MP4: Virtual Memory Management and Memory Allocation

Kai-Chih Huang
UIN: 333000021
CSCE611: Operating System

Assigned Tasks

Part1: Finished
Part2: Finished
Part3: Finished

System Design

The goal of this machine problem is to

Code Description

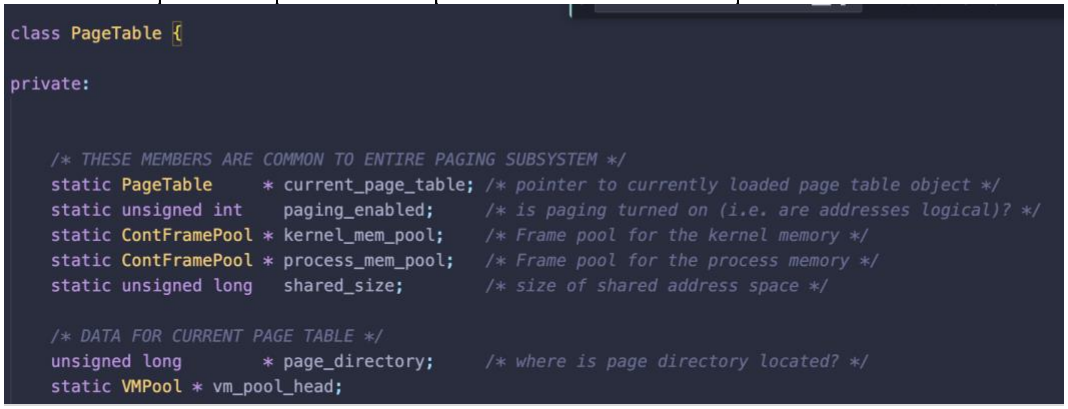
I changed page_table.C, page_table.H, vm_pool.C and vm_pool.H for this machine problem. To compile code, simply run following command lines under MP4_Sources directory:

```
$ make clean    // clean the old compile files before we compile

$ make          // compile files

$ ./copykernel.sh (if permission denied, try chmod u+x ./copykernel.sh then do it again)    //copy kernel

$ bochs -f bochsrc.bxrc     // run bochs
```

I will walk through the functions/methods defined in above files as follows.

page_table.H

I added a vm pool head pointer to keep track of the linked list of pools.

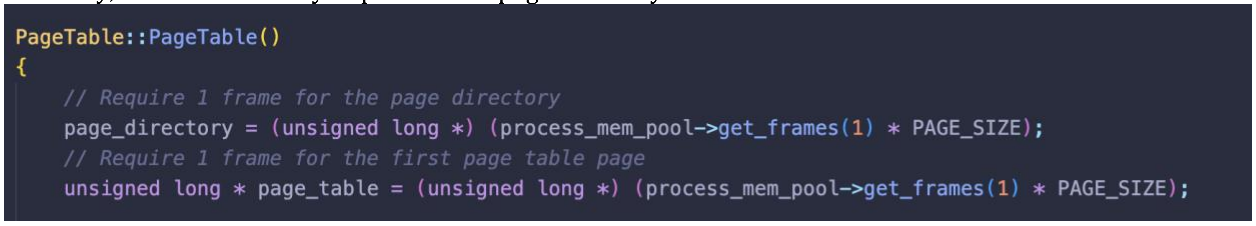


```
class PageTable {

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;
```

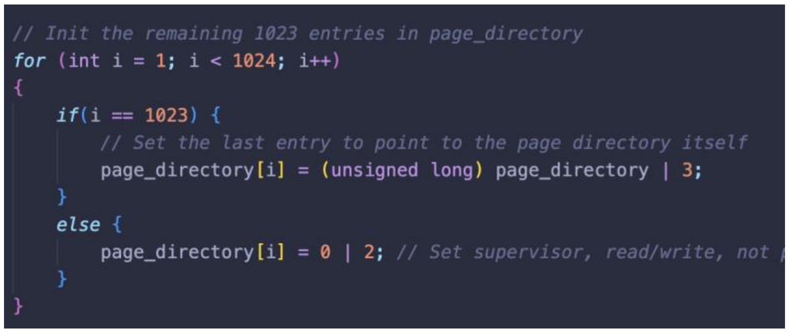
page_table.C

PageTable()

I changed the frame pool source from the kernel to the process mem pool. When initializing the page directory, I set the last entry to point to the page directory itself.



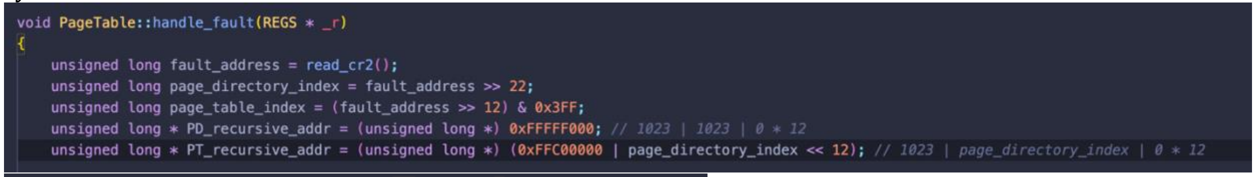
```
PageTable::PageTable()
{
    // Require 1 frame for the page directory
    page_directory = (unsigned long *) (process_mem_pool->get_frames(1) * PAGE_SIZE);
    // Require 1 frame for the first page table page
    unsigned long * page_table = (unsigned long *) (process_mem_pool->get_frames(1) * PAGE_SIZE);
```



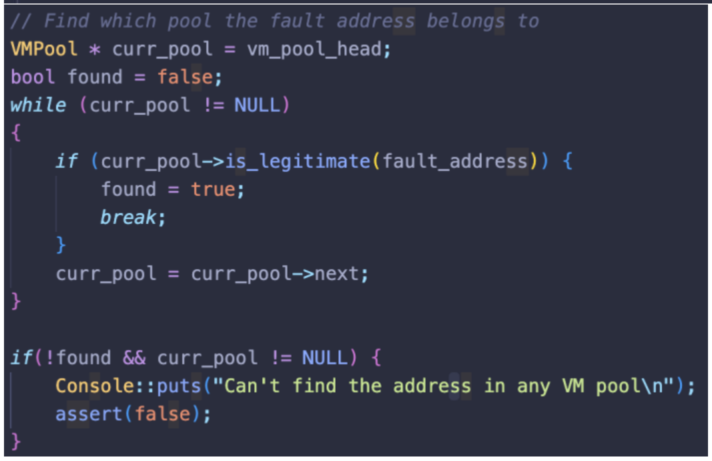
```
// Init the remaining 1023 entries in page_directory
for (int i = 1; i < 1024; i++)
{
    if(i == 1023) {
        // Set the last entry to point to the page directory itself
        page_directory[i] = (unsigned long) page_directory | 3;
    }
    else {
        page_directory[i] = 0 | 2; // Set supervisor, read/write, not
    }
}
```

handle_fault(REGS * _r)

I set the "recursive address" of the page directory and the faulty page table, and use these to replace the former expression in MP3. I also checked if the faulty address is in one of the registered pool. If not, the system will be terminated.



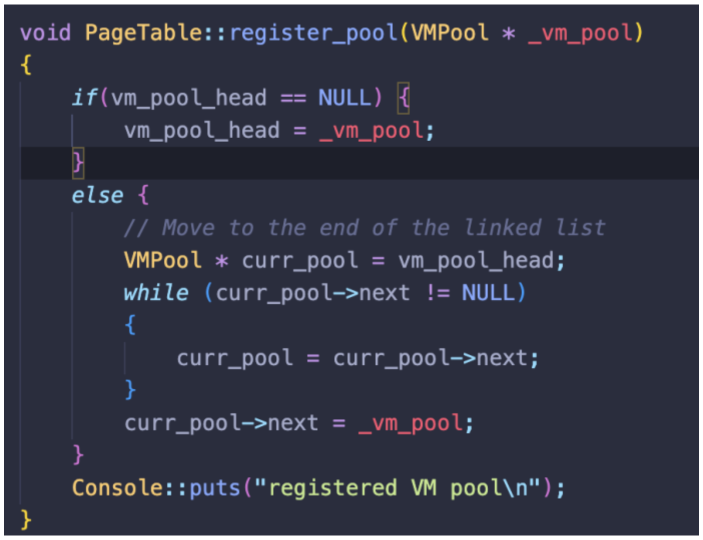
```
void PageTable::handle_fault(REGS * _r)
{
    unsigned long fault_address = read_cr2();
    unsigned long page_directory_index = fault_address >> 22;
    unsigned long page_table_index = (fault_address >> 12) & 0x3FF;
    unsigned long * PD_recursive_addr = (unsigned long *) 0xFFFFF000; // 1023 | 1023 | 0 * 12
    unsigned long * PT_recursive_addr = (unsigned long *) (0xFFC00000 | page_directory_index << 12); // 1023 | page_directory_index | 0 * 12
```



```
// Find which pool the fault address belongs to
VMPool * curr_pool = vm_pool_head;
bool found = false;
while (curr_pool != NULL)
{
    if (curr_pool->is_legitimate(fault_address)) {
        found = true;
        break;
    }
    curr_pool = curr_pool->next;
}

if(!found && curr_pool != NULL) {
    Console::puts("Can't find the address in any VM pool\n");
    assert(false);
}
```

register_pool(VMPool * _vm_pool)

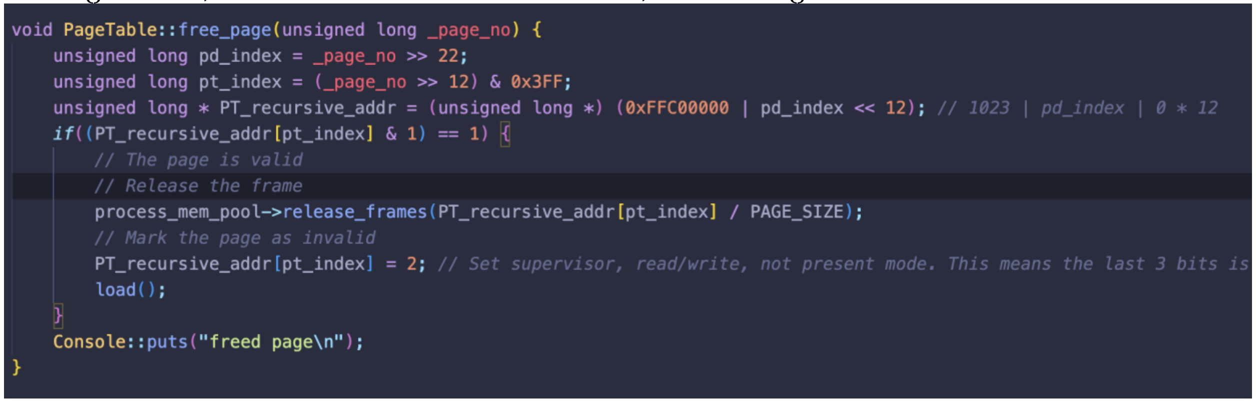
If there is no pool, then set the new pool to the vm_pool_head. If there is existing pool, we simply walk through the linked list and append the new pool to the tail of the list.



```cpp
void PageTable::register_pool(VMPool * _vm_pool)
{
    if(vm_pool_head == NULL) {
        vm_pool_head = _vm_pool;
    }
    else {
        // Move to the end of the linked list
        VMPool * curr_pool = vm_pool_head;
        while (curr_pool->next != NULL)
        {
            curr_pool = curr_pool->next;
        }
        curr_pool->next = _vm_pool;
    }
    Console::puts("registered VM pool\n");
}
```

free_page(unsigned long _page_no)

We use the same way as in handle_fault(REGS * _r) to get the "recursive address" of page table, and the page directory index, page table index as well. If the target PTE is valid, we release the frame by calling release frames() function, then set the PTE to invalid. At the end, we call load() to flush the TLB.

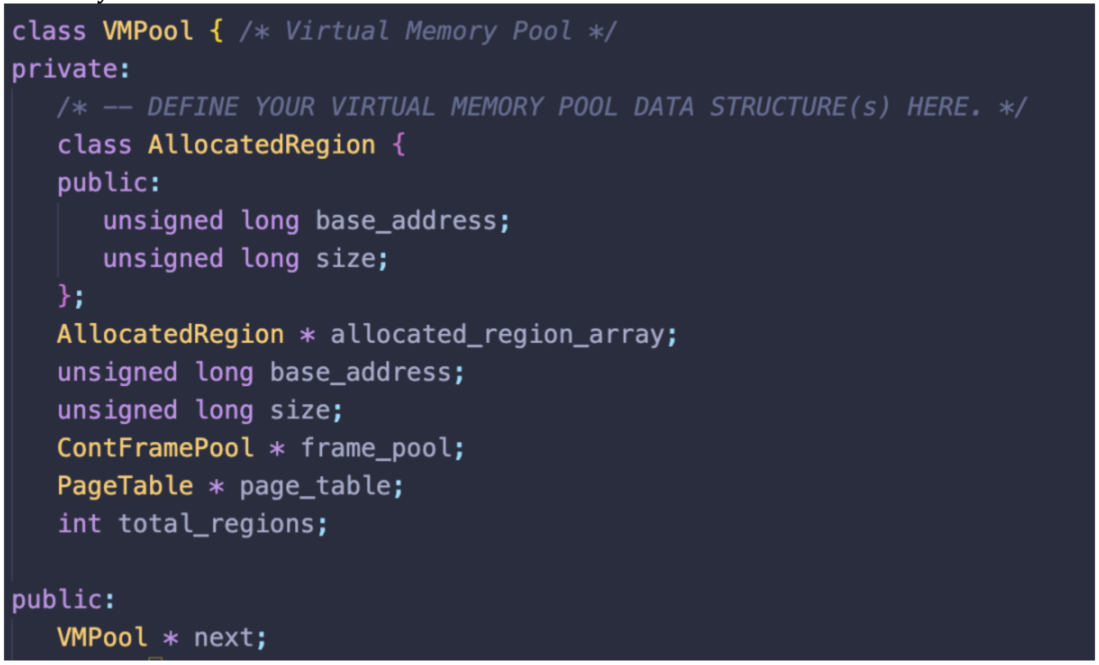


```cpp
void PageTable::free_page(unsigned long _page_no) {
    unsigned long pd_index = _page_no >> 22;
    unsigned long pt_index = (_page_no >> 12) & 0x3FF;
    unsigned long * PT_recursive_addr = (unsigned long *) (0xFFC00000 | pd_index << 12); // 1023 | pd_index | 0 * 12
    if((PT_recursive_addr[pt_index] & 1) == 1) {
        // The page is valid
        // Release the frame
        process_mem_pool->release_frames(PT_recursive_addr[pt_index] / PAGE_SIZE);
        // Mark the page as invalid
        PT_recursive_addr[pt_index] = 2; // Set supervisor, read/write, not present mode. This means the last 3 bits is
        load();
    }
    Console::puts("freed page\n");
}
```

vm_pool.H

In addition to the variables to store input parameters, I added some private variables for the VMPool class. Since we need to manage the allocated "regions" in the pool, I defined an AllocatedRegion object to manage the data of 1 region, which includes its base address and size of the memory in Bytes it takes. Then, I use 1 page to store these AllocatedRegion instances, which I can store 256 regions maximum, and I used an array to manage those regions.
I also added a VMPool pointer to point to the next pool, and total_regions to show how many regions do we

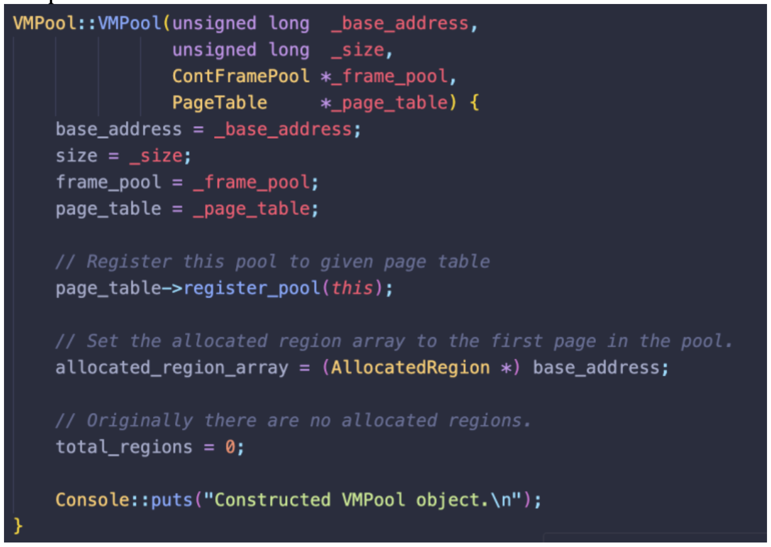
currently have.



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

public:
   VMPool * next;
```

vm_pool.C

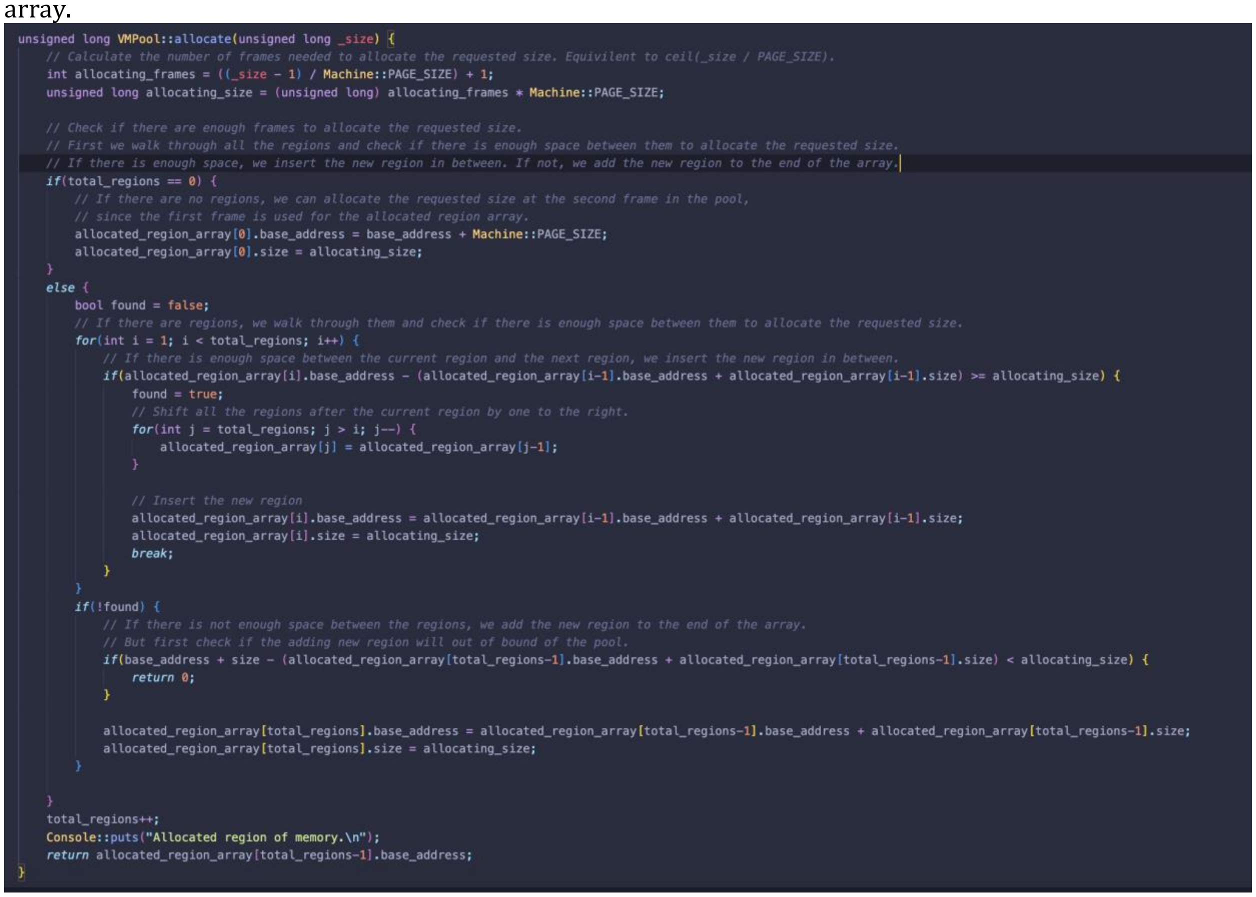
VMPool(unsigned long _base_address, unsigned long _size, ContFramePool *_frame_pool, PageTable _page_table)
To initialize the VMPool, first we need to register the pool to given page table. Then, we put our array that manage allocated regions in the first page, so the address of the array will be the same as the base address of the pool.



```
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;

    // Register this pool to given page table
    page_table->register_pool(this);

    // Set the allocated region array to the first page in the pool.
    allocated_region_array = (AllocatedRegion *) base_address;

    // Originally there are no allocated regions.
    total_regions = 0;

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

allocate(unsigned long _size)
To allocate the frames, it first calculated the required number of frames based on the required memory size. Then, we walk through all the regions and check if there is enough space between them to allocate the requested size. If there is enough space, we insert the new region in between. If not, we add the new region to the end of the array. If there is not enough space between the regions, we add the new region to the end of the

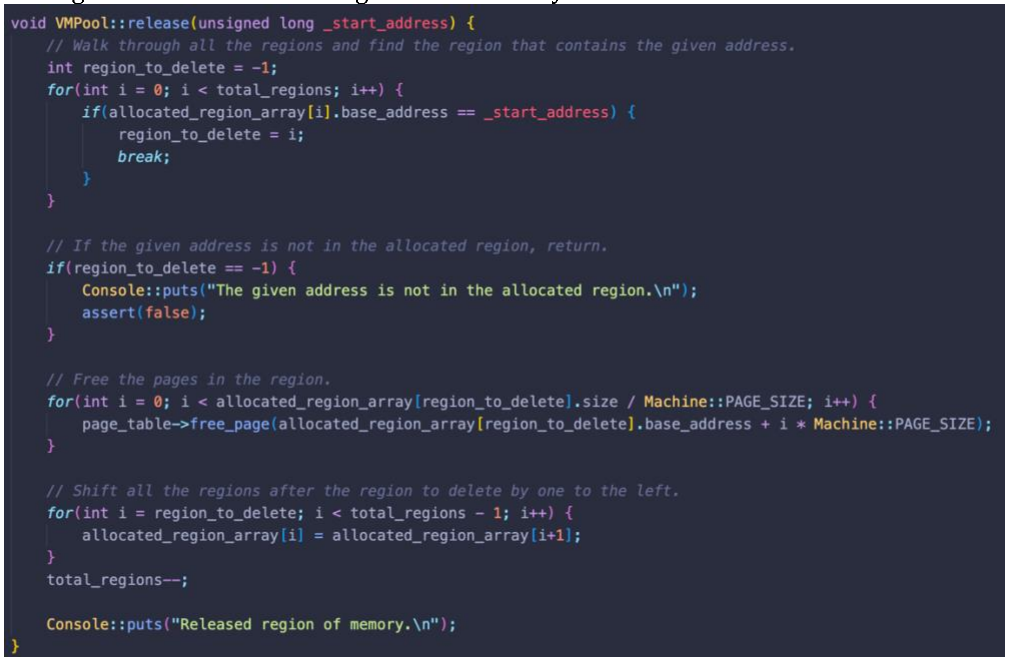
array.



```
unsigned long VMPool::allocate(unsigned long _size) {
    // Calculate the number of frames needed to allocate the requested size. Equivilent to ceil(_size / PAGE_SIZE).
    int allocating_frames = ((_size - 1) / Machine::PAGE_SIZE) + 1;
    unsigned long allocating_size = (unsigned long) allocating_frames * Machine::PAGE_SIZE;

    // Check if there are enough frames to allocate the requested size.
    // First we walk through all the regions and check if there is enough space between them to allocate the requested size.
    // If there is enough space, we insert the new region in between. If not, we add the new region to the end of the array.
    if(total_regions == 0) {
        // If there are no regions, we can allocate the requested size at the second frame in the pool,
        // since the first frame is used for the allocated region array.
        allocated_region_array[0].base_address = base_address + Machine::PAGE_SIZE;
        allocated_region_array[0].size = allocating_size;
    }
    else {
        bool found = false;
        // If there are regions, we walk through them and check if there is enough space between them to allocate the requested size.
        for(int i = 1; i < total_regions; i++) {
            // If there is enough space between the current region and the next region, we insert the new region in between.
            if(allocated_region_array[i].base_address - (allocated_region_array[i-1].base_address + allocated_region_array[i-1].size) >= allocating_size) {
                found = true;
                // Shift all the regions after the current region by one to the right.
                for(int j = total_regions; j > i; j--) {
                    allocated_region_array[j] = allocated_region_array[j-1];
                }

                // Insert the new region
                allocated_region_array[i].base_address = allocated_region_array[i-1].base_address + allocated_region_array[i-1].size;
                allocated_region_array[i].size = allocating_size;
                break;
            }
        }
        if(!found) {
            // If there is not enough space between the regions, we add the new region to the end of the array.
            // But first check if the adding new region will out of bound of the pool.
            if(base_address + size - (allocated_region_array[total_regions-1].base_address + allocated_region_array[total_regions-1].size) < allocating_size) {
                return 0;
            }

            allocated_region_array[total_regions].base_address = allocated_region_array[total_regions-1].base_address + allocated_region_array[total_regions-1].size;
            allocated_region_array[total_regions].size = allocating_size;
        }

    }
    total_regions++;
    Console::puts("Allocated region of memory.\n");
    return allocated_region_array[total_regions-1].base_address;
}
```

release(unsigned long _start_address)

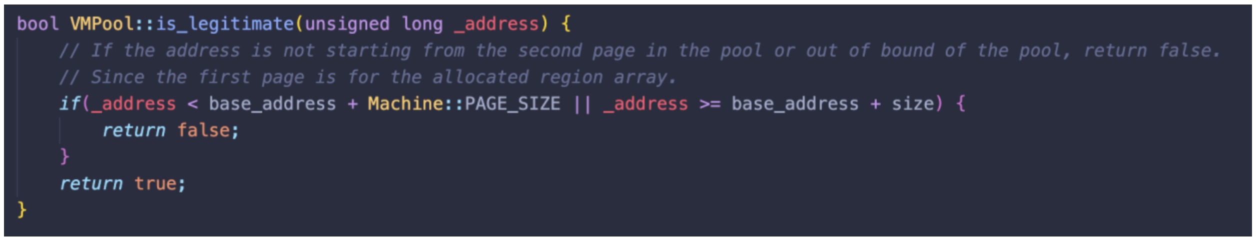
The function release the region that start with given address. Therefore, we simply walk through all the regions and find the region that contains the given address. Then, we free the pages in the region, and shift the regions after the deleted regions to the left by 1.



```
void VMPool::release(unsigned long _start_address) {
    // Walk through all the regions and find the region that contains the given address.
    int region_to_delete = -1;
    for(int i = 0; i < total_regions; i++) {
        if(allocated_region_array[i].base_address == _start_address) {
            region_to_delete = i;
            break;
        }
    }

    // If the given address is not in the allocated region, return.
    if(region_to_delete == -1) {
        Console::puts("The given address is not in the allocated region.\n");
        assert(false);
    }

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

    // Shift all the regions after the region to delete by one to the left.
    for(int i = region_to_delete; i < total_regions - 1; i++) {
        allocated_region_array[i] = allocated_region_array[i+1];
    }
    total_regions--;

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

is_legitimate(unsigned long _address)

The function checks if the given address is legitimate in this VMPool. Since the first page in the pool is for the allocated_region_array, the valid area start from the second page of the pool, and not larger than base_address + size.

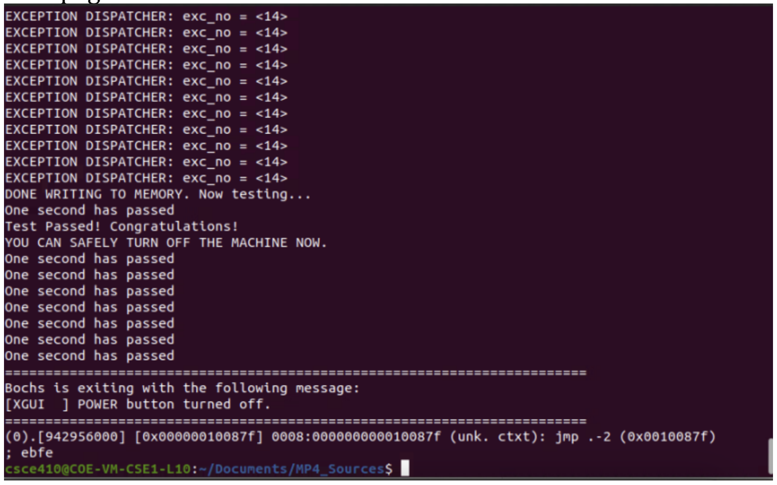


```
bool VMPool::is_legitimate(unsigned long _address) {
    // If the address is not starting from the second page in the pool or out of bound of the pool, return false.
    // Since the first page is for the allocated region array.
    if(_address < base_address + Machine::PAGE_SIZE || _address >= base_address + size) {
        return false;
    }
    return true;
}
```

Testing

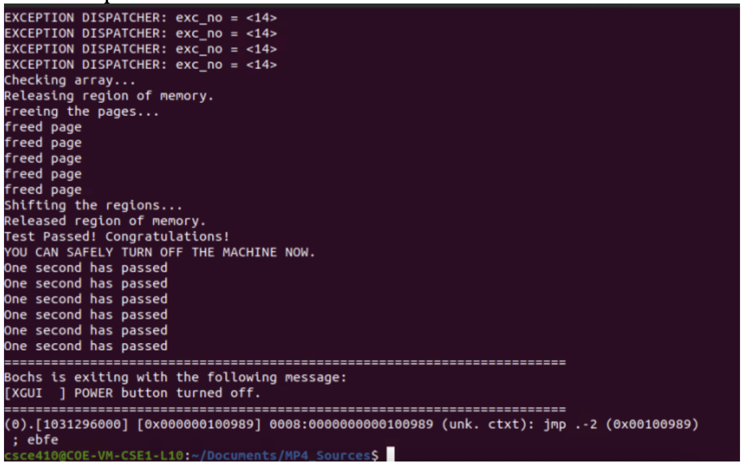
I did not add additional test data in this MP4, since I believe the provided test set is sufficient to check if the page table system works as expected. The test result is shown as follows.

Test page table



```
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...
One second has passed
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
========================================================================
Bochs is exiting with the following message:
[XGUI  ] POWER button turned off.
========================================================================
(0).[942956000] [0x00000010087f] 0008:000000000010087f (unk. ctxt): jmp .-2 (0x0010087f)
; ebfe
csce410@COE-VM-CSE1-L10:~/Documents/MP4_Sources$ 
```

Test VM pool



```
EXCEPTION DISPATCHER: exc_no = <14>
EXCEPTION DISPATCHER: exc_no = <14>
EXCEPTION DISPATCHER: exc_no = <14>
EXCEPTION DISPATCHER: exc_no = <14>
Checking array...
Releasing region of memory.
Freeing the pages...
freed page
freed page
freed page
freed page
freed page
Shifting the regions...
Released region of memory.
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
========================================================================
Bochs is exiting with the following message:
[XGUI  ] POWER button turned off.
========================================================================
(0).[1031296000] [0x000000100989] 0008:0000000000100989 (unk. ctxt): jmp .-2 (0x00100989)
 ; ebfe
csce410@COE-VM-CSE1-L10:~/Documents/MP4_Sources$ 
```