18-344 Lab 3

24 October, 2024

Introduction

In this lab you will implement parts of a software virtual memory system and understand the role of hardware in optimizing virtual memory implementation.

The purpose of this lab is help you understand how virtual memory works, including memory mapping, page fault handling, and Translation Lookaside Buffers (TLBs), but not including page allocation and replacement.

Your first task in this lab is to implement a page table and then to use it for mapping and memory accesses. Your implementation should count how many memory accesses and how many page faults happen during an execution with a page table of a particular size.

Your second task in this lab is to implement a TLB to cache translations and avoid the need to access the page table. Your TLB's size and associativity is up to you and you need to justify your design choice and argue for its implementability in your write-up for this lab.

Your page table implementation should work approximately like the intel Core i7 processor. The Core i7 has a 4 level page table hierarchy with 512 entries per page table.

You can and we will test your implementation using a driver program like the one that we provide in `vm_test.cpp`. This test driver sets up and initializes a virtual memory system and then runs a series of memory mapping operations and memory accesses. Throughout the end of the execution, your virtual memory implementation should track the number of accesses, page faults, and TLB hits. At the end of the execution, the driver will report the number of each of these.

Optional: You may choose to additionally implement a flat/linear hashing implementation of a page table that uses chaining to resolve hashing conflicts, instead of using a hierarchical page table, like we looked at in class. This implementation variant will allow you to model and measure how the cost of linear hashing differs from the cost of a hierarchical page table.

Starter Code

Your implementation should be built around the existing base code that we provide with the lab. You are free to extend these files with anything you need to add, but you should not substantially change `vm-provided.cpp`, nor `vm-util.h`, nor `pte-util.h`. If you must change these files, please carefully document the required changes in your write-up and justify why the interfaces had to change.

`vm-student.h/.cpp`, `vm-util.h`

The top-level VM implementation. `VM *vm` is the pointer to the VM implementation that the driver code will use to manipulate your VM implementation. Here, you must implement `vmPageFaultHandler(pte)`, `vmMap(addr)`, and `vmTranslate(addr)`. These functions are documented in `vm-student.cpp`. `vmMap()` will be called directly in the driver programs. `vmTranslate()` will be called in `VM::Load()/Store()` on each memory access. You must call `vmPageFaultHandler()` on page faults that you encounter during translation. You must handle a page fault by attempting to allocate a new page (if there are physical pages to allocate) or by replacing a page, if not. To allocate a new page, you should use `VM::bumpAllocate()`. To replace a page, use `VM::replacePage()`. Both of these functions are given in `vm-provided.cpp`. Your VM implementation should update `_page_faults`, `_num_accesses`, and `_tlb_hits` when each of these events happen during an execution.

`ptab.h/.cpp`

The page table interface. You should implement `getEntry(addr,level)` and `createEntry(addr,level)`. These functions get or create an entry in the page table at the specified level based on the address provided. Recall: depending on the level, a different subset of bits from the address determine that address's entry in the page table. `getEntryDirect(index)` is an alternative interface that allows directly indexing into a page table to access the entry at index. `getEntryIdFromAddr(addr,level)` is a convenience function you will probably want to implement that computes the entry for an address in a page table at the given level. An entry in a page table is of type PTE, which is a union that you can interpret as either a pointer to another page table (`pageTable *`) or a pointer to a PTE (`pageTableEntry *`). You may want to think through the idiom that you use to traverse through the levels in your page table hierarchy, computing the index based on the address and level. The table pointer in this module is the storage for the table's entries and should be allocated when the object gets constructed.

`pte.h/.cpp`, `pte-util.h`

A page table entry. A page table entry holds the physical page number of a page. Do not confuse this structure with a PTE, which is a related union, defined in `pte-util.h`.

`tlb.h/.cpp`

The TLB implementation. The TLB should cache previous translations from virtual to physical addresses. On each access your code should `lookup(addr, &PPN)` the address in the TLB and populate the PPN reference parameter if a valid mapping is cached. `lookup()` returns true if there was a valid mapping cached. `update(addr,new_PPN)` updates the TLB's cache state with the new translation from virtual address addr to the new physical page number `new_PPN`.

What to Turn In

You should turn in your code and a writeup describing your implementation and its behavior. Your writeup should describe how your page table and your TLB works and should include a quantitative evaluation. Your evaluation should include a quantitative argument justifying your TLB's organization and should specifically argue its implementability and cost-effectiveness. Planning and executing an interesting and informative quantitative evaluation is part of the work of this lab. You should design experiments and tests that demonstrate features and interesting behavior in your design.

A few things to keep in mind:

- Your code should work on a variety of traces that include page faults, access violations (e.g., segfaults), and normal accesses. It is *your* job to create interesting traces that test interesting cases in the operation of your paging system. We will run your system on your traces, which you should turn in, and we may run your system on our traces, which will test interesting behavior.

- You should not modify `vm-provided.cpp`, and we will use our own version of that code during grading.

- As part of your quantitative justification for your TLB organization, you should include a plot showing how the frequency of page table walks (i.e., TLB misses) changes with different TLB configurations (the subset of the design space that you traverse for this study is up to you). Describe the scale of the improvement in your system's performance with reference to the benefits of your TLB: how many fewer memory accesses does your system have to make in order to do virtual memory translations with a TLB vs. without a TLB?

- You should include an overall performance summary based on the total number of program memory accesses, the number of memory accesses per translation (without a TLB), and the number of TLB hits in your system. The specific details of this performance summary are up to you, but the point is to show that you can summarize the behavior of a system over a design space in terms of several relevant and meaningful figures of merit.
- If you choose to implement a linear hashing page table, you should show off your work by comparing its cost to the cost of the hierarchical version. Are there cases where one or the other is better?

Errata

- There once was a bug that allowed `VM::replacePage()` to return a PPN that is `0xDEADBEEF`, AKA `VM_PAGEDOUT`. We believe this bug is removed, but if you encounter this condition please report it to the course staff. A workaround is to repeatedly call `replacePage()` in a loop until the return value is not `VM_PAGEDOUT`.