EMAT

Now that we are getting some results from the simulator, we are ready to analyze the output. In this part, you will complete a function that calculates the Estimated Memory Access Time, or EMAT. This is computed by calculating the average amount of time it takes for a memory access. Keep in mind that this metric is influenced by what memory access patterns are processed by your code.

Conceptual Questions

1. Assuming a memory access time of 100 ns, an average disk access time of 10 ms, how long would it take to access memory ten times if two of those accesses resulted in page faults (these require disk access to resolve!) and four of the accesses resulted in TLB misses? Remember that accessing the page table requires two memory accesses.
2. While accounting for two memory access times for a TLB miss, one memory access for a TLB hit, and two memory accesses and a disk access for a page fault, what would be the generalized formula for the average time taken per memory access?

Implementation

Calculating the EMAT by hand every time you run the simulator is quite tedious. Since it is generally better to make the computer do these kinds of computations for us, this step of the project asks you to fix the function `compute_emat` found in `emat.c`. This function takes no parameters, but it has access to the global statistics maintained in `statistics.h`. Namely, it has access to:

- `count_pagefaults` - The number of page faults that occured.
- `count_tlbhits` - The number of TLB hits that occured.

- `count_writes` - The number of stores and writes that occured.
- `count_reads` - The number of loads and reads that occured.

In your computation, you should utilize the constant values `DISK_ACCESS_TIME` and `MEMORY_ACCESS_TIME` which are defined in `emat.c`. For the purposes of this function, treat a TLB hit as taking no time when looking up the VPN in the page table.

Style Checking

We will be enforcing a code styling convention on any of the C code you write. Just because we are not writing Java code anymore does not mean we should forego good style! Having well formatted code makes your code more readable, maintainable, and debuggable.

To enforce this, we will be using two Python scripts that will statically analyze your source code. Passing these tests will be worth **5 points each**. These scripts are adapted from open-source projects and are not perfect, but they will catch a good number of styling errors. If you believe that either script is incorrectly marking your code, please let one of your TAs know. These scripts are adapted from these two open-source projects from Google and Caltech:

1. https://github.com/google/styleguide/tree/gh-pages/cpplint
2. http://courses.cms.caltech.edu/cs11/material/c/mike/misc/c_style_guide.html

Important: Do not use these!

We have created adapted versions of these scripts for you to use. If you try to use the scripts straight from these websites, you will notice strange errors. Please only use the style-checker scripts we have provided you in this assignment. Note that you will need **Python 2** to run these. We will be running the scripts the same way as shown below to test your code.

(5 points) - CPPLint

To use `cpplint.py`, run the command:

```
$:  python cpplint.py --linelength=120 student-src/*.c
```

The program will output "Total errors found: 0" if you have passed its test.

(5 points) - C Style Check

To use `c_style_check.py`, run the command:

```
$:  python c_style_check.py student-src/*.c
```

The program will output nothing if there are no errors.

Deliverables

When you are done with the project, move to the top level directory of your project and run the following:

```
$:  make submit
```

This will generate a file called `p3-submit.tar.gz` that contains everything you need to submit for this assignment. Please remember to submit this file on T-Square!

Don't forget to sign up for a demo slot! We will announce when these are available. Failure to demo results in a 0!

Precaution: You should always re-download your assignment from T-Square after submitting to ensure that all necessary files were properly uploaded.

Appendix A - The Simulator

This section is meant to serve as a reference to the simulator code. Use it if you are interested in knowing about how the simulator works, or you need to figure out how to use something.

Simulator Files

Simulator files are located in the `simulator-src` directory. You should not have to modify any of these files. If you need to look at any of the simulator code, you should really only look at the header files. To help you understand how the simulator is organised, examine the following list:

- `global.h` - Provides a definition of globally accessible values.
- `memory.h` - Simulates the physical memory found in a computer.
- `pagetable.h` - Provides an implementation of a page table. You will probably be most interested in `pagetable_lookup`, which you have to write, and `get_free_frame`, which will help you implement the page fault handler. The struct `pte_t` represents a single page table entry. If you need to get the page table of the currently executing process, the variable `current_pagetable` will point to the appropriate array.
- `process.h` - Provides an implementation for a Process Control Block, as well as various methods for manipualting processes. You probably won't need anything in the PCB.
- `sim.c` - This file includes the main function for the simulator and serves as an entry point into the rest of the code.
- `statistics.h` - This file is used to gather and display statistics.
- `swapfile.h` - Simulates moving and loading pages to and from the disk
- `tlb.h` - Provides an implementation of the TLB.
- `types.h` - Provides useful typedefs that allows us to refer to more descriptive types such as `vpn_t` instead of just `int`.
- `useful.h` - Provides useful macros.