INF-2201
Project 5, Virtual memory
April 2022

Name: Magnus Dahl-Hansen
Github Username: MagnusDH
Email: mda105@uit.no

Introduction

This report describes how the implementation of a two-level paging system, page allocation and a page fault handler supports virtual memory. This system is used to load and run threads and processes on demand from a storage device/image-file.

Technical background

Before reading this report the reader should be familiar with the concepts of virtual memory, two-level paging, pages and page-faults.

Virtual memory is a layer of indirection which allows programs to run even if they require more memory space than available in a computers random access memory (RAM). The idea behind this concept is that each program has their own address space with “imaginary” addresses that can be translated to real physical addresses in physical memory. This allows a program to run even if only parts of the program is in physical memory.

The virtual memory of a program is divided into pages, which in this implementation contains 4 kilo bytes of data. The main reason for dividing a program into pages is to save memory space because translating each and every virtual address to a physical address requires a lot of memory. Therefore, translating and moving bigger chunks of data requires fewer translations and saves memory. Page faults occur when a program requires a page that is not currently in physical memory and so the page needs to be loaded from a storage device.

Two-level paging is a way to save even more memory space and allows multiple programs to run at the same time. Instead of storing pages for a program in RAM we can store a directory containing pointers to pages in RAM. In other words, two-level paging is a way to organize data of a program by having a master directory containing pointers to different page tables, and each page table containing pointers to different pages containing physical data.

Design&Implementation

This implementation consists of six different helper functions that allocates memory, creates a new page, place the page into a table or directory and manually place a given address a table or directory.

When a new page is created, 4096 bytes of memory is allocated from a memory pool starting at address 0x100000. The start address of the page is then set to the allocated memory space and all 1024 entries in the page are set to zero. The created page can then be placed inside a table or a directory at a index based on a given virtual address. Before the page address is placed inside a table or directory the 12 least significant bits are set to zero and any entry bits that the user wants to set are placed in the page address.
There is also an option to modify or insert an address into a given index in a table or directory and is done similarly as placing a page into a table or directory, but only specifying an address instead of the page address.

The kernel memory hierarchy is set up by creating a page directory and a page table where all entries are mapped to physical pages. The kernel code starts at address zero so there is no need to set up any translation between the kernels virtual memory and the physical memory, meaning virtual address zero corresponds to physical address zero and so on. All pages in the kernel table are then marked as present, placed inside the kernel table which is also marked as present and placed inside the kernel directory.

When setting up the memory hierarchy for a process a page directory is first created. For a process to run correctly it needs a code table that stores pages with translations to physical memory, a stack table and access to the kernel table. Therefore two page tables for stack and code are allocated and placed inside the process directory along with the kernel table containing kernel code. Entry bits allowing reads/writes and user privileges are set along with the present bit for each table.

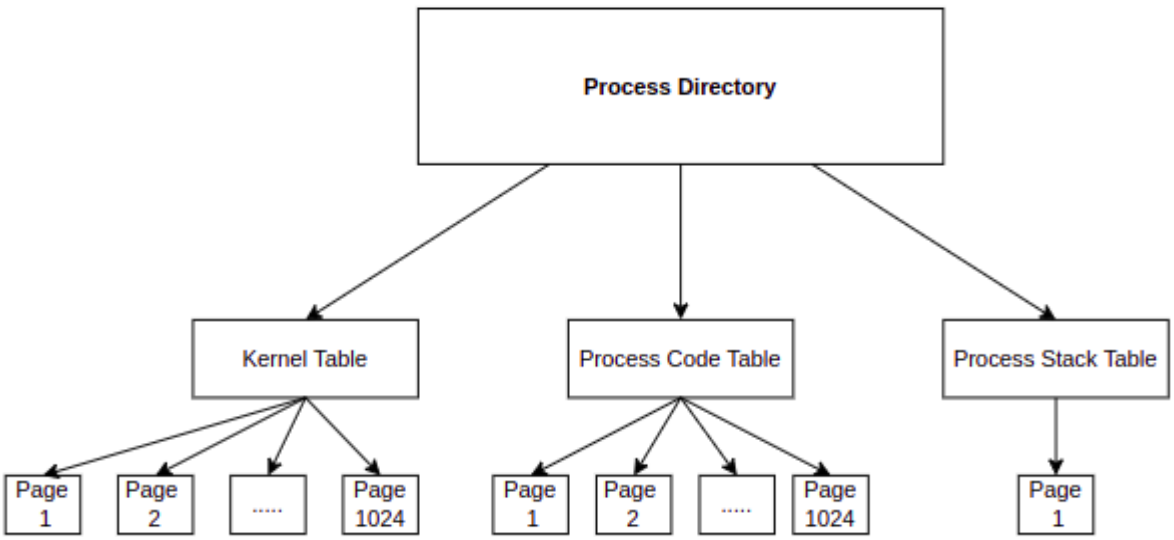

Figure 1. ***Visualization of the created memory hierarchy.***

If a requested page is not in memory a page fault will occur and the page fault handler function is called. This function will take different paths based on if there are any free space left in memory to create a new page or if there have been created enough pages to take up all possible memory.
If there is space to create a new page then a new empty page is created. Then the start address of the process's code along with the faulting virtual address is used to calculate how many blocks of data the process has been able to run so far. This is because we only want to read the required code

from a certain point of execution to the new page. How many blocks of data the process have left to execute is also calculated because we do not want to write code that does not belong to the process into the page. The code is then read from disk with a certain block offset into the new page, then the page is placed inside the process's code table, marked as present and the execution is continued.

If there is no free space left to create pages then a page needs to be evicted and make space for another page containing the missing code. A random page except the kernel directory, kernel table, process directory, process code table and stack table is then chosen to be evicted. Before eviction the page is checked for modification. The contents of a modified page is first written back to its start location in physical memory with a certain offset before it is evicted/erased. When this has been done then the page is reset by setting all entries to zero and the translation lookaside buffer is flushed. The location of the missing code and the block offset is then calculated and written into the newly reset page. The page is then inserted into the process code table, marked as present and the execution continues.

Results & Discussion

The implemented code is able to load the starter thread which loads the shell, the kernel threads and up to three processes. After loading the third process the program will run out of available pages to write code to and a page fault will occur that is not handled correctly. Therefore it will not be possible to load a fourth process. The page fault handler function is able to handle page faults when there are enough space to create new pages, but will fail when this is not possible and pages have to be evicted. The reason for this may be a combination of how the page is evicted, that the missing code is not written to the page correctly and how the entry bits are placed in the code table for each page.

Trying to resolve this problem was hard since the directories, tables and pages were not easily organized and manageable. Therefore an attempt to convert each page into a data structure were made to make the pages more easy to manage. The page structures were made to contain a start address for a page, its size, a buffer to hold 4096 bytes of data, a pinned variable and a modified variable. By having this kind of structure it would have been easier to evict, write to and replace pages from the start of the implementation.

The implementation has been run and tested on a HP laptop with the following specs: AMD Ryzen 5 2500U processor, Linux operating system, 8GB RAM, using the Bochs emulator.

Conclusion

This report described how pages are created and placed inside page tables and page directories to create a two-level paging system. The implementation of the page fault handler and how it works is also described along with an attempt to handle the eviction of pages in a correct way.
The implemented code is so far able to load the shell, run the kernel threads and load up to three processes before running out of free pages to write missing code to.