OS 2021 – Exercise 4
Virtual Memory – Hierarchical Page Tables

Supervisor – Ihab Zhaika

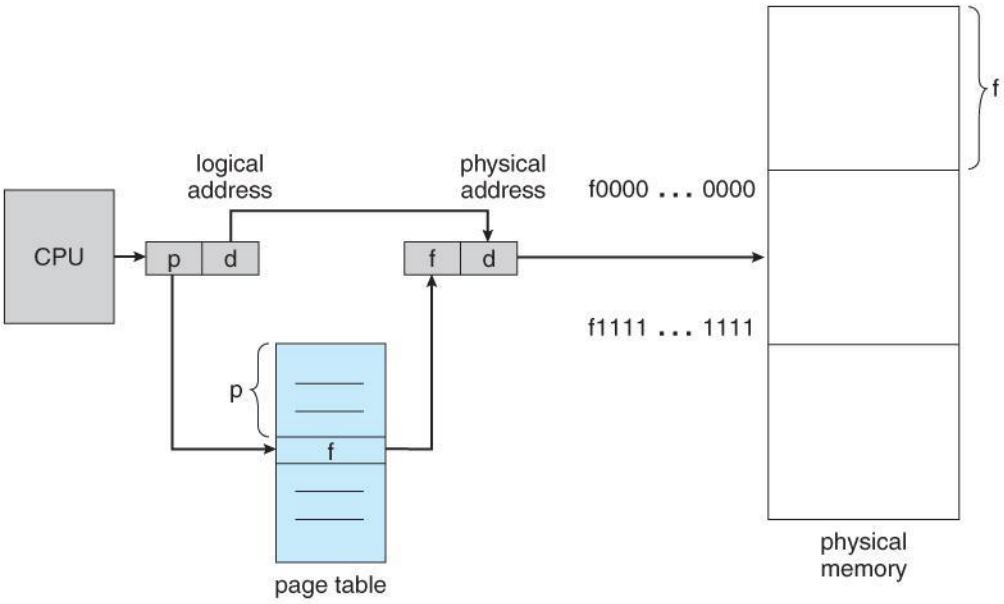
Virtual Memory Review

Virtual memory is a memory-management model that allows processes to use more memory than is actually available on the host. This section is mostly a summary of Tirguls 7-8.

This is achieved by mapping *virtual (a.k.a. logical) addresses* (what the process sees) into *physical addresses* (actual locations in the RAM). Since the *physical address space* is significantly smaller than the *virtual address space*, not all memory space of all processes (or even of one process!) can fit in the RAM at the same time. Therefore, parts of the virtual memory must be stored in the hard drive until the next time they are used by their process.

Paging is an approach where the virtual address space is partitioned into fixed-sized contiguous blocks, called *pages*. The physical address space is similarly partitioned into blocks the same size as pages, called *frames*. At any given time, each page is either mapped to a frame in the physical memory or is stored in the hard drive. When a process tries to access a virtual memory address in a page that is not in the physical memory, that page must be brought into the physical memory (swapped in). If there are no unused frames, another page must be evicted from the physical memory (swapped out).

The mapping between pages and frames is done using *page tables*. The naïve implementation will have a big table where the number in the p^{th} row is the index of the frame to which the p^{th} page is mapped.

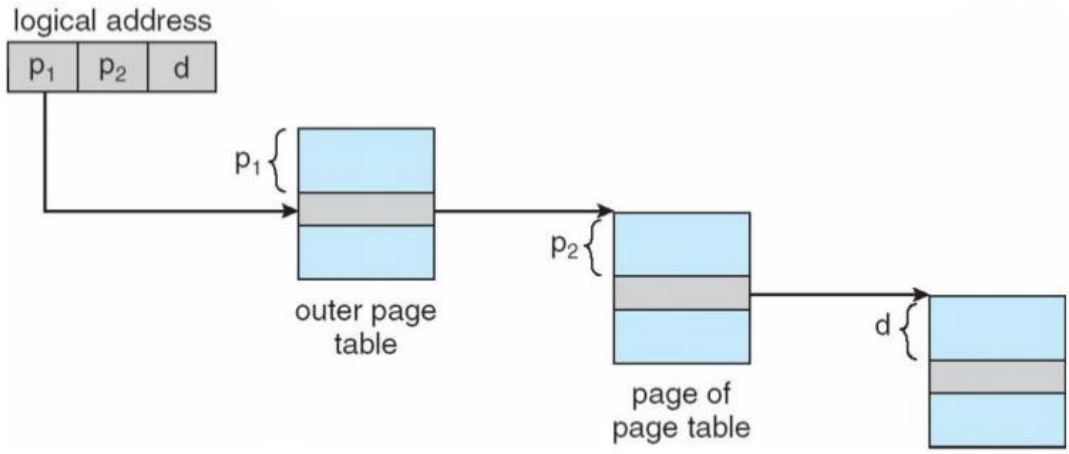


In this implementation each address is split into two parts: the offset and the page number. The offset (d in the above figure) is the position within the page and hence doesn't need any translation. Its width in bits is **$\log_2$(page_size)**. The page number is the rest of the address and its width is the remaining bits: **$\log_2$(virtual_memory_size) - $\log_2$(page_size)**.

The problem with a single page table is that it can be very wasteful in terms of physical memory consumption, usually there are many unused pages in the middle of the virtual address space which needlessly bloat the table, even to the point that it may not fit in the physical memory at all.

Hierarchical page tables use a tree of smaller tables to save memory. This effectively separates the translation into multiple steps. These tables reside in frames of the RAM just like any other page. Each row of a table either contains the number of frames containing the next table in the layer below it or, if it is in the lowest layer, the number of frames containing the relevant page.

Each table can only contain a small number of rows, so there are only so many bits it can translate on its own, $\log_2$(number_of_rows) bits to be precise. In total the page tables are supposed to translate **$\log_2$(virtual_memory_size) - $\log_2$(page_size)** bits, so the depth of the page tables tree must **be ceil(($\log_2$(virtual_memory_size) - $\log_2$(page_size)) / $\log_2$(number_of_rows))**. Below is an example of address translation using a tree with 2 layers of tables.



In this exercise you will implement a virtual memory interface using hierarchical page tables of arbitrary depth using simulated physical memory.

Tasks Overview

Your main task is to implement the virtual memory API, as defined by the functions in ***VirtualMemory.h***:

1) **VMinitialize()** – will be called before any other function is called.
2) **VMread(uint64_t *virtualAddress*, word_t* *value*)** – reads the word from the virtual address *virtualAddress* into **value*. Returns 1 on success and 0 on failure.
3) **VMwrite(uint64_t *virtualAddress*, word_t *value*)** – writes the word *value* into the virtual address *virtualAddress*. Returns 1 on success and 0 on failure.

You will do so by using the physical memory API in ***PhysicalMemory.h***

1) **PMread(uint64_t *physicalAddress*, word_t* *value*)** – reads the word from the physical address *phsyicalAddress* into **value*.
2) **PMwrite(uint64_t *physicalAddress*, word_t *value*)** – writes the word *value* into the physical address *phsyicalAddress*.
3) **PMevict(uint64_t *frameIndex*, uint64_t *evictedPageIndex*)** – evicts the page *evictedPageIndex* from the frame *frameIndex* in the physical memory and stores it in the simulated hard drive.
4) **PMrestore(uint64_t *frameIndex*, uint64_t *restoredPageIndex*)** – restores the page *restoredPageIndex* from the simulated hard drive into the frame *frameIndex* in the physical memory.

An implementation of this API exists in *PhysicalMemory.cpp*.

The file ***MemoryConstants.h*** contains many important constants such as the sizes of the memory spaces and the depth of the page tables tree. Make sure you understand the calculations within. **Notice we will test your code with different memory sizes and trees of different depths, so your implementation must not rely on the specific values of constants in this file.**

Similar to how virtual address translation is done using the page tables in the RAM, you will also have to use the simulated physical memory to store your page tables. **Therefore, you are forbidden from using global variables or dynamic allocation for the tables**. That includes dynamic allocation done by STL data structures. You must not use std::vector, std::map etc. This may sound difficult, but in fact it is quite possible. The next section details how.

Note that our implementation simulates the page tables of a single process. In reality there are multiple page table trees, one for each process.

Design

In our design, the root page table is always in frame 0 of the physical memory. Each row of the table (and of any other table) fits into a single word. Translating an address therefore is quite simple:

Assuming 2^4=16 words in each page, a tree of depth 2 and that the address is in the physical memory, writing to the address [**101**][**0001**][**0110**] will take 3 steps:

- PMread(0 + **5**, &addr1) // first translation
- PMread(addr1 * PAGE_SIZE + **1**, &addr2) // second translation
- PMwrite(addr2 * PAGE_SIZE + **6**, value)

A read will be quite similar with only the last step different.

The complexity arises when we consider that the virtual address may not be in the physical memory.

But first how do we know if a page or any of its parent tables are in the physical memory? Since frame 0 always contains the root table (which is never evicted), no row in any table will ever point to frame 0. Therefore, we can just put 0 in rows whose page is currently swapped out. Now, if during any step of the translation we reach a row that contains the number 0, we know that this is not a real frame index and we have reached a page fault. Now we have to swap the page back in and add it to the tree, creating tables above it and evicting other pages if necessary.

The full write therefore is:

- PMread(0 + 5, &addr1) // first translation
- If (addr1 == 0) then
  - Find an unused frame or evict a page from some frame. Suppose this frame number is f1
  - Write 0 in all of its contents (only necessary in tables)
  - PMwrite(0 + 5, f1)
  - addr1 = f1
- PMread(addr1 * PAGE_SIZE + 1, &addr2) // second translation
- If (addr2 == 0) then
  - Find an unused frame or evict a page from some frame. Suppose this frame number is f2
    - Make sure you are not evicting f1 by accident
  - Restore the page we are looking for to frame f2 (only necessary in actual pages)
  - PMwrite(addr1 * PAGE_SIZE + 1, f2)
  - addr2 = f2
- PMwrite(addr2 * PAGE_SIZE + 6, value)

The big issue left is finding the exact frame to put our page in and evicting another page if necessary.

We choose the frame by traversing the entire tree of tables in the physical memory while looking for one of the following (prioritized):

1) A frame containing an empty table – where all rows are 0. We don't have to evict it, but we do have to remove the reference to this table from its parent.
2) An unused frame – When traversing the tree, keep a variable with the maximal frame Index referenced from any table we visit. Since we fill frames in order, all used frames are contiguous in the memory, and if max_frame_index+1 < NUM_FRAMES then we know that the frame in the index (max_frame + 1) is unused. [it is inefficient to calculate it each time but in sake of this exercise do not use global variable]
3) If all frames are already used, then a page must be swapped out from some frame in order to replace it with the relevant page (a table or actual page). The frame that will be evicted is the frame that is mapped to a data page where the sum of the node weights from the root page to the required page + the page is maximal **(**node/page weight = WEIGHT_EVEN if the frame or page number of the node is even and WEIGHT_ODD if its odd**)**, if the two weights are equal, we will pick page with the smallest number to be evacuated, this page must be swapped out before we use the frame, and we also have to remove the reference to this page from its parent
   Note 1: check the "Flow Example" Page 19 for algorithm demonstration.
   Note 2: "WEIGHT_EVEN" and "WEIGHT_ODD" are defined in "MemoryConstants.h", and you can assume that they are non-negative integers

According to this design, VMInitialize() only has to clear frame 0. We've taken the liberty to already implement this in *VirtualMemory.cpp*.

Your Assignment

- Implement the functions defined in *VirtualMemory.h* using the design described above.
- Your code must support different memory space and page sizes in *MemoryConstants.h*. You can assume these are all valid.
- You must not use global variables or dynamic allocations.
- Don't submit any of the PhysicalMemory files and don't compile them with your Makefile

Tips

- There is a lot of bit twiddling in this exercise. These calculations are very easy to get wrong so try to separate them into small functions where possible and verify that they do what you think they do.
- Pay special attention to variable types and signedness. Optional further reading here.
- When swapping a page in, you may need to also create new tables for its ancestors in the tree. Make sure that you create these from top to bottom and that you don't accidentally evict a table you've just created.
- Test your code with different *MemoryConstants.h* and different tree depths.
- Don't include <iostream> in the submitted files as it defines global variables.

- Read and understand the “Flow Example” that appears in Ex4 section in the Moodle

Submission

Submit a tar file containing the following:

- A README file. The README should be structured according to the **course guidelines**. In order to be compliant with the guidelines, please use the **README template** that we provided
- The source files for your implementation of the library
- Your Makefile. Running *make* with no arguments should generate the *libVirtualMemory.a* library. You can use this **Makefile** as an example
- Don’t submit any of the PhysicalMemory files and don’t compile them with your Makefile
- Don’t submit the “Flow Example”

Late submission policy						
Submission time	25.6, 23:55	26.6, 23:55	27.6, 23:55	28.6, 23:55	29.6, 23:55	30.6
Penalty	0	4	12	26	48	100