Project 4 - Virtual Memory Management

Introduction

In this project, you will implement a virtual memory system using the simulator provided. The project will walk you through the implementation step-by-step, asking questions about each step to make sure you understand what you have to do before you go off and write code.

The Assignment

This assignment shouldn't be extremely difficult. If you find yourself struggling with the code, step back and think about what you are supposed to be doing in theory. If you don't understand how the step should work, it will be impossible to write it out in code.

The assignment is organized into 5 steps, as follows:

- Step 1 - Split the Address
- Step 2 - Address Translation
- Step 3 - Computing the EMAT (estimated memory access time)
- Step 4 - Handling Page Faults
- Step 5 – Adding a TLB
- Step 6 - Improving Page Replacement (bonus)

Each step starts with some theoretical questions. **You do not have to submit answers to these questions. They are there to guide you. If you can answer the questions without much trouble, you will probably have an easy time with this assignment.** As was mentioned earlier, the code will be really hard to write if you aren't comfortable with the concepts first. The questions are designed to prepare you for the coding in each step, so do them first.

The Simulator

Code that you have to complete for this assignment can be found in the files in the directory `student-src`, while the simulator code can be found in the directory `simulator-src`.

To recompile the simulator, we have provided a `Makefile` in the top level directory. This can be used to build the project by simply typing `make`. After you have built the program, you should find an executable named `vm-sim` in the

top level directory. To run this program, you **must** specify a references file. The references file describes a series of memory references. The directory `references` contains four different potential references files for your use:

- `basic` - A basic references that yields no page faults due to eviction (**note:** some page faults are *always* necessary).
- `eviction` - A reference file that should produce a large number of page faults.
- `everything` - A reference file that produces a little bit of everything.
- `tlb` - A reference file that should cause quite a few TLB hits.

To run the simulator with a given references file, for example the `basic` file, issue the following command:

```
./vm-sim references/basic.txt
```

There are several other command line options for the simulator that you can play around with to adjust the memory size, page size and tlb size. You don't need to use them, but you can play around with different settings to see the effect that they have on the memory access time. The default settings, are a memory size of 16 values, a page size of 2 values, and a tlb size of 4.

Understanding the current code

The first thing you should do is to spend some time looking at the current simulator code (open and look through all files in simulator-src folder) in order to understand what it does. You can compile and run it as is, you will see that any memory loads fail a check to see if they load the correct value, eventually you will fix that. For now, you should be able to answer the following questions

- page, memory, and tlb size are all global variables that can be accessed from any file (including the ones you will work with in student-src). Where are they defined and what are their default sizes?
- How big is a word in memory? How many bits are in a memory address?
- memory.c contains what is basically the MMU, what is the sequence of events that occurs on a load or store?
- process.c is basically the OS (note the name is misleading, it's not just a single process, but the manager of all processes). What is stored in the PCB? In an entry of the reverse lookup table? Where does a process'

page table actually get allocated? What happens to the TLB on a context switch?
- What are the fields in an entry of the per-process page table? How do you access the page table for the process running now?
- How does the simulator keep track of pages that have been swapped out? Does it actually write/read them from disk? What happens if you have a page fault and the page doesn't exist yet (e.g. when you load a program for the first time)?
- What are the fields in an entry in the TLB?

Step 1 - Split the Address

In modern virtual memory systems, the program written by the user accesses memory using virtual addresses. The hardware and operating system work together to translate these addresses into a physical address which can be used to access physical memory. The first step of this process is to take the virtual address and convert it into a physical address.

Remember that a virtual address consists of two parts. The high-order bits make up the *virtual page number* (VPN), and the low-order bits make up the *offset*.

Part 1a - Some Questions About Address-Splitting

On a certain machine, virtual addresses are made up of 16 bits. Each page on this machine has 2^8 addressable locations. Answer the following questions:

Question - How many bits must the offset be?

Question - Recalling that the virtual address is split into the offset and VPN, how many bits is the VPN?

Question - What is the VPN given the address 0xBEEF?

Question - What is the offset given the address 0xBEEF?

Correcting the Address-Splitting macro

Look at file `page-splitting.h`. You should find two macros that are used by the simulator to split the address. They are named `VADDR_PAGENUM` and `VADDR_OFFSET`. They take a virtual address and return the page number or offset, respectively.

They currently do not function correctly. Fix them so they properly return the virtual page number and offset.

[Hint 1: Use the global variable `page_size` to access the size of a page]

[Hint 2: Think about using modulus and integer division. It should be possible to implement each macro by replacing the current "0" with one very short equation.]

Step 2 - Address Translation

Now that we can split the address, we are ready to translate the virtual address into a physical address. This requires a page table to store the mapping between virtual addresses and physical addresses. In the simulator there is a page table for each process consisting of an array of pte_t structs (page table entry type). You'll notice, that the VPN doesn't appear as part of the page table. This is because the page table is an array of these entries. The index into the array corresponds to the VPN. This allows the mapping from VPN to page table entry to be performed very easily.

Implementing Address Translation

Open up the file `page-lookup.c`. In this file you will find a function called `pagetable_lookup`. Modify this function to behave correctly. Note that you will have to check to make sure the entry is valid. If it isn't, you should increment the variable `count_pagefaults` and call the function `pagefault_handler`. You do not need to complete the implementation of that handler function in order to test this step (it should work, just not give you a correct frame number if there is a page fault yet).

When implementing address translation, keep in mind that the global variable `current_pagetable` is a pointer to an array of page table entries (`pte_t`), indexed by VPN.

You can check the results of this step by running the simulator again and looking at the values of VPN, FPN, and offset for each memory access.

Step 3 - Handling Page Faults

What happens when the CPU encounters an invalid address in the page table? When this occurs, the OS should allocate a physical frame for the requested

page, by either locating a free frame, or evicting a physical frame that is already in use. After this occurs, the OS should update the page table to reflect the new mapping, and then continue with the memory access.

What exactly can cause this? Well, when a program is initially started, none of the virtual addresses are valid. When a given address is first used, the OS will allocate a physical frame to store the information. If this keeps occurring, the OS might be trying to allocate a new physical frame, when there is no memory remaining. In this situation, the page fault handler will have to evict a physical frame. When it does this, it moves the information stored there to the hard disk, and then uses the recently cleared frame.

Implementing the Page Fault Handler

Look in the file `page-fault.c`, and you will find a partially implemented page-fault handler. The `FIX ME` comments in there walk you through the changes you should have to make. Each of these will require **a few** lines of code, no more.

While working on this, keep in mind that each process has a page table. `current_pagetable` refers to the pagetable of the currently running process (which is the one that needs 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 what all needs updated in each of these tables!

After completing this step you should be able to run the program and get correct values when you load memory, no more errors!

Step 4 - Computing the EMAT

Now that we are getting some results from our simulator, we are ready to perform some real computations with our results. EMAT stands for Average Memory Access Time. It is computed (quite simply) by figuring out the amount of time for each access and dividing the total access time by the number of accesses performed.

Questions about EMAT

Question - Assuming a memory access time of 100ns, and that an average disk access takes 10ms, how long would it take to access memory 10 times, if 2 of

those accesses resulted in page faults, and 4 of the accesses resulted in TLB misses?

[Hint: Don't forget to take into account the time it takes to access the page table.]

Question - What would a general formula for the EMAT be, in terms of average disk access time and memory access time?

Automate EMAT Computation

This step of the project asks you to fix the 2 `compute_emat` functions found in `emat.c`. This function takes no parameters, but it has access to the global statistics maintained by in `statistics.h`, specifically:

- `count_pagefaults` - The number of page faults that occurred. This number ***includes*** the unavoidable initial page faults.
- `count_tlbhits` - The number of TLB hits that occurred.
- `count_writes` - The number of stores/writes that occurred.
- `count_reads` - The number of loads/reads that occurred.
- `count_diskaccesses` - The number of disk accesses that occurred. This number ***does not include*** the unavoidable initial initial disk accesses.

Since we haven't yet implemented the TLB, there will be no TLB hits. After the TLB has been correctly implemented, make sure that your implementations still work correctly.

Your computation should use 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.

Step 5 - Adding a TLB

Accessing memory is slow. Virtual memory doesn't really help this, because page tables mean that we will have to access memory twice -- once to translate the virtual address to the physical address, and again to actually access the correct location. As useful as virtual memory is, if it caused every memory access to take twice as long (or longer, if the hard drive came into play), it would be an unacceptable cost for smaller programs that didn't need virtual memory. Luckily, there are ways of reducing the performance hit.

We obviously can't eliminate the actual memory access, but we can attack the page table lookup by adding a small piece of hardware that keeps a small buffer of past translations. If we can locate the virtual address in this buffer, we can bypass the page table lookup. This buffer is called the Translation Lookaside Buffer (TLB) because it provides an alternative means of performing the lookup for translation

Questions about the TLB

The structure of the TLB is remarkably similar to the page table. The biggest difference is the fact that in the page table, the VPN serves as an index into an array. In the TLB, the VPN is simply another entry in the TLB. This is because, the TLB is relatively small, and can't store every single entry. Use the TLB provided below to answer the questions. The TLB is only capable of holding four entries. Any entry not explicitly present in the page table is assumed to be invalid.

Page Table

VPN	PFN	valid	dirty	used
0xDE	0xBE	YES	NO	NO
0xF0	0xD0	YES	YES	YES
0xBE	0x42	YES	NO	NO
0xBC	0x43	YES	NO	NO
0x42	0xAB	YES	NO	NO
0x21	0x23	YES	NO	NO
...	----	NO	NO	NO

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

Question - What address does the virtual address 0xDEAD translate to? Is this translation found in the TLB or the page table?

Question - What address does the virtual address 0xBE21 translate to? Is this translation found in the TLB or the page table?

Question - When we lookup the address 0xBC87, we miss the TLB. This requires us to evict an entry form the TLB. Which entry would we pick to evict, assuming we use a standard clock-sweep algorithm?

Adding a TLB

Open up the file `tlb-lookup.c`. You will find a partially implemented function, with comments describing what you need to change. The code for the TLB can be found in the file `tlb.c`. Of special interest are the structure for TLB entries, and the pointer `tlb` which points to an array of TLB entries. 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 were unable to find an entry.

Pay attention to the comments, as they describe all of the different things you must do. When scanning through the array of TLB entries, it might be useful to know that the array has `tlb_size` entries in it.

Step 6 - Improving Page Replacement

Up to this point we haven't really worried about what happens when we have to make more pages than we have room to store in physical memory. In reality, the virtual memory system uses the hard drive to make it seem like there is considerably more memory than there really is. While it is really slow to use the hard drive in this manner, if page replacement is performed in an intelligent manner it is much better than just stopping the user's program.

Right now, the virtual memory system you have built uses a random page replacement algorithm that we provided for you. As was discussed above, an intelligent system would be much better than this. After asking some questions about page replacement, we will ask you to implement one of the algorithms we have discussed in class, and observe the impact that this can have upon EMAT.

The optimal page replacement algorithm would be to look into the future. Of all the physical frames, we should pick the one that is first used the longest time from now. We know that we will not have to evict a page again until that page is accessed. Unfortunately, this algorithm requires me to be able to look into the future, which isn't something that a modern computer is capable of doing. Instead, we take advantage of temporal locality to justify the claim that if a

page was used recently, it is likely to be used again in the very near future. While not optimal, the clock-sweep algorithm is a very reasonable algorithm, because it is easy to implement and has relatively decent results in practical use. For this project, you will be implementing the clock-sweep algorithm.

The basic idea of clock-sweep is that you mark each page when it is used. When you need to evict a page, you iterate (sweep) through memory examining each frame's marked bit. If the page is marked, you unmark the bit. When you encounter a page that isn't marked, that is the page you will evict. If you reach the end of memory the search wraps around (this means that if all pages are marked, we will unmark all the pages and then choose the page we started with for eviction). In Linux, the clock-sweep algorithm is sometimes also referred to as the second chance algorithm because it gives each page that has been marked as recently used a second chance at not-being evicted.

Questions About Page Replacement

Answer the following questions about page replacement, given the following page table. Unlike in the previous question (or the question to come up soon), assume that any entry not listed is VALID and USED.

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

Question 12 - What is the first page that should be used when we need to allocate a new page?

Question 13 - Assuming that we are now using the page we selected in question 12 and no pages have been marked as used since then, what is the next page we will select?

Question 14 - Again, assuming that we are using the pages selected in questions 12 and 13 and no pages have been marked as used since then, which page is selected?

Implementing a Page Replacement Policy

Now that you understand how page replacement works, open up the file `page-replacement.c` and change the replacement algorithm to be more intelligent. You should do two things. If there is an invalid page, simply use that one. If there are no invalid pages, perform a clock-sweep algorithm to decide which page you should evict.