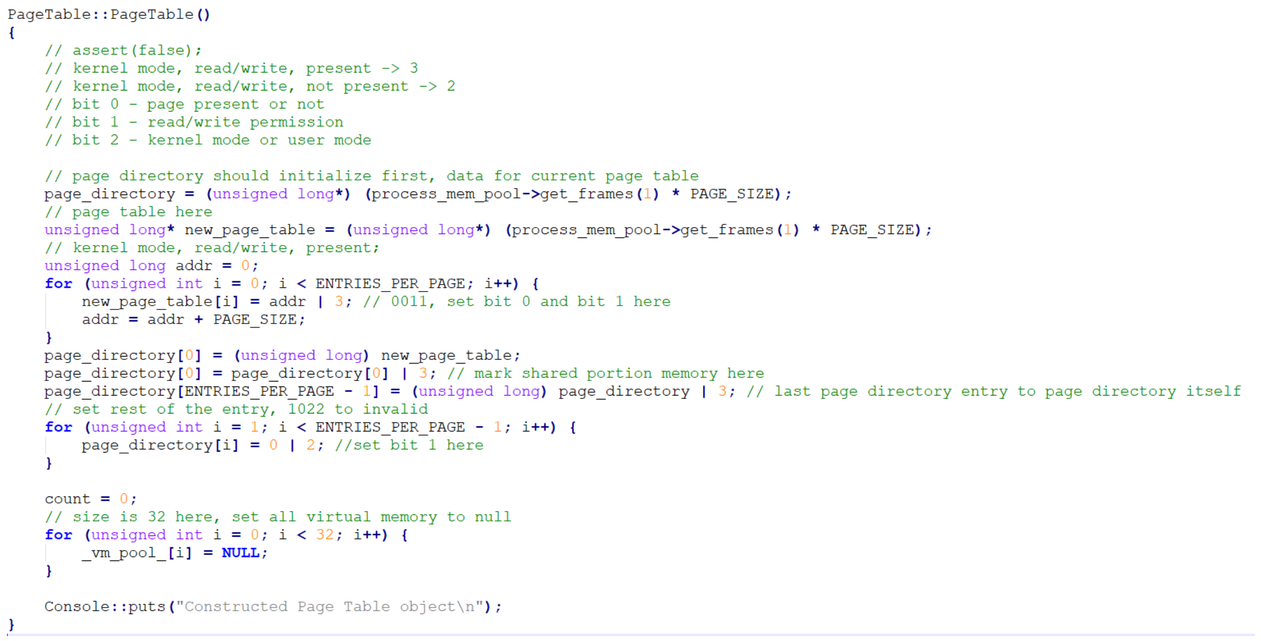
Machine Problem 4: Virtual Memory Management and Memory Allocation

In this machine problem, we basically focus on the three directions. In the first part, we need to do some adjustments in page table source codes to change the directly-mapped kernel memory into virtual memory. Basically, we need to implement recursive page table look up method. In part two, we just need to prepare page table to be ready for supporting virtual memory. In part three, we just need to implement the virtual memory pool.

In this machine problem, I did not change anything from cont_frame_pool source codes, I copied codes from the previous machine problem.

In the page_table.C file, I set up the new page table mechanism to make the last page directory entry point to the page directory itself, and mark all things to valid at the beginning. Also, here I used process_mem_pool instead of kernel_mem_pool.



```
PageTable::PageTable()
{
    // assert(false);
    // kernel mode, read/write, present -> 3
    // kernel mode, read/write, not present -> 2
    // bit 0 - page present or not
    // bit 1 - read/write permission
    // bit 2 - kernel mode or user mode

    // page directory should initialize first, data for current page table
    page_directory = (unsigned long*) (process_mem_pool->get_frames(1) * PAGE_SIZE);
    // page table here
    unsigned long* new_page_table = (unsigned long*) (process_mem_pool->get_frames(1) * PAGE_SIZE);
    // kernel mode, read/write, present;
    unsigned long addr = 0;
    for (unsigned int i = 0; i < ENTRIES_PER_PAGE; i++) {
        new_page_table[i] = addr | 3; // 0011, set bit 0 and bit 1 here
        addr = addr + PAGE_SIZE;
    }
    page_directory[0] = (unsigned long) new_page_table;
    page_directory[0] = page_directory[0] | 3; // mark shared portion memory here
    page_directory[ENTRIES_PER_PAGE - 1] = (unsigned long) page_directory | 3; // last page directory entry to page directory itself
    // set rest of the entry, 1022 to invalid
    for (unsigned int i = 1; i < ENTRIES_PER_PAGE - 1; i++) {
        page_directory[i] = 0 | 2; //set bit 1 here
    }

    count = 0;
    // size is 32 here, set all virtual memory to null
    for (unsigned int i = 0; i < 32; i++) {
        _vm_pool_[i] = NULL;
    }

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

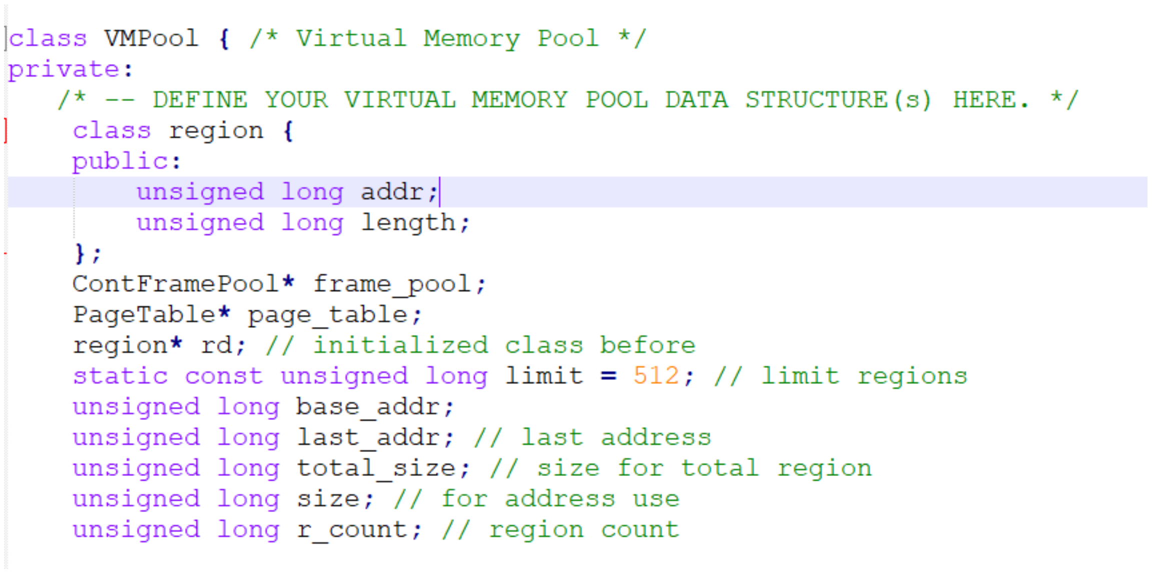
Another change is for handle the page fault, I find the PDE and PTE based on the machine problem hand out, it is the same mechanism like the previous machine problem. If it is a page fault, check if it is in memory. If it is not in memory, just allocate; else, just obtain it directly. The basic logic is the same as the last time.

```
void PageTable::handle_fault(REGS * _r)
{
    // assert(false);
    unsigned long error = _r->err_code;
    if ((error &1) == 1) {
        Console::puts("error handl page fault here!");
        return;
    }
    // marked the format here, 10bit address for page table number, 10bit address for page number, and 12bit address for offset;
    // get current page directory, need to do some slightly change
    // unsigned long* current = (unsigned long*) read_cr3();
    unsigned long* current = (unsigned long*) 0xFFFFF000;
    unsigned long page_fault = read_cr2();
    // get first 10 bit page table number address;
    unsigned long page_table_no = (page_fault >> 22) & 0x3FF; // 10bit AND here, PDE here
    // get second 10 bit address for page number;
    unsigned long page_no = (page_fault >> 12) & 0x3FF; // 10 bit AND here again, PTE here
    unsigned long* page_table;
    if ((current[page_table_no] & 1) == 1) {
        // this shows that page fault already in the memory, do not hit page fault, just put it into page table
        // page_table = (unsigned long*) (current[page_table_no] & 0xFFFFF000);
        page_table = (unsigned long*)  ((page_table_no * PAGE_SIZE) | 0xFFC00000);
    }
    else if ((current[page_table_no] & 1) == 0) {
        // page fault occur, allocate memory
        current[page_table_no] = (process_mem_pool->get_frames(1) * PAGE_SIZE) | 3;
        // page table here
        page_table = (unsigned long*) ((page_table_no * PAGE_SIZE) | 0xFFC00000);
        for (unsigned int i = 0; i < ENTRIES_PER_PAGE; i++) {
            page_table[i] = 0 | 2;
        }
    }

    // page the page table
    // kernel mode, read/write, present here
    page_table[page_no] = (process_mem_pool->get_frames(1) * PAGE_SIZE) | 3;
    Console::puts("handled page fault\n");
}
```

Two new functions in page_table.C are register_pool and free_page. For registering pool for virtual memory, I just set the size 32, and make sure the virtual memory can fit. In the freeing page, I just implement PDE and PTE to call release frame after I obtained their address. After marking page table to invalid, I put flush TLB at the bottom, which should be optional here.

```
void PageTable::register_pool(VMPool * _vm_pool)
{
    // assert(false);
    Console::puts("registered VM pool begin. \n");
    // register the pool, size 32
    if (count < 32) {
        _vm_pool_[count++] = _vm_pool;
    }
    else {
        Console::puts("Cannot register more pool!");
    }
}

void PageTable::free_page(unsigned long _page_no) {
    // assert(false);
    // page table number address, 10 bit, PDE here
    unsigned long page_table_no = (_page_no >> 22) & 0x3FF;
    // page table here
    unsigned long* page_table = (unsigned long*) ((page_table_no * PAGE_SIZE) | 0xFFC00000); // 11111111110000000000000000000000
    // page number, 10 bit, PTE here
    unsigned long page_no = (_page_no >> 12) & 0x3FF;
    // Release the frame
    process_mem_pool->release_frames(page_table[page_no]);
    // not present (invalid) mark
    page_table[page_no] = 0 | 2;
    // load here
    write_cr3(read_cr3());
    Console::puts("freed page\n");
}
```

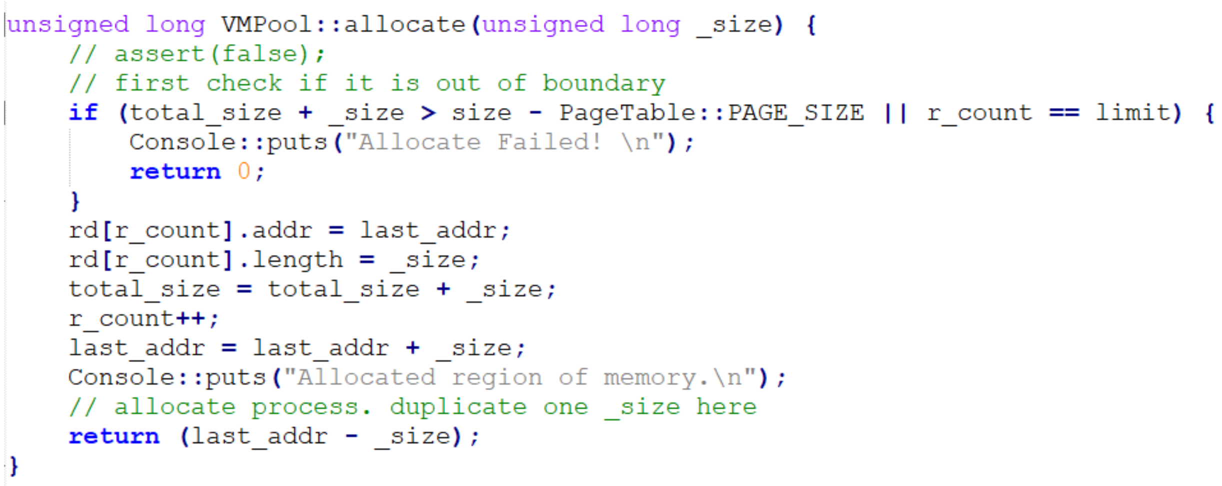
For vm_pool.H, I added some useful parameters in order to help me complete the functions in vm_pool.C more conveniently, which basically I can check the region descriptors with addresses and sizes.



```
class VMPool { /* Virtual Memory Pool */
private:
   /* -- DEFINE YOUR VIRTUAL MEMORY POOL DATA STRUCTURE(s) HERE. */
    class region {
    public:
        unsigned long addr;
        unsigned long length;
    };
    ContFramePool* frame_pool;
    PageTable* page_table;
    region* rd; // initialized class before
    static const unsigned long limit = 512; // limit regions
    unsigned long base_addr;
    unsigned long last_addr; // last address
    unsigned long total_size; // size for total region
    unsigned long size; // for address use
    unsigned long r_count; // region count
```

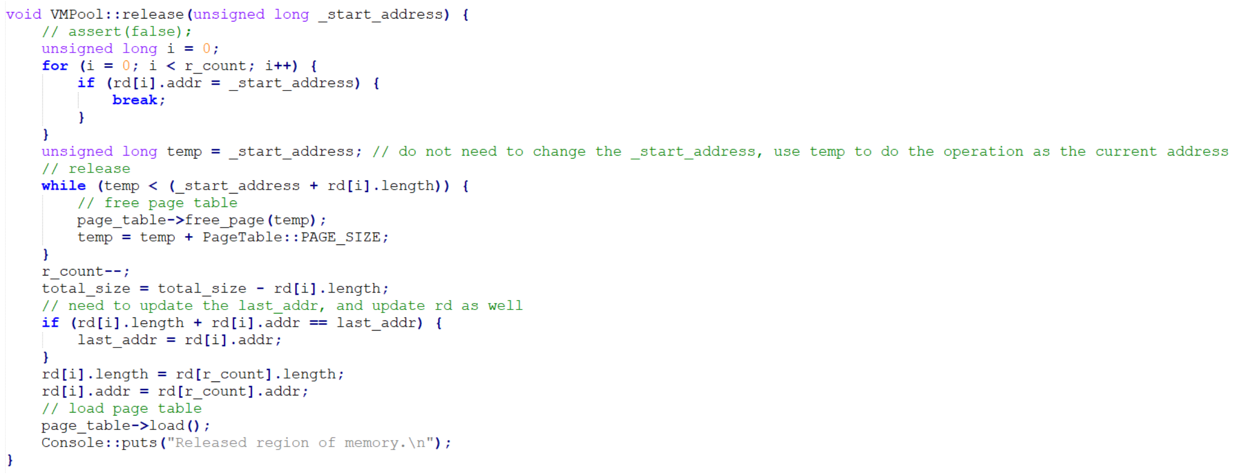
In vm_pool.C, the initialization part is not very annoying, just follow the instruction to assign values correctly and register pool at the beginning.

The major part in this file is allocate and release. In allocate function, I always update region descriptors' address and their size and always check if it exceeds the memory limitation.



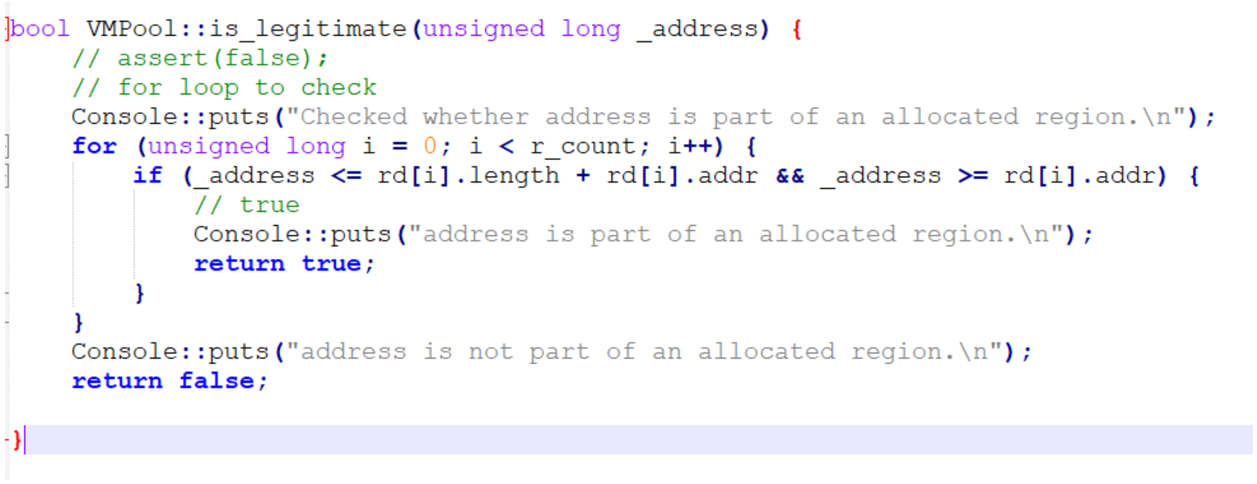
```
unsigned long VMPool::allocate(unsigned long _size) {
    // assert(false);
    // first check if it is out of boundary
    if (total_size + _size > size - PageTable::PAGE_SIZE || r_count == limit) {
        Console::puts("Allocate Failed! \n");
        return 0;
    }
    rd[r_count].addr = last_addr;
    rd[r_count].length = _size;
    total_size = total_size + _size;
    r_count++;
    last_addr = last_addr + _size;
    Console::puts("Allocated region of memory.\n");
    // allocate process. duplicate one _size here
    return (last_addr - _size);
}
```

In the release function, I implemented the free page function from page_table.C, and update the address all the time. Moreover, the length and address of region descriptors need to be updated.



```
void VMPool::release(unsigned long _start_address) {
    // assert(false);
    unsigned long i = 0;
    for (i = 0; i < r_count; i++) {
        if (rd[i].addr = _start_address) {
            break;
        }
    }
    unsigned long temp = _start_address; // do not need to change the _start_address, use temp to do the operation as the current address
    // release
    while (temp < (_start_address + rd[i].length)) {
        // free page table
        page_table->free_page(temp);
        temp = temp + PageTable::PAGE_SIZE;
    }
    r_count--;
    total_size = total_size - rd[i].length;
    // need to update the last_addr, and update rd as well
    if (rd[i].length + rd[i].addr == last_addr) {
        last_addr = rd[i].addr;
    }
    rd[i].length = rd[r_count].length;
    rd[i].addr = rd[r_count].addr;
    // load page table
    page_table->load();
    Console::puts("Released region of memory.\n");
}
```

is_legitimate function is very easy here, I just need to check if region descriptors is in bound, which the address is less than or equal to the address given, and the given address is in the region descriptors' address plus length. Other than that, return false.



```
bool VMPool::is_legitimate(unsigned long _address) {
    // assert(false);
    // for loop to check
    Console::puts("Checked whether address is part of an allocated region.\n");
    for (unsigned long i = 0; i < r_count; i++) {
        if (_address <= rd[i].length + rd[i].addr && _address >= rd[i].addr) {
            // true
            Console::puts("address is part of an allocated region.\n");
            return true;
        }
    }
    Console::puts("address is not part of an allocated region.\n");
    return false;

}
```