# CSCI 432 Lab 2 Report

Angus Li, Chris Brown

21-11-2023

## 1 Introduction

In this assignment, we wrote an external pager, a userspace process that handles virtual address space initialization, manipulation, and destruction for application processes. The goal was to understand the basic functionality of virtual memory management with pages, and to understand the implementation of key elements of the system such as faulting and the page swap algorithm.

A software-based infrastructure package that emulated a MMU was provided by Jeannie. Applications must include both our pager and this virtual MMU, allowing them to request more memory, read and write data, and ask for a range within memory to be logged by the system. We evaluated our work by writing small tests that allowed us to gauge our progress and correct subtle mistakes in our codebase.

## 2 Design

For clarity, virtual pages (vpage), physical pages (ppage) and disk blocks are numbered using the `vpage_num_t`, `ppage_num_t` and `disk_block_num_t` types respectively, which are all aliases for unsigned longs.

We introduce two structs to model vpages and processes. The vpage struct contains a pointer to the page's entry within the MMU's page table, a unique disk block assigned upon first access that is never reallocated or changes, a dirty bit, a zero bit and the owner process' pid. The process struct includes a page table and a map of vpages.

In terms of global state, we track all processes as mapped by their pid, a queue of free ppages, a queue of free disk blocks, a queue of resident vpages[1] and the current pid.

As with the first assignment, a collection of carefully designed helper functions was instrumental, although we use fewer in this pager than in our thread library. The majority of our non-trivial helper functions are used by vm_fault: these include functions to abstract away the process of swapping in and out a given vpage, and to update its bits according to its current state and the control flow of our fault implementation.

[1] This is used by the page swapping algorithm, which is described below.

Of particular note is `next_swappable_vpage`, which uses the clock algorithm to determine what vpage can be swapped out next. All resident vpages are tracked in a queue, as noted above. When called, `next_swappable_vpage` pops from this queue, and uses the read and write bits to determine whether it has been referenced recently. If the read and write bits are zero, indicating it hasn't, it returns this vpage to the caller. Otherwise, it sets the read and write bits to zero and pushes the page back onto the queue. Importantly, this algorithm is guaranteed to terminated after one iteration through all elements in the queue. Pages that are referenced again by an application process after this process are handled by vm_fault.

# 3 Evaluation

One requirement for the lab was to provide a suite of tests that catches various bugs in our virtual memory pager. Our memory pager caught 17/20 of the buggy memory libraries. It is able to print out the characters stored in virtual memory given a start address and a length, even if the start address is in the middle of a virtual page and the requested region of memory ends in the middle of a virtual page. The pager also makes sure that each virtual address being read from has had a virtual page assigned to it (through extend being called), and that each virtual address is valid. The pager is also able to allocate and de-allocate virtual memory pages if multiple processes are used. We did not write a specific test for this, but we received a 75/75 on the external pager tests, so this condition must be met. For future work, we could explicitly stress our memory pager on concurrent workloads. Our memory pager also makes sure to only write virtual pages to disk if they are dirty; this implies that our memory pager only writes non-zeroed out pages to disk because pages that are zeroed out are not dirty. Lastly, our memory pager has the ability to read from pages whose "reference" bits have been cleared by the clock algorithm by "faulting" them in, even though these pages are still in physical memory and not on disk. This nuance underscores the difference between "resident" and "referenced" pages.

# 4 Conclusion

We learned a lot in this project. The most important lesson was the significance of making good design choices when designing a large piece of software with many parts that rely on each other. Talking with Jeannie early on in the project about the structs we panned on using was really helpful because it helped us lay out what our vm_init, vm_create, and vm_extend functions were individually responsible for. Thinking about what needed to be in the process struct and vpage struct early on, before we wrote any code, gave us helpful experience designing large software systems. We also learned the importance of using commenting and process of elimination to find bugs in our code. This helped us find the source of a seg fault—we were using the value stored in

page_table_entry− >ppage before checking to make sure that the physical page was resident and not -1. This lab also made us better at creating tests for the code we wrote—I personally found myself spending a lot less time writing test cases for this lab than for the thread lab. Overall, we had a lot of fun with this lab!