INF-2201-1 23V: Operating system fundamentals

Assignment 5

Amund Strøm

Email: amund.strom@gmail.com

GitHub user: ast283

May 1, 2023

1 Introduction

This report describes the design and implementation of a simple demand-paged virtual memory system, where each process has its own virtual address space. This introduces a protection system so that processes may not access or modify data belonging to another process. Virtual memory solves physical memory shortage by allowing a process to swap parts of its data between physical memory and a mass storage device, depending on if the data is being executed.

2 Technical Details

These are some of the concepts and problems that is discussed in this report, and it is important to have an common understanding of these.

- Physical memory describes the memory addresses that actually exist on a memory unit, most commonly referred to is the random-access memory (RAM).
- Virtual memory is the memory addresses created by the memory management unit (MMU). A process uses the virtual addresses to execute and think it referrers to the physical memory, but in reality it may point to either the physical memory or a storage device. Whenever a process operates on a virtual address, the MMU translate the address to see if the block of code resides on physical memory or on a storage device. If it resides on a storage device it is moved into the physical memory, (most likely) replacing some other information that resides on physical memory, that is then moved to a storage device. This technique is used to compensate for the lack of physical memory.
- Pages and frames referrers to a block of information. Pages referrers to a storage device and frames referrers to physical memory. Pages and frames have the same fixed size, commonly used is 4096 bytes (4 Kb).
- Two-level paging is the process of dividing a process into equal sized pages and sort the processes into a three level hierarchy. The first and lowest level called a physical page contains the actual information of a process (such as code/data and stack), and is 4 Kb in size. The second level labeled the page table contain entries that point to a single physical page on the first level, and is also 4 Kb in size. Each entry in the table is 4 bytes and a table is able to hold 1024 entries. Therefore a single page table *can* point to a 4 Mb area. The final and third level labeled the page directory contain entries that points to the second level, and is also 4 Kb in size. Using the same logic as before, each entry in the directory is 4 bytes and can hold up to 1024 entries. Therefore a single page directory *can* point to a 4 Gb area.

  Each process is assigned a page directory, thus a process thinks that it owns a 4 Gb consecutive area in physical memory. But in reality the process's information is discontinuous and scattered through physical memory and storage devices. (Refer to figure 1 on next page)

- Page fault is an exception that is raised when a process accesses a physical page without proper preparations. A common problem is that the physical page is not present in physical memory and resides on disk, therefore a swap operation must be done.

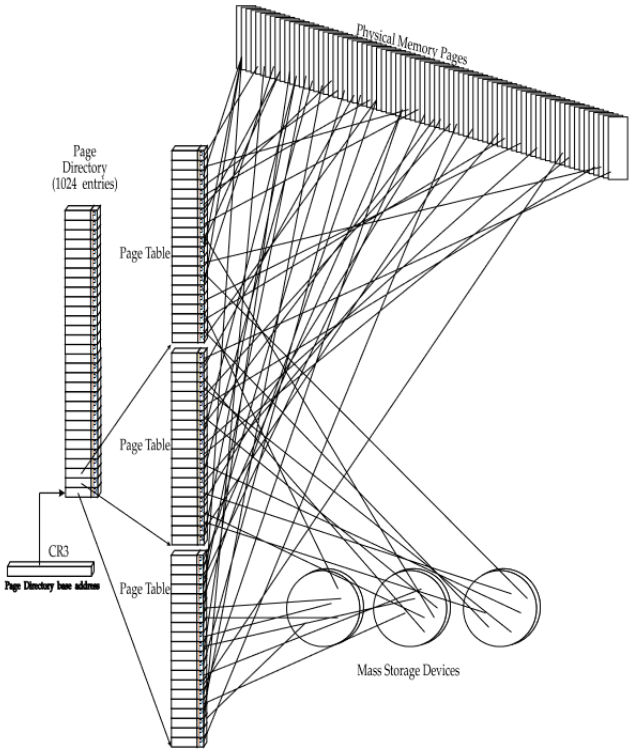

Figure 1: Memory hierarchy. Page directory, page table, and physical pages

3 Design

Including the extra credits, this project required the implementation of two-level paging per process, a page fault handler that allocates physical pages and fetches pages from disk, a replacement policy that replaces frames on physical memory when there is no free frames available, and a swap area on disk allowing multiple copies of the same process to run simultaneous.

Since all of these requirements are interconnected, this section will describe the functionalities of the four main functions in the program.

3.1 Initialize memory

This function is only called once and initializes several data structures and variables that will be used throughout the program.

- A buffer the size of a page filled with only 0, used for clearing a page on disk whenever a swap is about to occur.
- Locks that will be used for memory allocation and paging operations.
- Meta data about frames on physical memory, such as the physical memory address where the frame resides, if there exists information on the frame or not, if the frame is pinned or not, and the order of which frame should be evicted first.

- Meta data about pages residing on the dedicated swap-space on disk, such as the address on disk where the page resides, and if there exists information on the page or not.
- The kernel page directory, which is mapped to where the kernel resides on physical memory.

3.2 Setup

This function is called once per thread and process and sets up the necessary structures for a thread and process to run. Since this function is a paging operation it must acquire a lock. If a thread is the caller, set the threads page directory as the same as the kernel directory, since a thread operates inside the kernel. If a process is the caller it needs a separate page directory, which must also be mapped to the kernel, but since it is a process it has less privilege and must therefore set the user bit. This is done to achieve protection so that a process cannot access and modify data inside the kernel.

3.3 Get free frame

This function is called every time someone wants a frame in physical memory, it handles eviction if there is no free frames left. It takes three arguments, if the frame should be pinned or not, the base address of the "parent" page, and the index at which the acquired frame exists in the "parent" page. The parent page referrers to the page that is at a higher hierarchy level.

Firstly the function searches for a fitting frame. There is two cases, either a frame is free, meaning there is no information in the frame and the function can just return. Or there is no free frames, meaning there must be done an eviction before returning. The function follows the "First in First out" rule when it chooses which frame to evict.

If an eviction takes place it must flush the address at which the evicted frame exists on physical memory from the Translation Lookaside Buffer (TLB). Thereafter it must find a free space on disk where it can write the evicted frame into, and update the entry in the "parent" page with this address. Next time a page fault happens when this page is not present, it knows to read the disk at the updated address. Clear the free space on the disk, and write the frame into the space. Then clear the frame in physical memory so that the caller gets a clean frame, then return.

3.4 Page fault handler

This function handle exceptions that occur when paging is enabled. The only exception that should happen in the implemented operating system is that a page is not present in physical memory. When this function is called the currently running process contains the faulting address (virtual address at which the page fault occurred), the error code that tells what caused the exception, and the base address of the page directory unique for this process. By checking the bits in the error code it is possible to decipher what caused the problem. In this project we only care about the first bit, if it is not set to 1, it means that a page is not present in physical memory and must be inserted for the process to continue its execution.

When a page is not present there is two cases: Either a page table is not present or a physical page is not present. To figure out which of these cases is true, we must check the present bit in the entries of either the page directory or page table. (Refer to figure 2 on next page)

Case one, a page directory entry is not present i.e. a page table is not present. The last 10 bits of the fault address works as a page directory index, therefore combining it with the base address of the currently running process's page directory, you get the entry that caused the exception. Now identify if the entry is present by checking the first bit, if it is set the entry is present, if it is not set, acquire a new frame and insert it as an entry in the directory and return.

Case two, a page table entry is not present i.e. a physical page with data is not present. The 10 bits in the middle of the fault address works as a page table index, therefore combining it with the base address of a page table, you get the entry that caused the exception. The page table base address is acquired from the previous case. Now identify if the entry is present by checking the first bit, if it is set the entry is present, if it is not set, there is two new sub-cases. Either the data must be fetched for the first time, or it must be fetched from the swap-space on disk.

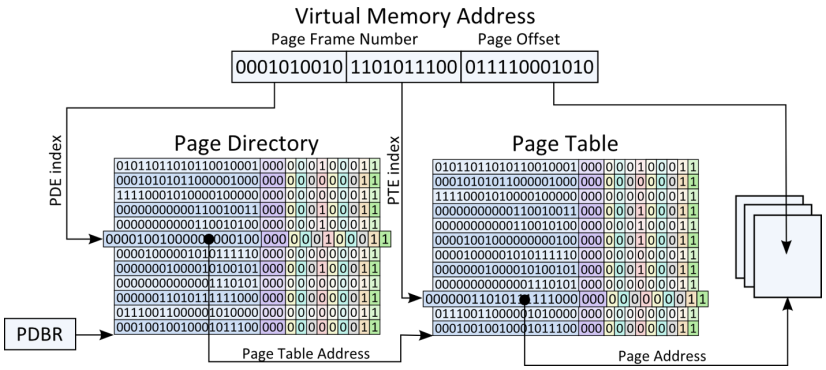

Figure 2: Present bit handling

Sub-case one, the data must be fetched for the first time. This is achieved via the same method as the previous project, by reading the process directory and finding the offset and size of the process. These variables is saved in the currently running process. Then simply read the disk at those values and insert the frame into the table and return.

Sub-case two, the data must be fetched from the swap-space on disk. As mentioned in the previous section, if a page is evicted from physical memory, it will write the address at where it resides on disk into the "parent" page. Therefore we simply read the size of a page on the disk, starting at the address in the page table entry, and insert the frame into the table and return.

4 Discussion

There is no known bugs or problems in the program, but there are things that could been solved differently. The setup function only sets up the page directory for a process, but it is possible to setup every page table too, meaning there is no point in checking if the page table is present in the page fault handler. This would cause a problem if the page table would be evicted, but the implemented program only evicts physical pages so it wouldn't matter. It is rather a good practice to handle a missing page table, since it would look more like a real page fault handler.

The extra credit suggest to implement a better eviction algorithm, such as FIFO. But FIFO is not necessary better compared to random eviction, since the "First in" does not implicitly say that it should be the first to be evicted. But FIFO has another huge advantage compare to random, which is that it is much easier to debug the program, since you could predict beforehand what page is about to be evicted. A better and much more difficult eviction algorithm could be "Least likely to be used", this could be implemented by looking though the ready queue, and evict a page that belong to the process that is either last or does not exist in the ready queue.

4.1 Conclusion

This report described a high-level view of the implementation of a simple demand-paged virtual memory system, and described the requirements and extra credits included in this project. It also discusses different design choices and eviction policies.

References

[1] Tom Shanley. Protected Mode Software Architecture. Chapters 13 and 14.

[2] OS dev contributors. Exceptions. In OSDev wiki. Retrieved: April 27th, 2023, from `https://wiki.osdev.org/Exceptions`