MP4: Page Manager 2

The PageTable class implements a basic and efficient paging system for an operating system. The structure of the page table was built based on my implementation from the previous MP. The alterations made were in order for the page table to work with the virtual memory class. The VMPool class manages pools of virtual memory, allocating, releasing, and verifying the legitimacy of memory for processes. Its design was loosely based on the frame pool manager from MP2 with changes to work in virtual memory and to integrate with the PageTable class.

page_table.h

The header file for the PageTable class was structured to declare the key components of the paging system, including the page directory, frame pools, and essential functions for paging. I altered this file to include an array of VMPools and the number of VMPools in order to register the pools with the page table.

PageTable Class: The PageTable class manages the translation between logical and physical memory addresses by using page directories and page tables. It contains the following.
Static Members:

- **current_page_table**: Points to the currently loaded PageTable object.
- **paging_enabled**: Indicates whether paging is turned on (logical memory addressing).
- **kernel_mem_pool**: A frame pool for kernel memory.
- **process_mem_pool**: A frame pool for process memory.
- **shared_size**: The size of the shared address space, which is crucial for managing shared memory between processes.

Instance Members:

- **page_directory**: Points to the physical location of the page directory for the current page table.

Constants

- **PAGE_SIZE:** Defines the size of each page (in bytes), as provided by the underlying hardware (Machine::PAGE_SIZE).
- **ENTRIES_PER_PAGE:** Specifies the number of entries in a page table (Machine::PT_ENTRIES_PER_PAGE).
- **MAX_POOLS**: Sets the maximum number of virtual memory pools to 256 for the array of VMPools.

Functions:

init_paging: Sets up the global paging parameters, including the kernel and process memory pools, and shared memory size.

PageTable(): Constructor that initializes the page table by intitializing a page table with a given location for the directory and the page table proper.

load(): Loads the current page table into the CPU's control register (CR3) to switch address spaces.

enable_paging(): Enables paging by setting the paging flag in the control register (CR0). This allows the CPU to translate logical addresses into physical addresses.

handle_fault(): Handles page faults by allocating new page tables or frames when a logical address is accessed that does not yet have a mapped physical page using recursive page table lookup.

register_pool: Registers a VMPool with the page table.

free_page: Checks if a page is valid and if so, releases the page and marks it as invalid.

PDE_address: Given a logical address from the CPU, returns the address of the PDE.

PTE_address: Given a logical address from the CPU, returns the address of the PTE.

page_table.cpp

The implementation of the PageTable class provides the functional aspects of paging, such as allocating page tables, mapping memory, and handling page faults. Below are the key points of the implementation:

Static Initialization
Static members, including current_page_table, paging_enabled, kernel_mem_pool, and process_mem_pool, are initialized to nullptr and 0 to establish the basic environment before paging is turned on.

init_paging()
This function sets up the global parameters required for paging. It links the kernel and process frame pools and sets the shared memory size. This provides the foundation for paging across both kernel and process memory spaces.

Constructor: PageTable()

- Allocates a frame from the process memory pool to serve as the page directory.
- Maps the first 4MB of memory by creating and populating the first page table.
    - This involves setting the page directory entry to point to the first page table and marking the pages as present, supervisor-level, and read/write (using bitwise OR with 0x3).
- Constructor sets up the recursive mapping by setting the last entry in the page directory to point to the page directory itself.

load()

- This function writes the page directory's address into the CR3 register, effectively loading the page table for use by the CPU. This function must be called when switching address spaces.

enable_paging()

- Paging is enabled by modifying the CR0 register, setting the most significant bit to 1. This activates logical memory addressing, allowing the CPU to manage memory using the page tables.

handle_fault()

- This function was overhauled from the MP3 implementation to function with the recursive mapping
- Reads the faulting address (stored in CR2)
- Checks that the address is within the allocated memory in the VMPool(s) registered with the PageTable
    - If the address is not legitimate, an error message is displayed and the function returns
    - If the address is legitimate, execution continues.
- The PTE and PDE are calculated using their respective functions
- Checks for present page table
    - If page table is not present, a frame is allocated, its address is written to the PTE, and the page table is initialized to be empty
- Checks for present page
    - If page is not present, a frame is allocated and its address is written to the PTE

register_pool()

- Adds a VMPool to the list of VMPools and increments the count of VMPools

free_pool()

- Takes a page number to be freed and calculates the address
- Calculates the PTE and PDE using their respective functions

- Marks the page as page as invalid and flushes the TLB by reloading CR3 register

PDE_address():
- Extracts the index of the PDE by shifting 22 bits
- Calculates the virtual address using the extracted index
- Returns the virtual address of the PDE for recursive mapping

PTE_address():
- Extracts the index of the PDE by shifting 22 bits
- Extracts the index of the PTE by shifting 12 bits
- Calculates the virtual address using the extracted indexes
- Returns the virtual address of the PTE for recursive mapping

vm_pool.cpp

The implementation of the VMPool class provides the functional logic for memory allocation, release, and validation. The class keeps track of free and allocated memory regions using simple arrays. This implementation is loosely based on the frame manager from MP2.

VMPool():
- Initializes all variables for the VMPool
- Sets up arrays to have free space to be allocated
- Registers VMPool with the page table

allocate():
- Calculates the number of pages needed for a given size
- Loops through the free space in the pool and finds a region with enough pages
    - If no region is found, display error and return 0
- Allocates the page(s) and updates arrays and counters to track

release():
- Calculates the page that the address should be in
- checks if the given address is legitimate
    - If not throws error
- Loops through the allocated memory to find the start page that corresponds to the address
    - When page is found the page is removed from allocated memory and added to free space
    - Arrays and counters are updated

is_legitimate():
- Loops through the array of allocated memory and checks if the given address falls within any allocated region

vmpool.h

Data structure:
In order to track the free and allocated regions of the pool, I have used simple arrays that hold the base page of the region (free or allocated) and the length of that region. The regions have a counter that tracks the number of regions for the purpose of indexing into the arrays.

Debugging

Extensive debug print statements are included throughout the implementation to track the state of the free and allocated regions during allocation, release, and legitimacy checks.

Known issues (unresolved):
Using the console outputs I have found that my implementation is able to initialize the page table and initially set up a VMPool. As memory is allocated however, after several successful allocations and page fault handlings, the free and allocated array structure is unexpectedly cleared. I have been unable to determine exactly what is causing the issue and as a result this implementation does not complete the necessary testing