MP4: Virtual Memory Management and Memory Allocation

Po-Chieh Wang
UIN: 933004975
CSCE 611: Operating System

Assigned Tasks

Part 1: Completed.
Part 2: Completed.
Part 3: Completed.

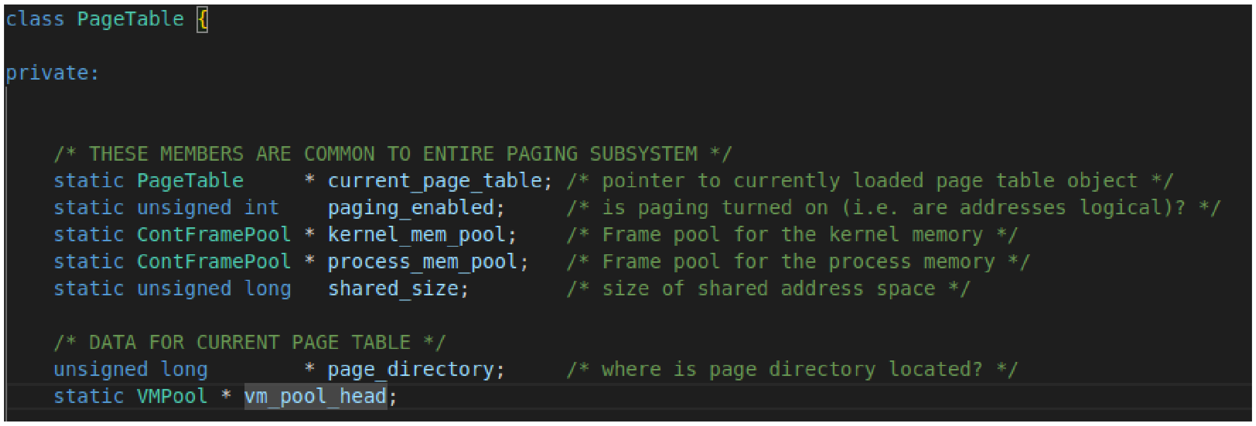
System Design

The goal of this machine problem contains two parts. First, move our page directory and page table from the kernel pool to the process memory pool. After moving the page directory and page table, I implemented a recursive PDE and PTE lookup to let the user modify the content inside the page directory and page table. Lastly, I implemented the virtual memory allocator to allocate the page, release the page and update the page table entry. Inside the virtual memory pool, I used a list to store the information of the corresponding regions. When it starts to allocate a region, it will look up the free regions between allocated regions and select the first big-enough free region to allocate.

Code Description

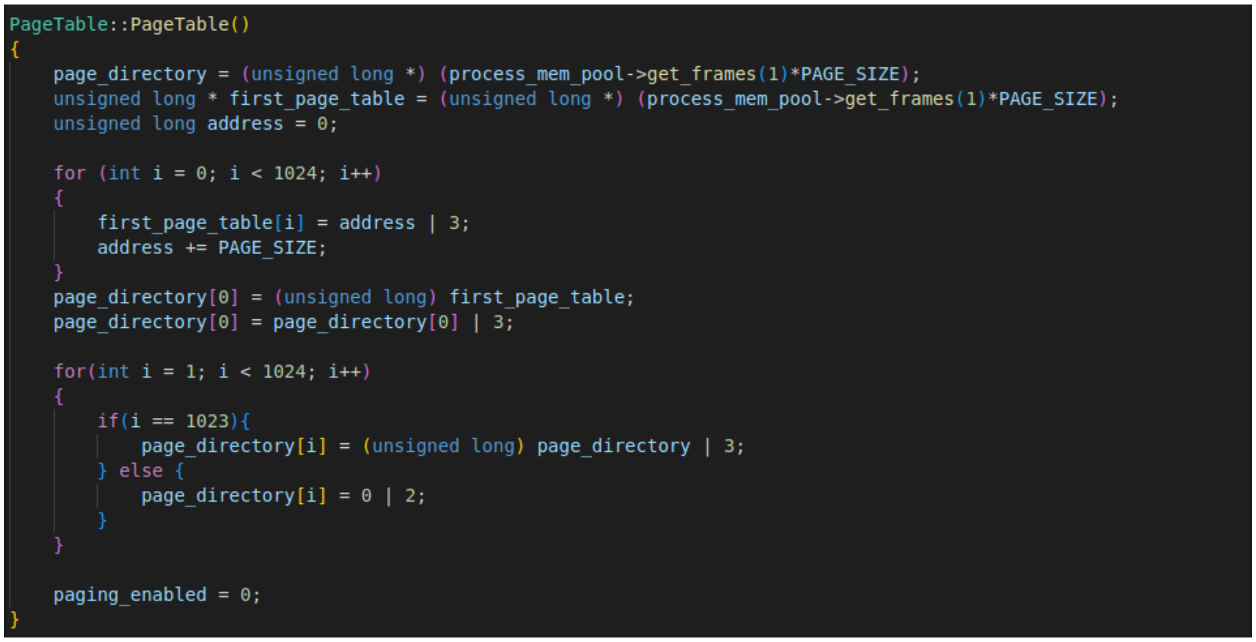
To finish this machine problem, I have changed page_table.C, page_table.H, vm_pool.C and vm_pool.H.

page_table.H: I added a VMPool pointer named head to form the linked list of VMPools.



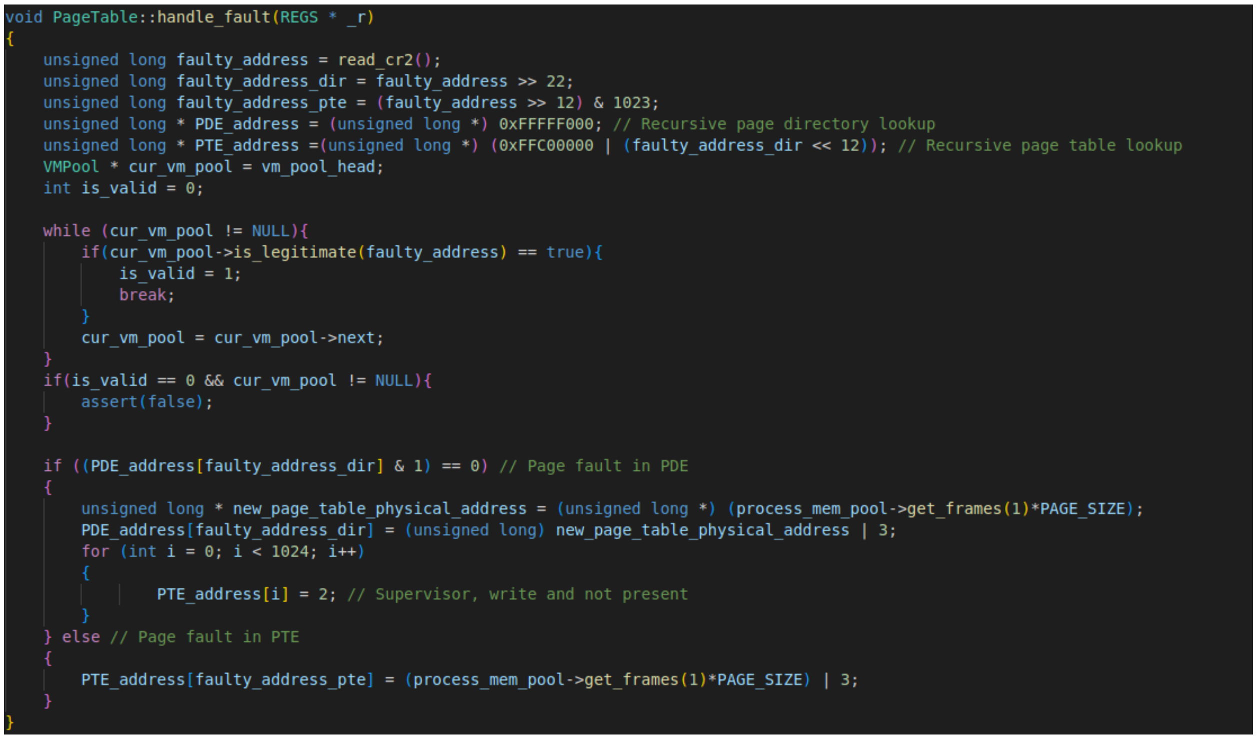
```
class PageTable {

private:


    /* THESE MEMBERS ARE COMMON TO ENTIRE PAGING SUBSYSTEM */
    static PageTable     * current_page_table; /* pointer to currently loaded page table object */
    static unsigned int    paging_enabled;     /* is paging turned on (i.e. are addresses logical)? */
    static ContFramePool * kernel_mem_pool;    /* Frame pool for the kernel memory */
    static ContFramePool * process_mem_pool;   /* Frame pool for the process memory */
    static unsigned long   shared_size;        /* size of shared address space */

    /* DATA FOR CURRENT PAGE TABLE */
    unsigned long        * page_directory;     /* where is page directory located? */
    static VMPool * vm_pool_head;
```

PageTable::PageTable: In this constructor, it will first find a frame in the process frame pool to store the page directory, and a frame to store the first page table, which is the direct-mapped space. It will set every page table entry's last three bits to 011, which means they are in supervisor level, read/ write, and presented. Also, it will set the 1st page directory entry in the page directory to the address of the page table, the last three bits are also 011. The rest of the page directory entries' three bits will be set as 010, which means they are not presented. Lastly, it will set the last page directory entry to itself for recursive PDE and PTE lookup and set paging_enabled to false.



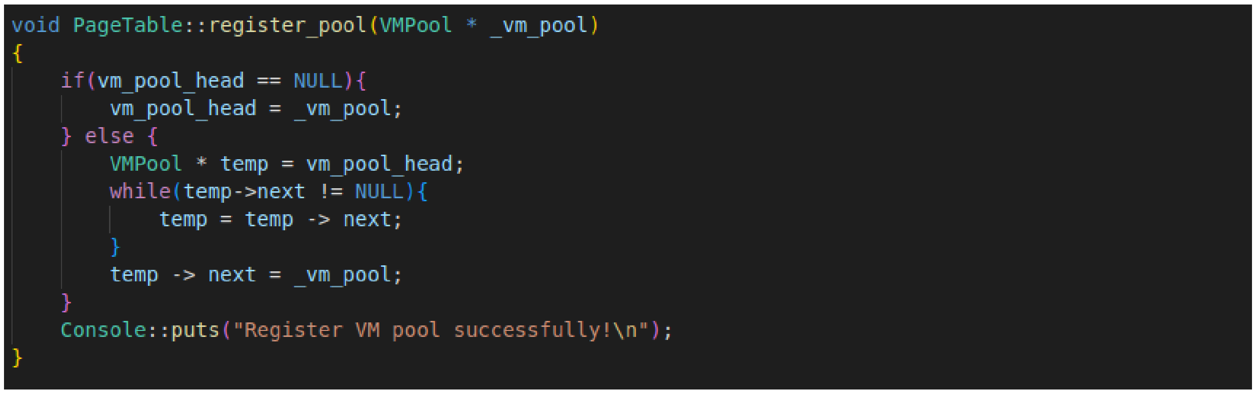
```
PageTable::PageTable()
{
    page_directory = (unsigned long *) (process_mem_pool->get_frames(1)*PAGE_SIZE);
    unsigned long * first_page_table = (unsigned long *) (process_mem_pool->get_frames(1)*PAGE_SIZE);
    unsigned long address = 0;

    for (int i = 0; i < 1024; i++)
    {
        first_page_table[i] = address | 3;
        address += PAGE_SIZE;
    }
    page_directory[0] = (unsigned long) first_page_table;
    page_directory[0] = page_directory[0] | 3;

    for(int i = 1; i < 1024; i++)
    {
        if(i == 1023){
            page_directory[i] = (unsigned long) page_directory | 3;
        } else {
            page_directory[i] = 0 | 2;
        }
    }

    paging_enabled = 0;
}
```

PageTable::handle_fault: This function handles page fault. First, it gets the page fault address by looking at the register cr2. Then it does some bit manipulation to get the logical frame number in the page directory and the logical frame number in the page table. It then saves the logical PDE address and PTE address by recursive PDE and PTE lookup. Furthermore, it checks if this address is legitimate or not by calling the function VMPool:is_legitimate and iterates through the vm pool list. Then, it checks if the page fault happens in the page directory or the page table by looking at the last bit in the page directory entry. If the page directory entry is not presented, the page fault happens in the page directory. It will find a frame in the process pool for the page table and update the page directory entry and page table entries. On the other hand, if a page fault happens in the page table, it will find a frame in the process pool and update the page table entry.



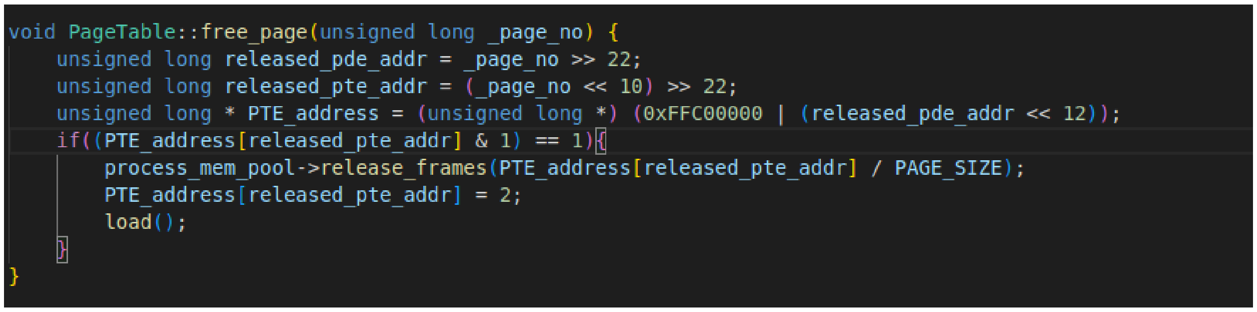
```cpp
void PageTable::handle_fault(REGS * _r)
{
    unsigned long faulty_address = read_cr2();
    unsigned long faulty_address_dir = faulty_address >> 22;
    unsigned long faulty_address_pte = (faulty_address >> 12) & 1023;
    unsigned long * PDE_address = (unsigned long *) 0xFFFFF000; // Recursive page directory lookup
    unsigned long * PTE_address =(unsigned long *) (0xFFC00000 | (faulty_address_dir << 12)); // Recursive page table lookup
    VMPool * cur_vm_pool = vm_pool_head;
    int is_valid = 0;

    while (cur_vm_pool != NULL){
        if(cur_vm_pool->is_legitimate(faulty_address) == true){
            is_valid = 1;
            break;
        }
        cur_vm_pool = cur_vm_pool->next;
    }
    if(is_valid == 0 && cur_vm_pool != NULL){
        assert(false);
    }

    if ((PDE_address[faulty_address_dir] & 1) == 0) // Page fault in PDE
    {
        unsigned long * new_page_table_physical_address = (unsigned long *) (process_mem_pool->get_frames(1)*PAGE_SIZE);
        PDE_address[faulty_address_dir] = (unsigned long) new_page_table_physical_address | 3;
        for (int i = 0; i < 1024; i++)
        {
                PTE_address[i] = 2; // Supervisor, write and not present
        }
    } else // Page fault in PTE
    {
        PTE_address[faulty_address_pte] = (process_mem_pool->get_frames(1)*PAGE_SIZE) | 3;
    }
}
```

PageTable::register_pool: This function creates a linked list of the pools. When there is no pool, the pool which calls this function will become head. If there are some pools in the list, it will connect the pool which calls this function to the end of the linked list.



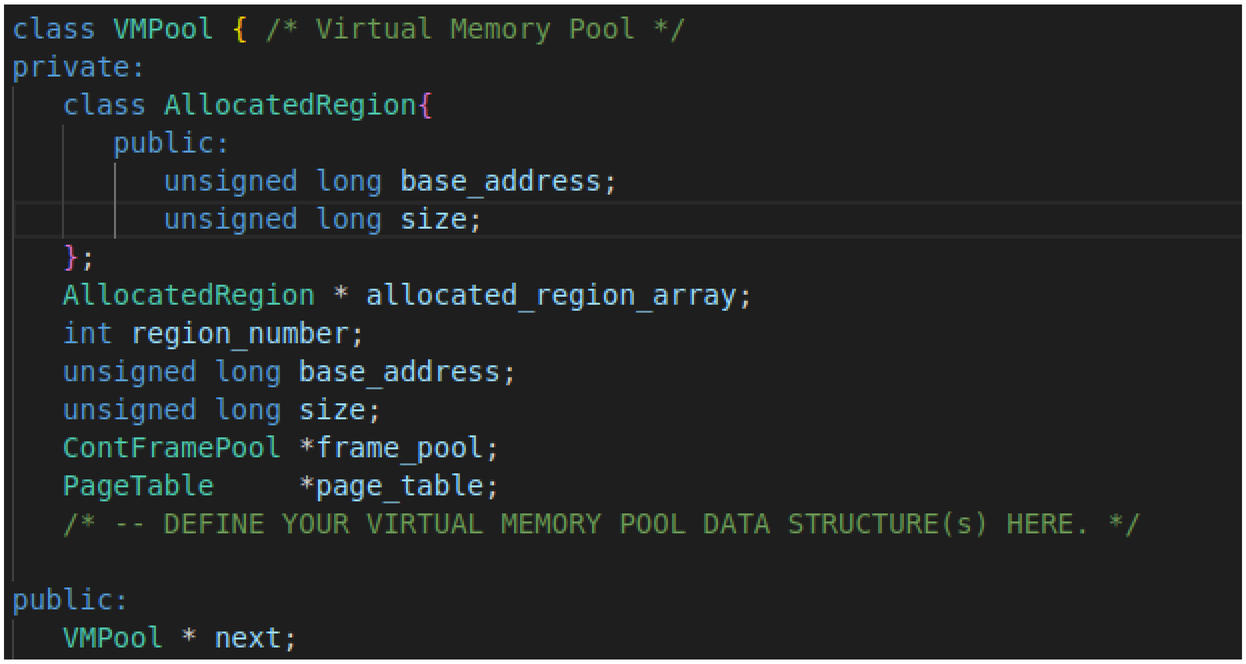
```cpp
void PageTable::register_pool(VMPool * _vm_pool)
{
    if(vm_pool_head == NULL){
        vm_pool_head = _vm_pool;
    } else {
        VMPool * temp = vm_pool_head;
        while(temp->next != NULL){
            temp = temp -> next;
        }
        temp -> next = _vm_pool;
    }
    Console::puts("Register VM pool successfully!\n");
}
```

PageTable::free_page: This function frees the page according to its logical address. It first does bit manipulation to get the PDE and PTE. Then it uses recursive lookup to access the PTE and check if it is valid or not. If it is valid, it calls the release frames function in the frame pool and updates the PTE. Lastly, it flushes the TLB by using load() function.



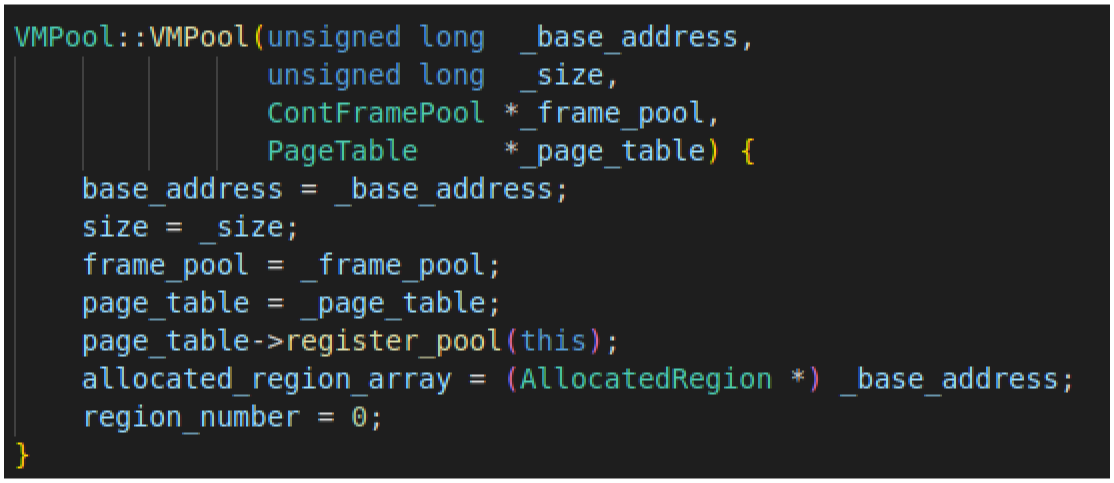
```
void PageTable::free_page(unsigned long _page_no) {
    unsigned long released_pde_addr = _page_no >> 22;
    unsigned long released_pte_addr = (_page_no << 10) >> 22;
    unsigned long * PTE_address = (unsigned long *) (0xFFC00000 | (released_pde_addr << 12));
    if((PTE_address[released_pte_addr] & 1) == 1){
        process_mem_pool->release_frames(PTE_address[released_pte_addr] / PAGE_SIZE);
        PTE_address[released_pte_addr] = 2;
        load();
    }
}
```

vm_pool.H: I added some variables in this class to store some parameters. Also, I added a class named AllocatedRegion to store the information of the corresponding region. This information will be stored in an array called allocated_region_array. Lastly, I added a pointer to point to the next VMPool for handle_fault function.



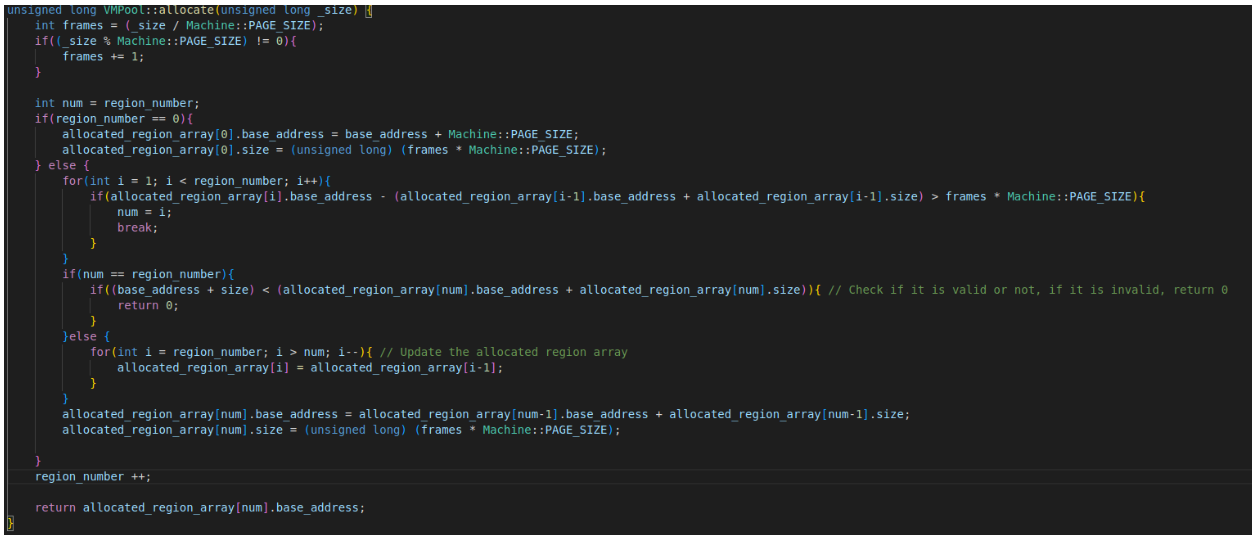
```
class VMPool { /* Virtual Memory Pool */
private:
    class AllocatedRegion{
        public:
            unsigned long base_address;
            unsigned long size;
    };
    AllocatedRegion * allocated_region_array;
    int region_number;
    unsigned long base_address;
    unsigned long size;
    ContFramePool *frame_pool;
    PageTable     *page_table;
    /* -- DEFINE YOUR VIRTUAL MEMORY POOL DATA STRUCTURE(s) HERE. */

public:
    VMPool * next;
```

VMPool::VMPool: In this constructor, it just saves some parameters and register this pool in the page table.



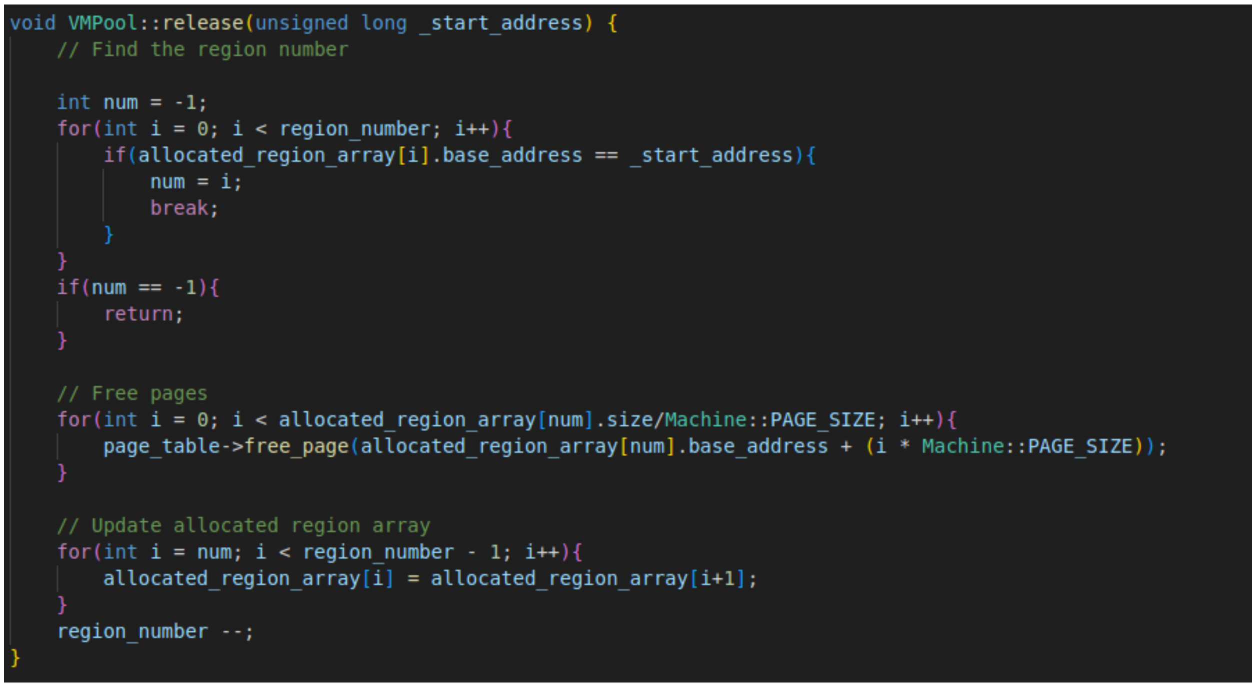
```
VMPool::VMPool(unsigned long  _base_address,
               unsigned long  _size,
               ContFramePool *_frame_pool,
               PageTable     *_page_table) {
    base_address = _base_address;
    size = _size;
    frame_pool = _frame_pool;
    page_table = _page_table;
    page_table->register_pool(this);
    allocated_region_array = (AllocatedRegion *) _base_address;
    region_number = 0;
}
```

VMPool::allocate: This function allocates the frames. It first defines how many frames it needs. After that, it looks up the free regions between the allocated regions. If the free region is big enough, then it will select this region and update the corresponding allocated_region_array which stores the regions' information. Lastly, it increases the number of regions if successful.



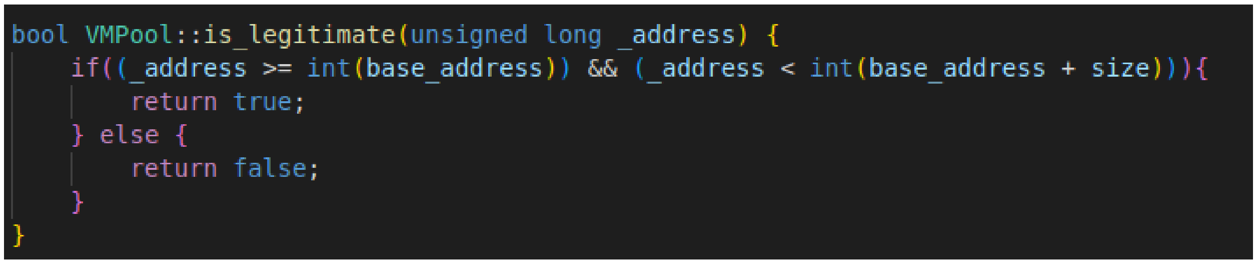
```
unsigned long VMPool::allocate(unsigned long _size) {
    int frames = (_size / Machine::PAGE_SIZE);
    if((_size % Machine::PAGE_SIZE) != 0){
        frames += 1;
    }

    int num = region_number;
    if(region_number == 0){
        allocated_region_array[0].base_address = base_address + Machine::PAGE_SIZE;
        allocated_region_array[0].size = (unsigned long) (frames * Machine::PAGE_SIZE);
    } else {
        for(int i = 1; i < region_number; i++){
            if(allocated_region_array[i].base_address - (allocated_region_array[i-1].base_address + allocated_region_array[i-1].size) > frames * Machine::PAGE_SIZE){
                num = i;
                break;
            }
        }
        if(num == region_number){
            if((base_address + size) < (allocated_region_array[num].base_address + allocated_region_array[num].size)){ // Check if it is valid or not, if it is invalid, return 0
                return 0;
            }
        }else {
            for(int i = region_number; i > num; i--){ // Update the allocated region array
                allocated_region_array[i] = allocated_region_array[i-1];
            }
        }
        allocated_region_array[num].base_address = allocated_region_array[num-1].base_address + allocated_region_array[num-1].size;
        allocated_region_array[num].size = (unsigned long) (frames * Machine::PAGE_SIZE);

    }
    region_number ++;

    return allocated_region_array[num].base_address;
}
```

VMPool::release: This function releases the frames inside this region. It first checks the base address in all regions. If it matches, it releases the frames by calling free_page in PageTable and updates the information of regions in allocated_region_array.



```
void VMPool::release(unsigned long _start_address) {
    // Find the region number

    int num = -1;
    for(int i = 0; i < region_number; i++){
        if(allocated_region_array[i].base_address == _start_address){
            num = i;
            break;
        }
    }
    if(num == -1){
        return;
    }

    // Free pages
    for(int i = 0; i < allocated_region_array[num].size/Machine::PAGE_SIZE; i++){
        page_table->free_page(allocated_region_array[num].base_address + (i * Machine::PAGE_SIZE));
    }

    // Update allocated region array
    for(int i = num; i < region_number - 1; i++){
        allocated_region_array[i] = allocated_region_array[i+1];
    }
    region_number --;
}
```

VMPool::is_legitimate: This function checks if this address is in this vm pool or not by using base address and size.

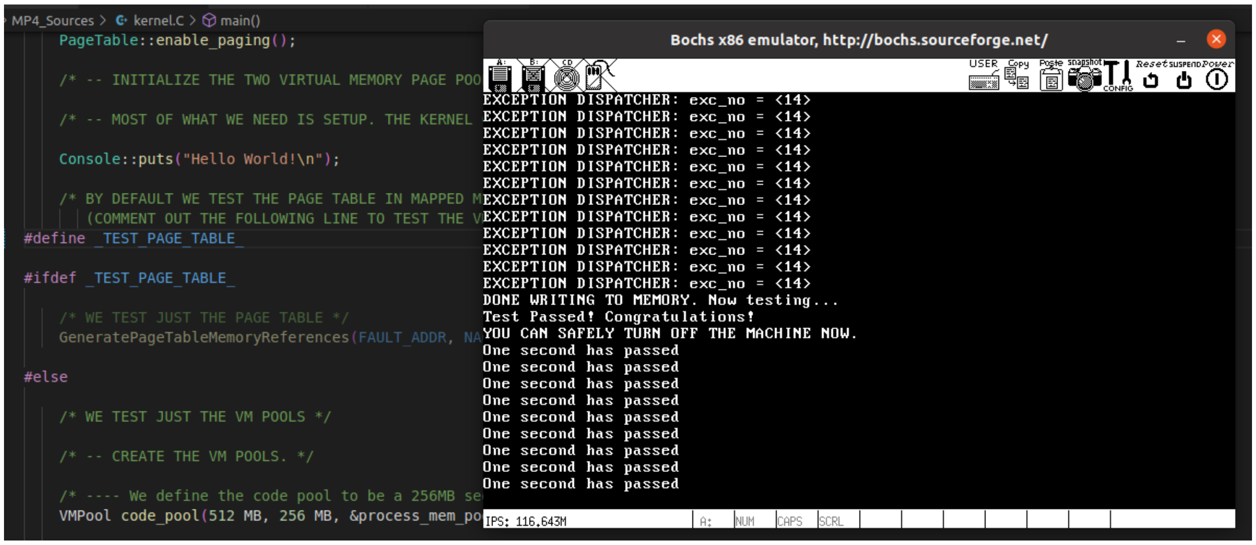


```
bool VMPool::is_legitimate(unsigned long _address) {
    if((_address >= int(base_address)) && (_address < int(base_address + size))){
        return true;
    } else {
        return false;
    }
}
```

Testing

I think the tests provided in Kernel.C are enough for this machine problem, nothing needs to be added. The test results show that the page table management system and the Virtual Memory Allocator I implemented are correct.

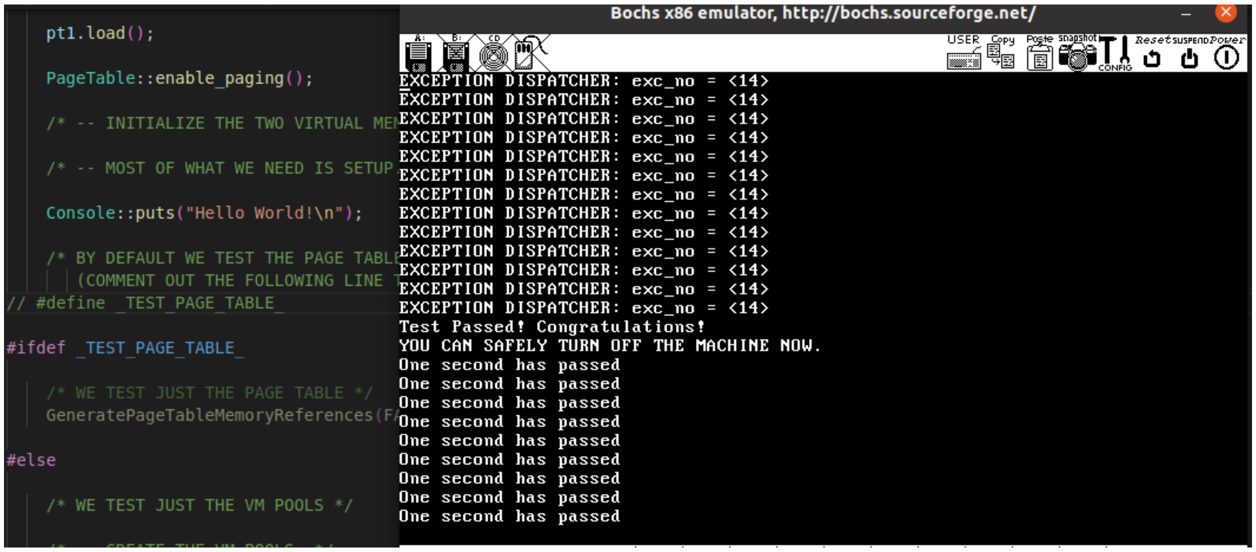
Test Page table:



```
MP4_Sources > kernel.C > main()
    PageTable::enable_paging();

    /* -- INITIALIZE THE TWO VIRTUAL MEMORY PAGE POO

    /* -- MOST OF WHAT WE NEED IS SETUP. THE KERNEL

    Console::puts("Hello World!\n");

    /* BY DEFAULT WE TEST THE PAGE TABLE IN MAPPED M
       (COMMENT OUT THE FOLLOWING LINE TO TEST THE V
#define _TEST_PAGE_TABLE_

#ifdef _TEST_PAGE_TABLE_

    /* WE TEST JUST THE PAGE TABLE */
    GeneratePageTableMemoryReferences(FAULT_ADDR, NA

#else

    /* WE TEST JUST THE VM POOLS */

    /* -- CREATE THE VM POOLS. */

    /* ---- We define the code pool to be a 256MB se
    VMPool code_pool(512 MB, 256 MB, &process_mem_po
```

```
EXCEPTION DISPATCHER: exc_no = <14>
EXCEPTION DISPATCHER: exc_no = <14>
EXCEPTION DISPATCHER: exc_no = <14>
EXCEPTION DISPATCHER: exc_no = <14>
EXCEPTION DISPATCHER: exc_no = <14>
EXCEPTION DISPATCHER: exc_no = <14>
EXCEPTION DISPATCHER: exc_no = <14>
EXCEPTION DISPATCHER: exc_no = <14>
EXCEPTION DISPATCHER: exc_no = <14>
EXCEPTION DISPATCHER: exc_no = <14>
EXCEPTION DISPATCHER: exc_no = <14>
EXCEPTION DISPATCHER: exc_no = <14>
DONE WRITING TO MEMORY. Now testing...
Test Passed! Congratulations!
YOU CAN SAFELY TURN OFF THE MACHINE NOW.
One second has passed
One second has passed
One second has passed
One second has passed
One second has passed
One second has passed
One second has passed
One second has passed
One second has passed
IPS: 116.643M
```

Test VM pool:



```
    pt1.load();

    PageTable::enable_paging();

    /* -- INITIALIZE THE TWO VIRTUAL ME

    /* -- MOST OF WHAT WE NEED IS SETUP

    Console::puts("Hello World!\n");

    /* BY DEFAULT WE TEST THE PAGE TABL
       (COMMENT OUT THE FOLLOWING LINE
// #define _TEST_PAGE_TABLE_

#ifdef _TEST_PAGE_TABLE_

    /* WE TEST JUST THE PAGE TABLE */
    GeneratePageTableMemoryReferences(F

#else

    /* WE TEST JUST THE VM POOLS */
```

```
EXCEPTION DISPATCHER: exc_no = <14>
EXCEPTION DISPATCHER: exc_no = <14>
EXCEPTION DISPATCHER: exc_no = <14>
EXCEPTION DISPATCHER: exc_no = <14>
EXCEPTION DISPATCHER: exc_no = <14>
EXCEPTION DISPATCHER: exc_no = <14>
EXCEPTION DISPATCHER: exc_no = <14>
EXCEPTION DISPATCHER: exc_no = <14>
EXCEPTION DISPATCHER: exc_no = <14>
EXCEPTION DISPATCHER: exc_no = <14>
EXCEPTION DISPATCHER: exc_no = <14>
EXCEPTION DISPATCHER: exc_no = <14>
EXCEPTION DISPATCHER: exc_no = <14>
Test Passed! Congratulations!
YOU CAN SAFELY TURN OFF THE MACHINE NOW.
One second has passed
One second has passed
One second has passed
One second has passed
One second has passed
One second has passed
One second has passed
One second has passed
One second has passed
```