Demand-paged virtual memory system

Alexander Einshøj

May 4, 2016

Introduction

In this project, a demand-paged virtual memory system, with the USB stick as the swapping area, was implemented.

Technical Background

Page faults

Page faults occur when something has "gone wrong" in terms of accessing a page in some sort. There are different page fault error codes to be aware of, but when you break it down, there's really only two different outcomes. The error code number is either divisible by 2, or not. If it is, we know that the page fault was caused by a request to a page or page table which does not currently exist, the presentbit was 0. In this case, a new page table or page has to be allocated in order to fulfill the request that was made. The other main scenario, is when the request has been made to access a part of memory that the requestee does not have access to. As an example, if a process wants to access the kernel directly, a page fault with an odd number should occur.

Mapping

In order for swapping to be possible, one needs to map the physical addresses available, to a virtual address. There are certain parts of memory that has to be mapped in a certain way. The kernel page directory should be identity mapped, so that its address is the same for every kernel thread, and because the kernel should never be swapped, we give it a fixed position. VGA should be available for processes as well as threads, so that they can appear on the screen. As for this assignment, we know we want to pin a set amount of the pages, so that they can not be swapped. The only pages we allow to be switched, is the ones on the bottom level of the hierarchy, which would be the actual pages.

Design

The design for this assignment was quite interesting, as there are a lot of ways to solve the issue at hand. When working with memory, it's vital to make sure that every value used, for reading or writing, and for assigning, is correct. When allocating a page, the current solution will simply assign a page until there is no more free memory, and eviction is needed for further continuation. When evicting with a random page replacement, the first thing to do is choosing a page to evict. The page cannot be a directory or a table, and it cannot be pinned (which means no stacks either). If there is an un-pinned page available, we can continue towards with the page allocate request. After setting the page to replace as not present, the tlb has to be flushed. The tlb (Translation Lookaside Buffer) keeps track of recently used pages, and in the case that our soon to be evicted page is in there, it has to be flush to avoid a situation where the tlb has an outdated value. Now, the evicted page can be written to disc with , so that there is room for the new page.

The page fault handler looks for two specific kinds of error codes, where only one of them is handled, as the other should not ever happen as implemented. Assume that the error code we get is not an odd number. In this case we know that what's missing is a page, because we already checked whether the page table was present. At this point, after allocating a page, we can read the information needed from the disc onto the physical page, and mark the page as present in the given page table.

Implementation

This code was written in C, on top of existing precode written in C and Assembly.

Discussion

There were many ways to implement the code for this assignment, as well as many choices for mapping. As is now, each process takes up 6 pinned pages, which means that when using 33 pageable pages, loading all four processes

won't be possible, as there will only be one page left to evict. With only one page left to evict, everything should in theory still run, and it does, but it takes a very long time, as the number of disc accesses will go through the roof, and you will not be able to see any change in terms of the processes. This is because with only one page left, every instruction has to be fetched from disc. Therefore, I have put number of pageable pages to 35, as there will be three unpinned pages with everything running, and all the processes will run, albeit slowly.

In a real scenario, we will probably not write back to disc, as this obviously overwrites what was there before. One would very rarely actually want that, but in this case it's a necessity to make things load back at the state they were previously in.

It is vital to store certain values when allocating, but only because eviction in this case requires write back to disc, and therefore the values of what is to be evicted.

Conclusion

With 34 pageable pages as a required minimum to run all the processes(1-4), I believe this assignment was solved in a reasonable manner. Eviction and page fault, which are the two main elements, both work perfectly according to the requirements.