MP4: Virtual Memory Management and Memory Allocation

Naveen Babu Krishnasamy Jeyakumar
UIN: 335007146
CSCE611: Operating System

Assigned Tasks:

Main: Virtual Memory Management and Memory Allocation - Completed.
Files Modified: cont frame pool.H, cont frame pool.C, page table.C, page table.H, vm pool.C and vm pool.H

System Design:

The objective of this machine problem is to enhance page table management to accommodate virtual memory and to develop a virtual memory allocator. In this task, page table pages are allocated in mapped memory, specifically in memory regions above 4 MB. These frames are managed by the process frame pool. The memory layout of the machine is as follows:

Total memory in the machine: 32 MB
Memory reserved for Kernel: 4 MB (Direct mapped to physical memory)
Process memory pool: 28 MB (Freely mapped to physical memory)
Frame size: 4 KB
Inaccessible memory region: 15 MB – 16 MB

Handling Large Address Spaces:

Due to the limited size of the directly mapped portion of memory, it cannot be used for larger address spaces. Therefore, we utilize the process memory pool to allocate memory for page table pages. When paging is enabled, the CPU generates logical addresses. To map these logical addresses to frames and modify entries in the page directory and page table pages, we use a technique called Recursive Page Table Lookup. In the recursive page table Lookup method, the last entry of the page directory points to the start of the page directory itself. Both the page directory and page table pages contain physical addresses. To access a page directory entry, we use the following logical address format:

| 1023 : 10 | 1023 : 10 | offset : 12 |

The MMU uses the first 10 bits (value 1023) to index into the page directory and look up the PDE. PDE number 1023 (the last entry) points to the page directory itself. The MMU treats the page directory like any other page table page. It then uses the second 10 bits to index into the (supposed) page table page to look up the PTE. Since the second 10 bits also have the value 1023, the resulting PTE points again to the page directory itself. The MMU uses the offset to index into the physical frame. To access a page table page entry, we use the following logical address format:

| 1023 : 10 | X : 10 | Y : 10 | 0 : 2 |

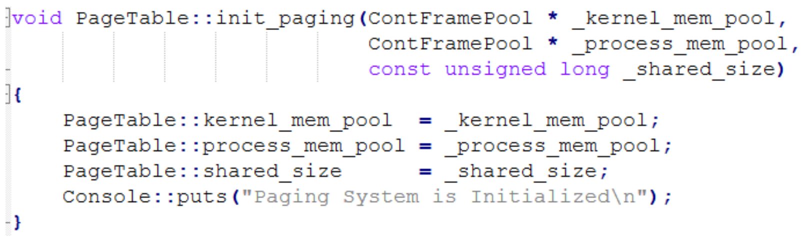
The MMU uses the first 10 bits (value 1023) to index into the page directory and look up the PDE. PDE number 1023 points to the page directory itself. The MMU treats the page directory like any other page table page. It then uses the second 10 bits (value X) to index into the (supposed) page table page to look up the PTE (which is the Xth PDE). The offset is used to index into the (supposed) physical frame, which is the page table page associated with the Xth directory entry. The remaining 12 bits can be used to index into the Yth entry in the page table page.

Virtual Memory Support for Page Tables:
The page table must be aware of all created virtual memory pools to distinguish between legitimate and invalid memory accesses. To support this, the page table maintains a list of all virtual memory pools. When a page fault occurs, it checks if the address is legitimate and releases previously allocated pages if necessary. We implement the following functions to support these features: register pool, is legitimate, free page. We use a linked list of virtual memory pool regions available for allocation. Each node in the linked list contains information about the base address, size of the memory pool region, number of regions, and available virtual memory pool size.

Simple Virtual Memory Allocator:
We implement a virtual memory allocator that can allocate and deallocate virtual memory in multiples of pages. Using the allocate function, we can allocate a region of virtual memory within the virtual memory pool. Similarly, using the deallocate function, we can deallocate a region of virtual memory within the virtual memory pool. An array is used to track the allocation and deallocation of virtual memory regions. Each array element is of the allocated vm_info structure type, which holds information such as the base address and length of the allocated virtual memory.

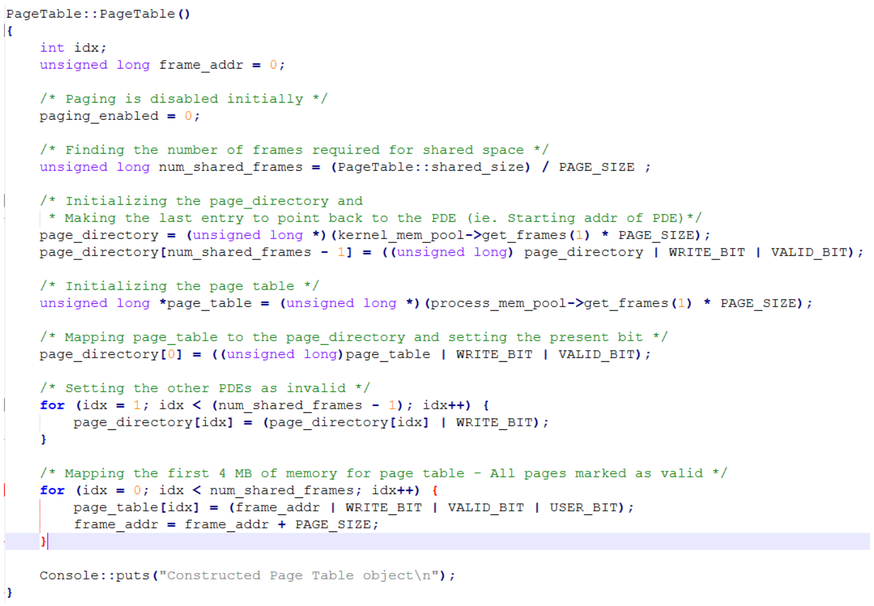
Code Description:

page_table.C : init paging: Configure the parameters for the paging subsystem.



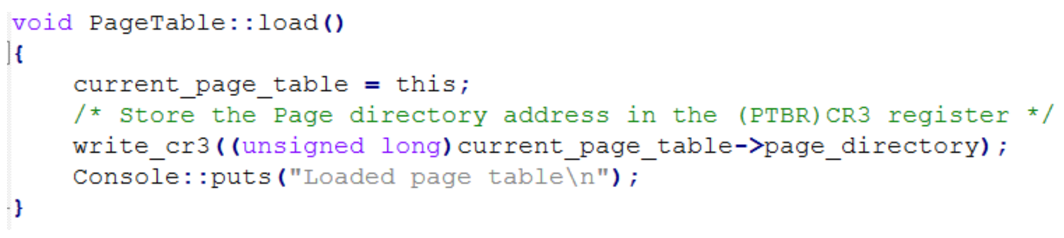
```
void PageTable::init_paging(ContFramePool * _kernel_mem_pool,
                            ContFramePool * _process_mem_pool,
                            const unsigned long _shared_size)
{
    PageTable::kernel_mem_pool  = _kernel_mem_pool;
    PageTable::process_mem_pool = _process_mem_pool;
    PageTable::shared_size      = _shared_size;
    Console::puts("Paging System is Initialized\n");
}
```

page_table.C : PageTable constructor: Initializes the entries in the page directory and page table for the shared memory segment. The page directory points to each page table in the shared memory and is marked as valid using the last bit of the entry. The page table entries for the shared memory segment (i.e., the first 4MB) are also marked valid using the last bit in each entry.



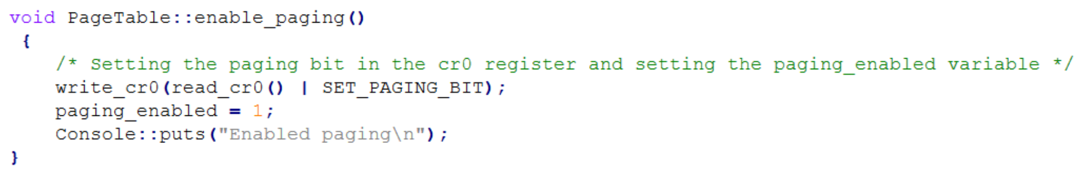
```
PageTable::PageTable()
{
    int idx;
    unsigned long frame_addr = 0;

    /* Paging is disabled initially */
    paging_enabled = 0;

    /* Finding the number of frames required for shared space */
    unsigned long num_shared_frames = (PageTable::shared_size) / PAGE_SIZE ;

    /* Initializing the page_directory and
     * Making the last entry to point back to the PDE (ie. Starting addr of PDE)*/
    page_directory = (unsigned long *)(kernel_mem_pool->get_frames(1) * PAGE_SIZE);
    page_directory[num_shared_frames - 1] = ((unsigned long) page_directory | WRITE_BIT | VALID_BIT);

    /* Initializing the page table */
    unsigned long *page_table = (unsigned long *)(process_mem_pool->get_frames(1) * PAGE_SIZE);

    /* Mapping page_table to the page_directory and setting the present bit */
    page_directory[0] = ((unsigned long)page_table | WRITE_BIT | VALID_BIT);

    /* Setting the other PDEs as invalid */
    for (idx = 1; idx < (num_shared_frames - 1); idx++) {
        page_directory[idx] = (page_directory[idx] | WRITE_BIT);
    }

    /* Mapping the first 4 MB of memory for page table - All pages marked as valid */
    for (idx = 0; idx < num_shared_frames; idx++) {
        page_table[idx] = (frame_addr | WRITE_BIT | VALID_BIT | USER_BIT);
        frame_addr = frame_addr + PAGE_SIZE;
    }

    Console::puts("Constructed Page Table object\n");
}
```

page_table.C : load() function: Loads the page table into memory by storing the address of the page directory into the CR3 register (PTBR).



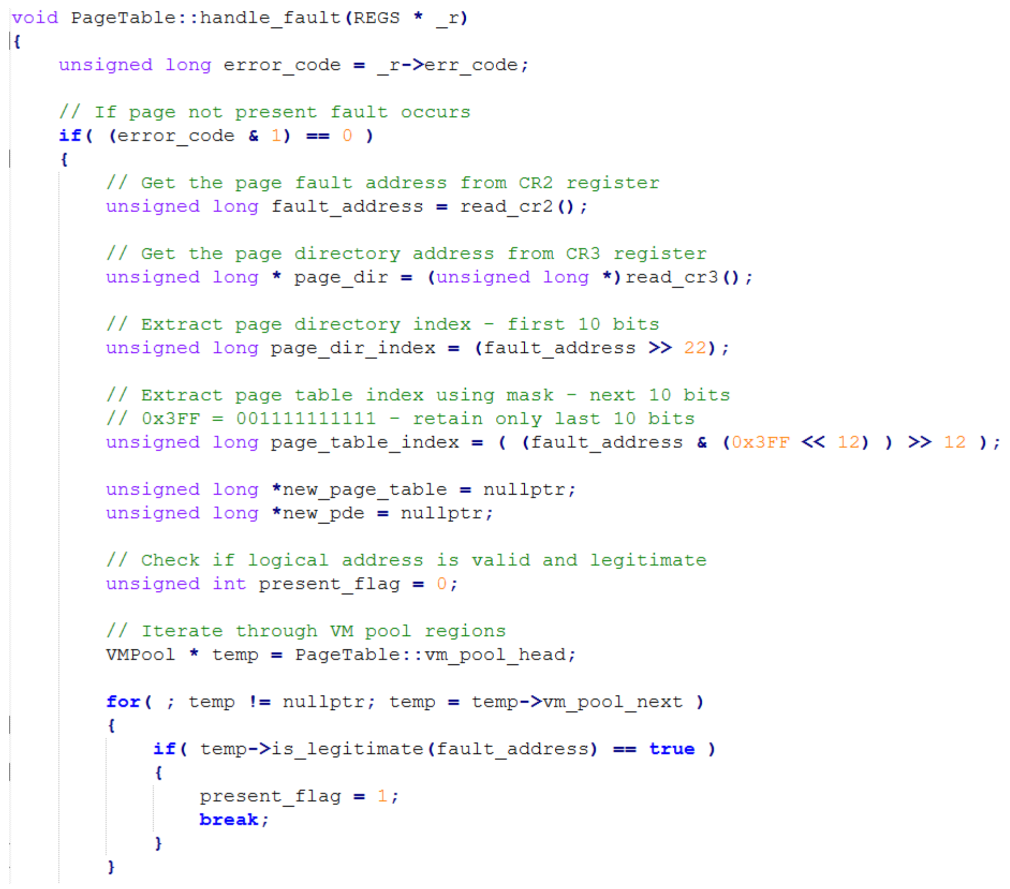
```
void PageTable::load()
{
    current_page_table = this;
    /* Store the Page directory address in the (PTBR)CR3 register */
    write_cr3((unsigned long)current_page_table->page_directory);
    Console::puts("Loaded page table\n");
}
```

page_table.C : enable paging() function: Enables the kernel to switch from physical addressing to logical addressing by setting the MSB in the CR0 register.



```
void PageTable::enable_paging()
 {
    /* Setting the paging bit in the cr0 register and setting the paging_enabled variable */
    write_cr0(read_cr0() | SET_PAGING_BIT);
    paging_enabled = 1;
    Console::puts("Enabled paging\n");
}
```

page_table.C : handle fault(): This method reads the logical address from the CR2 register and the error word from the error code in the REGS object. It then looks up the appropriate entry in the page directory and page table for faulty entries using the valid bit (0th bit). If there is no physical memory (frame) associated with the page, an available frame is allocated, and the page table entry is updated. If a new page table page needs to be initialized, a frame is allocated for it, and the new page table page and directory are updated accordingly.

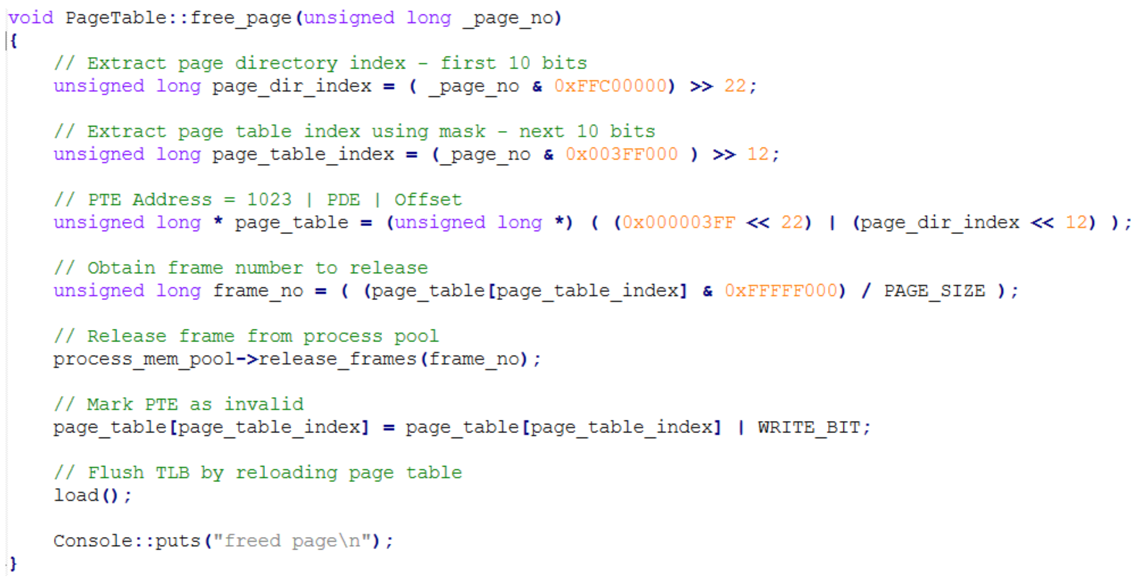


```
void PageTable::handle_fault(REGS * _r)
{
    unsigned long error_code = _r->err_code;

    // If page not present fault occurs
    if( (error_code & 1) == 0 )
    {
        // Get the page fault address from CR2 register
        unsigned long fault_address = read_cr2();

        // Get the page directory address from CR3 register
        unsigned long * page_dir = (unsigned long *)read_cr3();

        // Extract page directory index - first 10 bits
        unsigned long page_dir_index = (fault_address >> 22);

        // Extract page table index using mask - next 10 bits
        // 0x3FF = 001111111111 - retain only last 10 bits
        unsigned long page_table_index = ( (fault_address & (0x3FF << 12) ) >> 12 );

        unsigned long *new_page_table = nullptr;
        unsigned long *new_pde = nullptr;

        // Check if logical address is valid and legitimate
        unsigned int present_flag = 0;

        // Iterate through VM pool regions
        VMPool * temp = PageTable::vm_pool_head;

        for( ; temp != nullptr; temp = temp->vm_pool_next )
        {
            if( temp->is_legitimate(fault_address) == true )
            {
                present_flag = 1;
                break;
            }
        }
    if( (temp != nullptr) && (present_flag == 0) )
    {
     Console::puts("Not a legitimate address.\n");
     assert(false);
    }

    // Check where page fault occured
    if ( (page_dir[page_dir_index] & 1 ) == 0 )
    {
        // Page fault occured in page directory - PDE is invalid

        int index = 0;

        new_page_table = (unsigned long *)(process_mem_pool->get_frames(1) * PAGE_SIZE);

        // PDE Address = 1023 | 1023 | Offset
        unsigned long * new_pde = (unsigned long *)( 0xFFFFF << 12 );
        new_pde[page_dir_index] = ( (unsigned long)(new_page_table)| VALID_BIT | WRITE_BIT );

        // Set flags for each page - PTEs marked invalid
        for( index = 0; index < 1024; index++ )
        {
            // Set user level flag bit
            new_page_table[index] = USER_BIT;
        }

        // To avoid raising another page fault, handle invalid PTE case as well
        new_pde = (unsigned long *) (process_mem_pool->get_frames(1) * PAGE_SIZE);

        // PTE Address = 1023 | PTE | Offset
        unsigned long * page_entry = (unsigned long *)( (0x3FF << 22) | (page_dir_index << 12) );

        // Mark PTE valid
        page_entry[page_table_index] = ( (unsigned long)(new_pde) | VALID_BIT | WRITE_BIT );
    }

    else
    {
        // Page fault occured in page table page - PDE is present, but PTE is invalid
        new_pde = (unsigned long *) (process_mem_pool->get_frames(1) * PAGE_SIZE);

        // PTE Address = 1023 | PTE | Offset
        unsigned long * page_entry = (unsigned long *)( (0x3FF << 22)| (page_dir_index << 12) );

        page_entry[page_table_index] = ( (unsigned long)(new_pde) | VALID_BIT | WRITE_BIT );
    }
}

Console::puts("handled page fault\n");
```

page_table.C : register_pool(): This method is used to register a virtual memory pool with the page table. We use a Linked List to maintain a list of virtual memory pools. New virtual memory pools are appended to the end of the list.

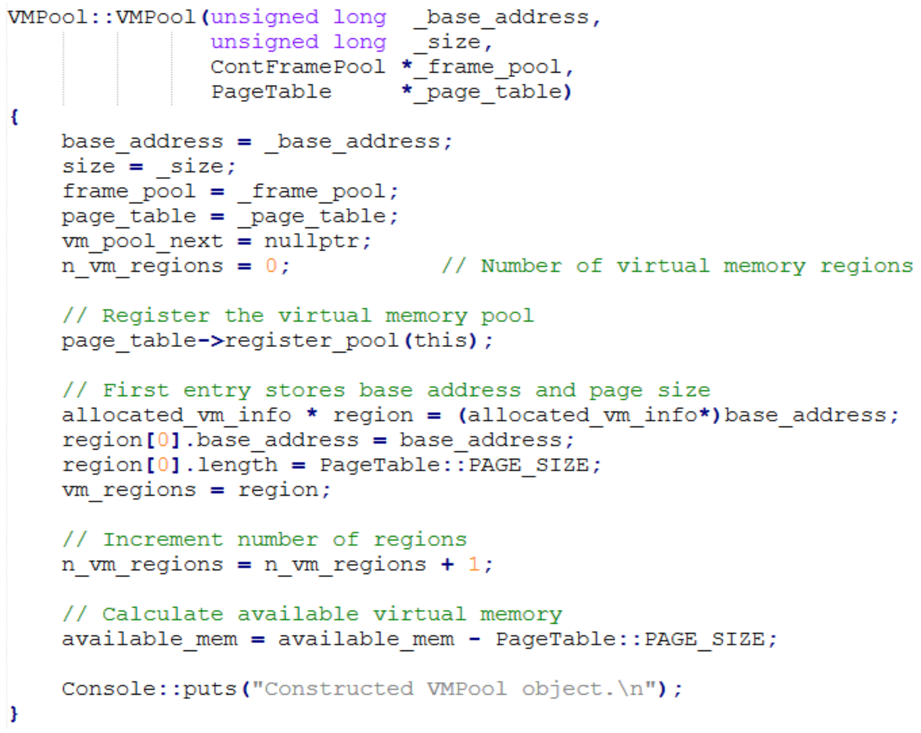
```
void PageTable::register_pool(VMPool * _vm_pool)
{
    // Register the initial virtual memory pool
    if( PageTable::vm_pool_head == nullptr )
    {
        PageTable::vm_pool_head = _vm_pool;
    }

    // Register subsequent virtual memory pools
    else
    {
        VMPool * temp = PageTable::vm_pool_head;
        for( ; temp->vm_pool_next != nullptr; temp = temp->vm_pool_next );

        // Add pool to end of linked list
        temp->vm_pool_next = _vm_pool;
    }

    Console::puts("registered VM pool\n");
}
```

page_table.C : free_page(): This method is used to release a frame and mark the page table entry ‘invalid’. We extract the page directory index and page table index from the page number. Next, we construct the logical address and access the page table entry to obtain the frame number. We release the frame from the process memory pool and mark the page table entry ‘invalid’. Finally, to erase all stale TLB entries, we flush the TLB by reloading the page table.



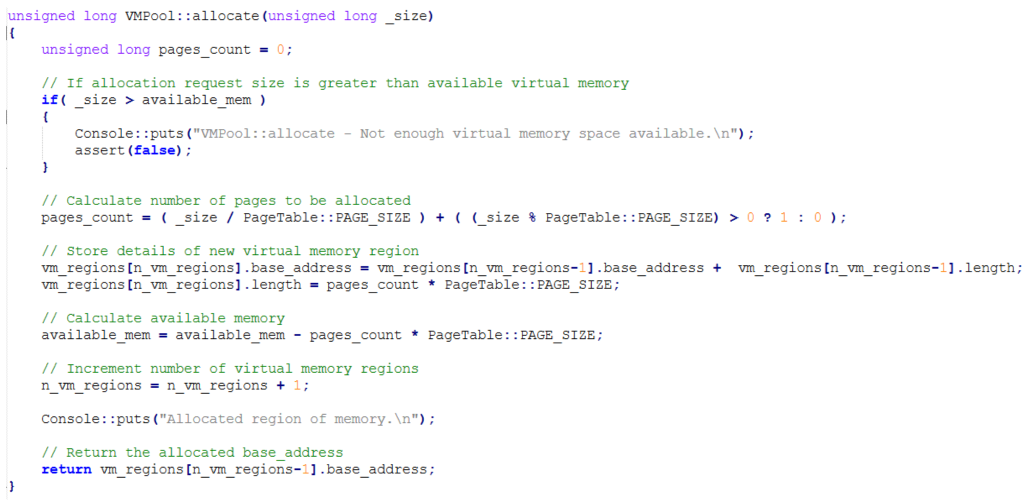
```
void PageTable::free_page(unsigned long _page_no)
{
    // Extract page directory index - first 10 bits
    unsigned long page_dir_index = ( _page_no & 0xFFC00000) >> 22;

    // Extract page table index using mask - next 10 bits
    unsigned long page_table_index = (_page_no & 0x003FF000 ) >> 12;

    // PTE Address = 1023 | PDE | Offset
    unsigned long * page_table = (unsigned long *) ( (0x000003FF << 22) | (page_dir_index << 12) );

    // Obtain frame number to release
    unsigned long frame_no = ( (page_table[page_table_index] & 0xFFFFF000) / PAGE_SIZE );

    // Release frame from process pool
    process_mem_pool->release_frames(frame_no);

    // Mark PTE as invalid
    page_table[page_table_index] = page_table[page_table_index] | WRITE_BIT;

    // Flush TLB by reloading page table
    load();

    Console::puts("freed page\n");
}
```

vm_pool.C : VMPool() Constructor: This method serves as the constructor for the VMPool class. It sets up all the necessary data structures for managing the virtual memory pool. The virtual memory pool is registered with the page table. The base address and length of the virtual memory region are stored in the allocated_vm_info structure. The variables vm_regions and available_mem are used to track the number of allocated virtual memory regions and the available memory in the pool, respectively.



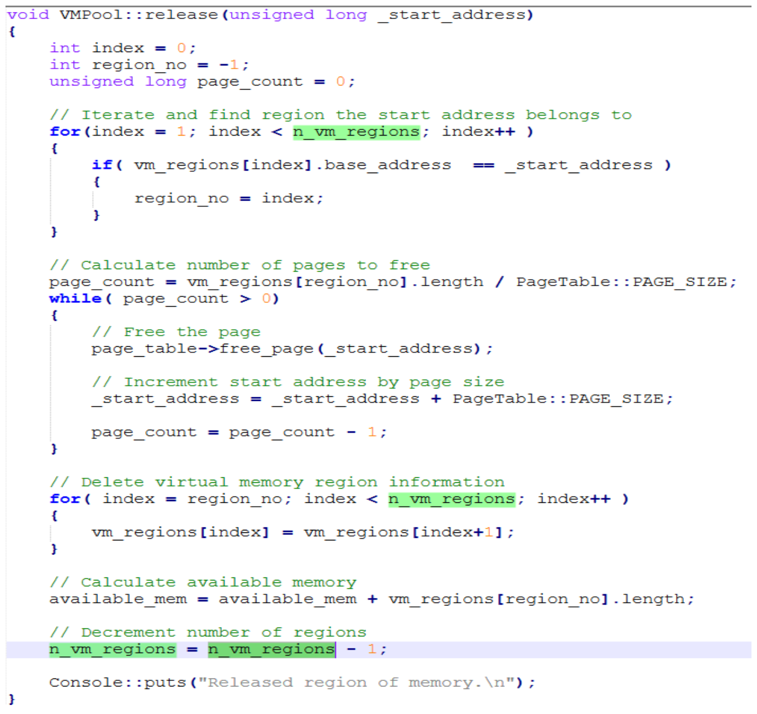
```
VMPool::VMPool(unsigned long  _base_address,
               unsigned long  _size,
               ContFramePool *_frame_pool,
               PageTable     *_page_table)
{
    base_address = _base_address;
    size = _size;
    frame_pool = _frame_pool;
    page_table = _page_table;
    vm_pool_next = nullptr;
    n_vm_regions = 0;            // Number of virtual memory regions

    // Register the virtual memory pool
    page_table->register_pool(this);

    // First entry stores base address and page size
    allocated_vm_info * region = (allocated_vm_info*)base_address;
    region[0].base_address = base_address;
    region[0].length = PageTable::PAGE_SIZE;
    vm_regions = region;

    // Increment number of regions
    n_vm_regions = n_vm_regions + 1;

    // Calculate available virtual memory
    available_mem = available_mem - PageTable::PAGE_SIZE;

    Console::puts("Constructed VMPool object.\n");
}
```

vm_pool.C : allocate(): This method allocates a memory region from the virtual memory pool. It begins by comparing the requested allocation size with the available virtual memory pool size. Then, it calculates the number of pages needed for the allocation. The base address and length are determined, and the details of the new virtual memory region are stored in the allocated_vm_info structure, which is added to the array of allocated regions. The available virtual memory pool size is recalculated, and the n_vm_regions variable is incremented. If successful, the method returns the virtual address of the start of the allocated memory region; if it fails, it returns zero.



```
unsigned long VMPool::allocate(unsigned long _size)
{
    unsigned long pages_count = 0;

    // If allocation request size is greater than available virtual memory
    if( _size > available_mem )
    {
        Console::puts("VMPool::allocate - Not enough virtual memory space available.\n");
        assert(false);
    }

    // Calculate number of pages to be allocated
    pages_count = ( _size / PageTable::PAGE_SIZE ) + ( (_size % PageTable::PAGE_SIZE) > 0 ? 1 : 0 );

    // Store details of new virtual memory region
    vm_regions[n_vm_regions].base_address = vm_regions[n_vm_regions-1].base_address +  vm_regions[n_vm_regions-1].length;
    vm_regions[n_vm_regions].length = pages_count * PageTable::PAGE_SIZE;

    // Calculate available memory
    available_mem = available_mem - pages_count * PageTable::PAGE_SIZE;

    // Increment number of virtual memory regions
    n_vm_regions = n_vm_regions + 1;

    Console::puts("Allocated region of memory.\n");

    // Return the allocated base_address
    return vm_regions[n_vm_regions-1].base_address;
}
```

vm_pool.C : release(): This method releases a previously allocated virtual memory region. By using the start address argument and iterating through the allocated virtual memory regions, we identify the region to which the start address belongs. We then calculate the number of pages to be freed and release each page iteratively. The virtual memory allocation information is updated by removing the array element that contains the details of the region to be released. Finally, we recalculate the available virtual memory pool size and decrement the n_vm_regions variable.



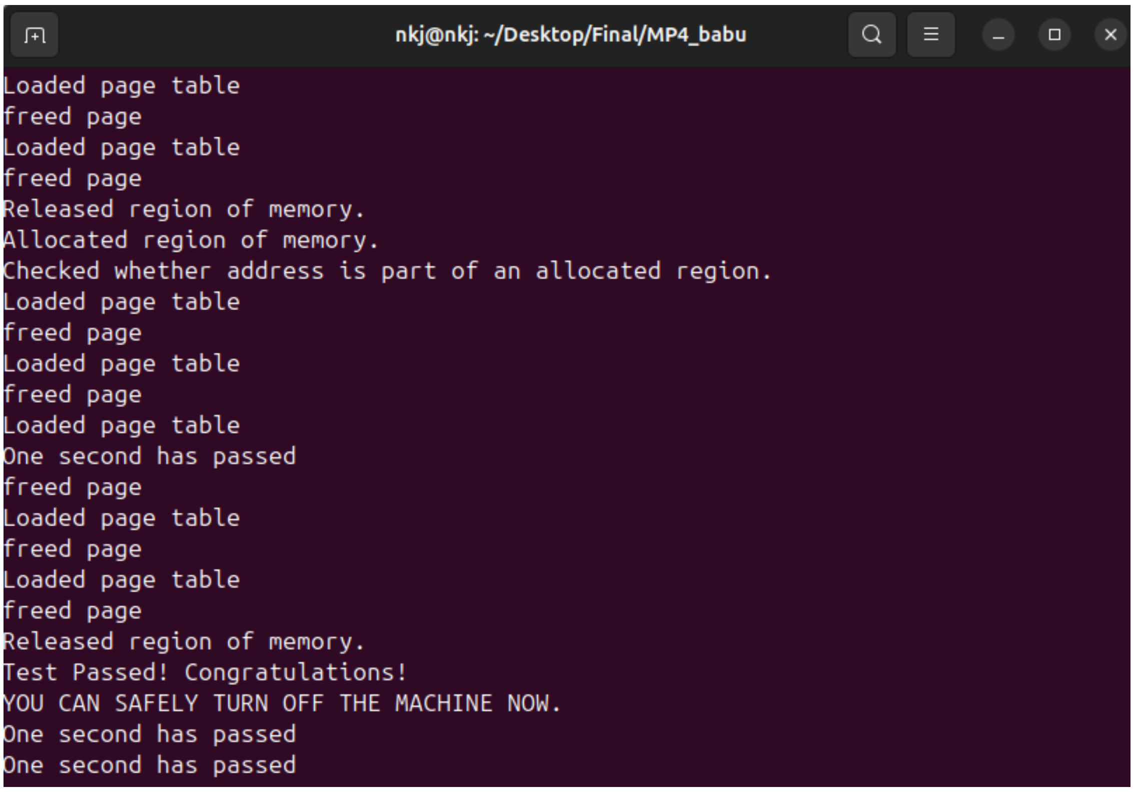
```
void VMPool::release(unsigned long _start_address)
{
    int index = 0;
    int region_no = -1;
    unsigned long page_count = 0;

    // Iterate and find region the start address belongs to
    for(index = 1; index < n_vm_regions; index++ )
    {
        if( vm_regions[index].base_address  == _start_address )
        {
            region_no = index;
        }
    }

    // Calculate number of pages to free
    page_count = vm_regions[region_no].length / PageTable::PAGE_SIZE;
    while( page_count > 0)
    {
        // Free the page
        page_table->free_page(_start_address);

        // Increment start address by page size
        _start_address = _start_address + PageTable::PAGE_SIZE;

        page_count = page_count - 1;
    }

    // Delete virtual memory region information
    for( index = region_no; index < n_vm_regions; index++ )
    {
        vm_regions[index] = vm_regions[index+1];
    }

    // Calculate available memory
    available_mem = available_mem + vm_regions[region_no].length;

    // Decrement number of regions
    n_vm_regions = n_vm_regions - 1;

    Console::puts("Released region of memory.\n");
}
```

vm_pool.C : is_legitimate(): This method is used to check if the address is part of a region that is currently allocated. We check if the address argument exists between the bounds base_address and (base_address + size). On success, it returns true. On failure, it returns false, if the address is not valid.

```
bool VMPool::is_legitimate(unsigned long _address)
{
    Console::puts("Checked whether address is part of an allocated region.\n");

    if( (_address < base_address) || (_address > (base_address + size)) )
    {
        return false;
    }

    return true;
}
```

Testing and Output:

The testing is conducted using Bochs, where the test case invokes page faults and outputs were observed to validate the paging functionality. The test case is tested by modifying **NACCESS** *(#define NACCESS ((1 MB) / 4) and #define NACCESS (2 KB))* values in Kernel.c and for vm pool testing, added extra pools like **VMPool heap_pool1** *(VMPool heap_pool1(2 GB, 256 MB, &process_mem_pool, &pt1) and GenerateVMPoolMemoryReferences(&heap_pool1, 50, 100))* and **VMPool heap_pool2** *(VMPool heap_pool2(2 GB, 256 MB, &process_mem_pool, &pt1) and GenerateVMPoolMemoryReferences(&heap_pool2, 50, 100))* with the same size back-to-back. The test is passed, and outputs were observed like below for all the cases.



```
Loaded page table
freed page
Loaded page table
freed page
Released region of memory.
Allocated region of memory.
Checked whether address is part of an allocated region.
Loaded page table
freed page
Loaded page table
freed page
Loaded page table
One second has passed
freed page
Loaded page table
freed page
Loaded page table
freed page
Released region of memory.
Test Passed! Congratulations!
YOU CAN SAFELY TURN OFF THE MACHINE NOW.
One second has passed
One second has passed
```