MP4: Virtual Memory Management and Memory Allocation

By: Mehul Varma

If there are any errors testing this code, Professor Bettati has instructured that the TA should contact him about my case.

Part 1 Introduction

In this assignment, we were supposed to implement our page table pages and the page directory inside of virtual memory itself. This was done using the recursive page table lookup technique implemented on top of the last assignment.

After this was implemented, we wanted to extend our memory management to handle VM pools inside of the logical memory space we created using our page table class. Once this VM pool class was implemented, we would hook up the C++ new and delete operators to the VM pool's allocate and release functions itself to mimic their use in an actual operating system.

Part 2 Data Design

We make use of two static variables inside of the page table class which are used for the implementation of the linked list of VM Pools:

- **`static VMPool* vm_pool_head -`** head of linkedlist of VMPools
- **`static VMPool* vm_pool_tail -`** tail of linkedlist of VMPools

Other than that, the other variables we make use of are the instance variables from the VMPools class:

- **`unsigned long  _base_address :`** address where the VMpool starts in logical memory
- **`unsigned long  _size:`** the size of VM Pool
- **`PageTable     *_page_table :`** pointer to the page table that the VMPool belongs to
- **`unsigned long* freelist_start_arr:`** pointer to the start of the free list start address list array
- **`unsigned long* freelist_end_arr:`** pointer to the start of the free list end address list array
- **`unsigned long* alloclist_start_arr:`** pointer to the start of the alloc list start address list array

- **`unsigned long* alloclist_end_arr:`** pointer to the start of the alloc list end address list array

Part 3 Architectural and component-level design

The majority of implementation for this Machine problem was the implementation of the Page Table fault handler and the VMPool class functions.

VMPool Constructor:

The VM Pool constructor's implementation is described by the following sequential steps:

1. The VMPool registers itself in the linked list of VMPools by calling the *register_pool()* function from its page_table instance object.
2. It initializes its instance variables from the given arguments.
3. It initializes the free lists and the array lists to the first two frames of itself.
4. It initializes the first elements of each of the free and array lists to show that the first two frames are allocated and the rest of the memory addresses are given to the free list.

While implementing these 4 steps, there are page faults because we write in the first two frames which are actually the free and alloc lists.

VMPool Allocate:

The allocate function implemented is simple as it just keeps a track of which regions are allocated and which aren't. This is done by keeping track of this in the free and alloc lists. Since we take the lazy approach, the page fault takes care of the actual allocation of the frames and proves to be more space-efficient.

The following steps are what the allocate function follows to implement this function-

1. It first converts the number of bytes to pages and takes the ceiling of this arithmetic.
2. It iterates through the free list's initialized indices and for each initialized index, it checks if that free region can accommodate the number of pages we need to allocate.
3. If it is able to find such a page, it finds an empty index in the alloc list and assigns it the start and end address which should be the space taken up by the number of pages we would allocate.
4. The index from the free list that accommodates our allocation is shrunk from the top and updated at the same address.
5. Finally, the function returns the address where we have allocated the required size in bytes. If it isn't able to allocate, it just returns 0.

VMPool Release:

The release function has a simple implementation as it just calls the *free_page()* function from the page_table and passes it to the page_no to be deleted. The following is how this function is implemented:

1. Iterates through the alloc list to find if the frame releasing is even allocated.
2. Once this is checked, *free_page()* function is called on every page belonging to the region we found out.
3. Once the pages are actually freed in physical memory, the index of the allocated region we just freed from the alloc list is put back into the free list.

VMPool Is Legitimate:

The *is_legitimate()* function just checks if the passed virtual address belongs to an allocated region in the VMPool.

The main catch to this part is hard coding the function so that all addresses in the first two frames are always legitimate since that is where we plan to store the free and alloc lists when the page fault happens when we are actually initializing them.

Other than that, we just iterate through each of the allocated regions in the alloc list and return true if the address is legitimate else we return false otherwise.

Page Table Fault Handler:

The page fault handler is changed in the sense that it iterates through each of the VMPools and calls their *is_legitimate()* function to make sure that the page that faulted is an allocated page in a VMPool, otherwise the kernel halts.

Other than that, the last MP’s solution is adapted to use the *PDE_address()* and *PTE_address()* functions as the paging are turned on and do not use physical addresses since the Page Directory and the Page Table Pages are stored in virtual memory.

Page Table Register Pool:

This function is a simple link list insert function where it uses the *next* instance variable of the VMPool it is called to insert it to the linked list we implement using the *head* and *tail* pointers in the page table class itself.

Page Table Free Page:

In this function’s implementation, given the page number in the virtual memory space, we get the page number’s PTE and from that, we get its physical which we use to call the *release_frames()* function so that that memory is physically released and we update the PTE after this is done.

PageTable Constructor changes:

Here just the recursive page table is implemented by initializing the last entry in the Page Directory to point to the Page Directory's physical address itself. Everything else remains the same from the last MP's implementation.

Part 4 Testing & Modifications

For this assign the modifications were only made to page_table.H, page_table.C, vm_pool.H, and VM_pool.C files.

Testing:

- The program was tested with the given tests where they were manipulated so that the VMPools could be implemented at different regions in the virtual address space.