MP4: Page Table II

Sathvik Kote Revanasiddappa
UIN: 634005706
CSCE611: Operating System

Assigned Tasks

Main: Extending Page Table implemented in MP3 that includes support for Large address space and Virtual Memory Allocator : Completed.

System Design

Objective

In this machine problem we complete our memory manager. For this we extend our solution from previous MPs in three directions:

- Part I: We will extend the page table management to support very large numbers and sizes of address spaces.
- We will prepare the page table to support virtual memory as described in Part III.
- We will implement a simple virtual-memory allocator (similar to the frame pool allocator in a previous MP) and hook it up to the new and delete operators of C++. As a result we will have a flexible simple memory management system that we can use for later machine problems. Most importantly, we will be able to dynamically allocate memory in a fashion that is familiar to us from standard user-level C++ programming.

Code Description

Page Manager Design

To increase the address space support, we have to move page table out of kernel space. This will induce problem in making reference to page directory. To solve this problem, we use recursive page look-up. Additionally, page manager should be able to allocate frames for a given virtual address space and free frames for a given address space.

Below are the steps to implement recursive page table look up.

- Keep the page directory in the kernel space.
- Add the physical address of page directory in the last entry of page directory. This will help in recursive look of page tables.
- Place the page tables in the virtual memory and map their address accordingly in the page directory.

Below are the steps to allocate frames for given virtual memory address.

- Calculate number of frames required for the given size.
- For each frame, map virtual address to the physical address.

- Get a physical frame and Calculate the page table entry value and update it in page table.

Below are the steps to free frames for given virtual memory address.

- Find the base address of each page.
- Invalidate the page table entry.
- Call ContFramePool release_frames to free the physical frame.
- Reload the CR3 register and invalidate TLB entries .

Virtual Manager Design

Virtual manager assigns memory based on request from a pool of virtual memory slots. VM pool requires to maintain a data structure that holds the metadata regarding the memory allocated to each process. It should be able to assign memory and release memory on request. VM pool uses a struct that represents a memory slot that are given by base address and size. It maintains two lists, free list and allocated list. Free list holds the memory slots of all the memory regions which is full space at start. Allocated list holds regions which are allocated. All the metadata related to the VM pool are maintained in the first frame of the VM pool.

Below are the steps to allocate requested size.

- Find a region in free slots list that is more than or equal to the requested size.
- If the fond slot is more than the requested size, then update the base address and size else delete the entry in the free list.
- Use the page tables' allocate method to map the virtual address to the physical address.
- Add the new region to allocated list if successfully. Else say unsuccessful.

Below are the steps to release a region.

- Delete the entry in the allocated list.
- For each frame invalidate the page table entry using page_tabel.free_page.

Implementation Details

The below files are modified. And description gives information regarding the modification.

Mac Support

I have modified the below files to support the functioning of MP3 problem on Mac M2.

- copykernel.sh
- bochsrc.bxrc

cont frame pool.H/C

Updated the contents of both the files from MP2.

page_table.H

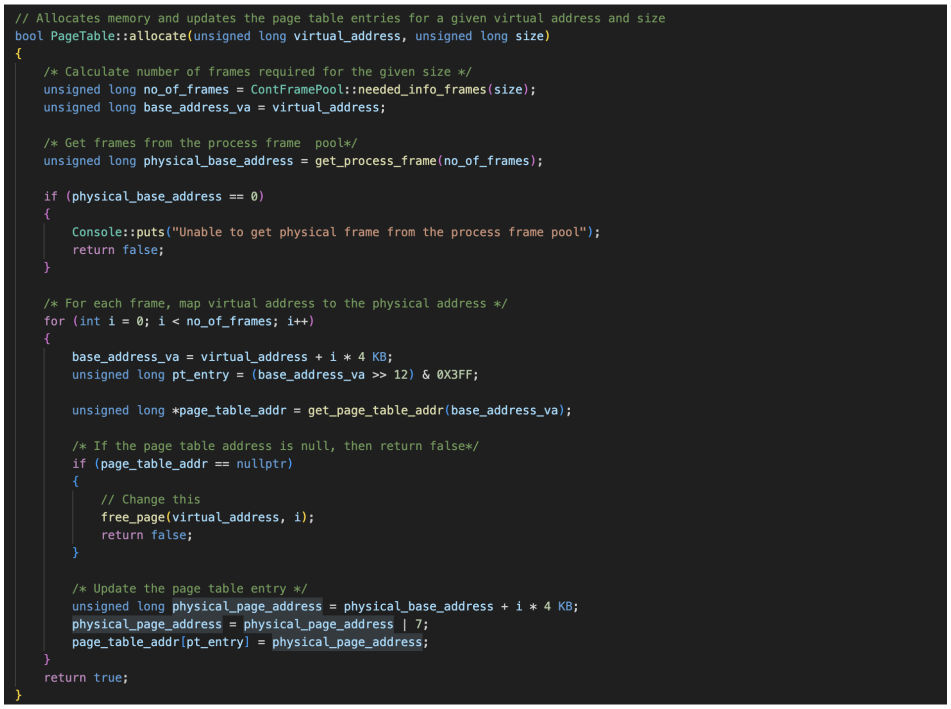
Updated the contents from MP3. And added extra methods to support allocate and free functionality.

- void register_pool(VMPool *_vm_pool): Register a virtual memory pool with the page table.
- void free_page(unsigned long _page_no): If page is valid, release frame and mark page invalid.
- bool allocate(unsigned long virtual_address, unsigned long size): Allocates memory and updates the page table entries for a given virtual address and size.
- unsigned long *get_page_table_addr(unsigned long address): Get address of the page table for the given virtual address.

page_table.C

Updated the contents from MP3. And added extra methods to support allocate and free functionality.

- bool PageTable::allocate(unsigned long virtual_address, unsigned long size): The method follows below steps to implemented the functionality.
  - Calculate the number of frames required for a given size.
  - Using process frame pool, get physical frames allocated for the given memory.
  - For each physical frame, map the virtual and physical address on to the page table entry.
  - If there is error in any part of the execution, return false and free physical frames if assigned.



```cpp
// Allocates memory and updates the page table entries for a given virtual address and size
bool PageTable::allocate(unsigned long virtual_address, unsigned long size)
{
    /* Calculate number of frames required for the given size */
    unsigned long no_of_frames = ContFramePool::needed_info_frames(size);
    unsigned long base_address_va = virtual_address;

    /* Get frames from the process frame  pool*/
    unsigned long physical_base_address = get_process_frame(no_of_frames);

    if (physical_base_address == 0)
    {
        Console::puts("Unable to get physical frame from the process frame pool");
        return false;
    }

    /* For each frame, map virtual address to the physical address */
    for (int i = 0; i < no_of_frames; i++)
    {
        base_address_va = virtual_address + i * 4 KB;
        unsigned long pt_entry = (base_address_va >> 12) & 0X3FF;

        unsigned long *page_table_addr = get_page_table_addr(base_address_va);

        /* If the page table address is null, then return false*/
        if (page_table_addr == nullptr)
        {
            // Change this
            free_page(virtual_address, i);
            return false;
        }

        /* Update the page table entry */
        unsigned long physical_page_address = physical_base_address + i * 4 KB;
        physical_page_address = physical_page_address | 7;
        page_table_addr[pt_entry] = physical_page_address;
    }
    return true;
}
```

Figure 1: Page table allocate

- void PageTable::register_pool(VMPool *_vm_pool): Register virtual manager pool into an array. This is required to check whether the given faulted address belongs to any of the pool. If not, page fault is unhandled.

- void PageTable::free_page(unsigned long _page_no): For a given page, find the page table entry and invalidate it. Additionally, free the physical memory that is assigned to it and reload the CR3 register.

- unsigned long *PageTable::get_page_table_addr(unsigned long address): The function does bit wise operation on the given address. Returns the address of page table page address.

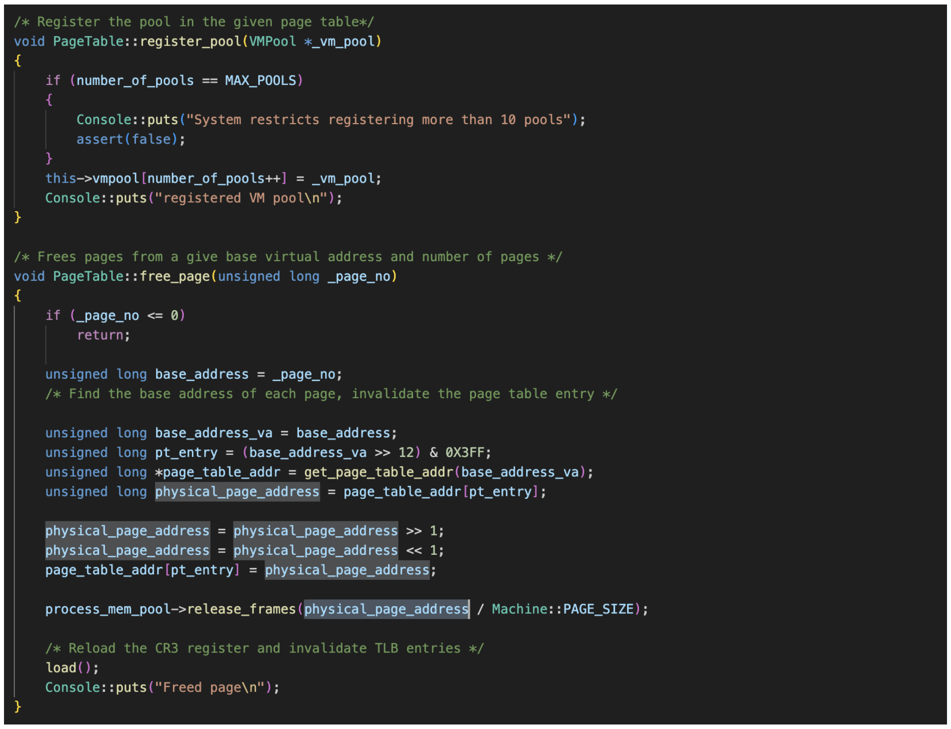
vm_pool.H

- struct node: A datastructure to represent a node in a linked list. It holds the base address and along with size of the memory region. It is used in both free and allocated list.

- unsigned long base_address: Variable to hold the base address

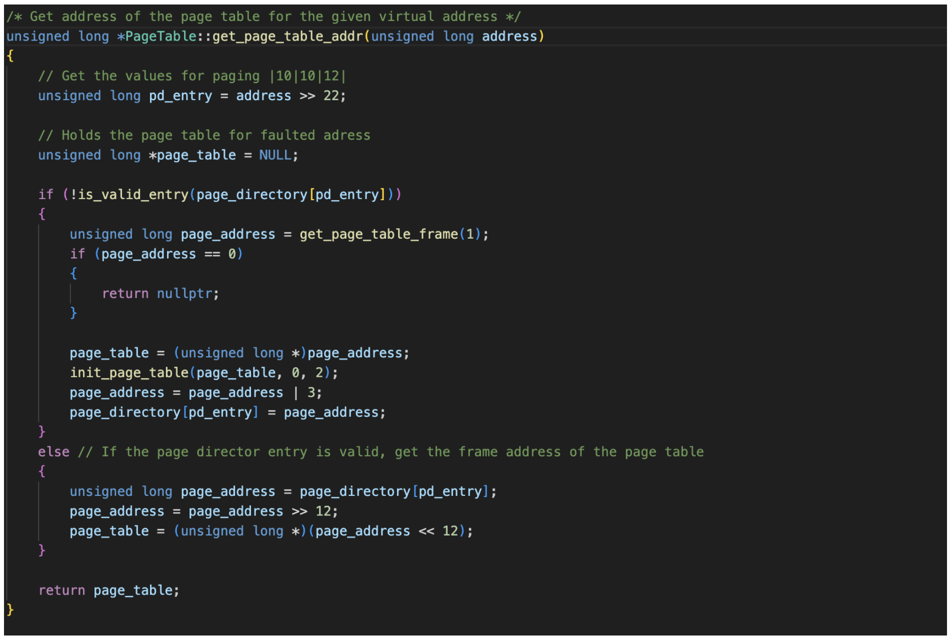
- unsigned long size: Variable to hold the size

- ContFramePool *frame_pool: Variable to hold the frame pool.



```
/* Register the pool in the given page table*/
void PageTable::register_pool(VMPool *_vm_pool)
{
    if (number_of_pools == MAX_POOLS)
    {
        Console::puts("System restricts registering more than 10 pools");
        assert(false);
    }
    this->vmpool[number_of_pools++] = _vm_pool;
    Console::puts("registered VM pool\n");
}

/* Frees pages from a give base virtual address and number of pages */
void PageTable::free_page(unsigned long _page_no)
{
    if (_page_no <= 0)
        return;

    unsigned long base_address = _page_no;
    /* Find the base address of each page, invalidate the page table entry */

    unsigned long base_address_va = base_address;
    unsigned long pt_entry = (base_address_va >> 12) & 0X3FF;
    unsigned long *page_table_addr = get_page_table_addr(base_address_va);
    unsigned long physical_page_address = page_table_addr[pt_entry];

    physical_page_address = physical_page_address >> 1;
    physical_page_address = physical_page_address << 1;
    page_table_addr[pt_entry] = physical_page_address;

    process_mem_pool->release_frames(physical_page_address / Machine::PAGE_SIZE);

    /* Reload the CR3 register and invalidate TLB entries */
    load();
    Console::puts("Freed page\n");
}
```

Figure 2: Page table free and register pool

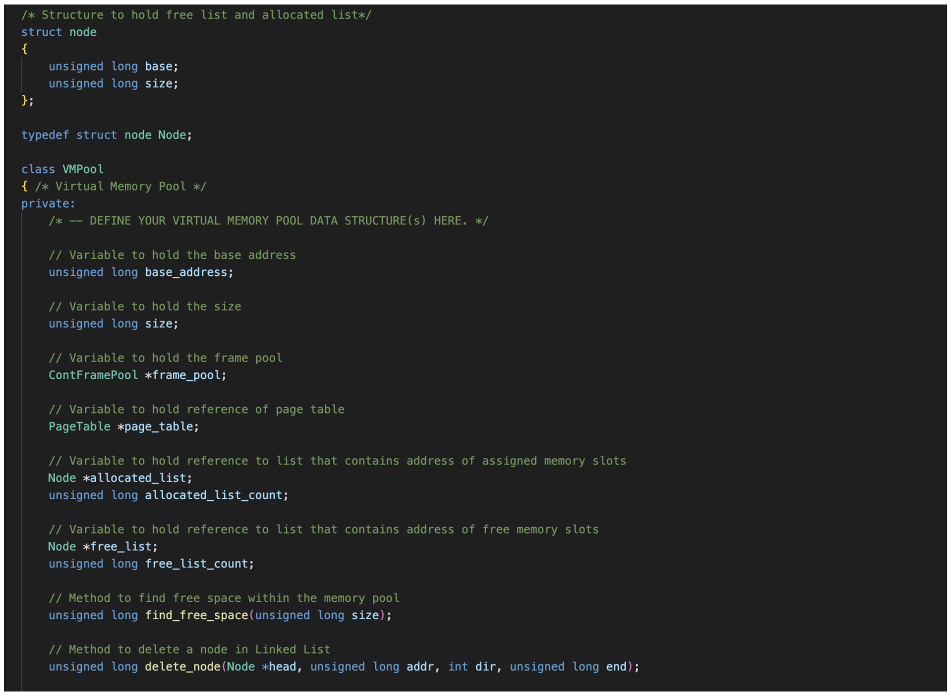


```
/* Get address of the page table for the given virtual address */
unsigned long *PageTable::get_page_table_addr(unsigned long address)
{
    // Get the values for paging |10|10|12|
    unsigned long pd_entry = address >> 22;

    // Holds the page table for faulted adress
    unsigned long *page_table = NULL;

    if (!is_valid_entry(page_directory[pd_entry]))
    {
        unsigned long page_address = get_page_table_frame(1);
        if (page_address == 0)
        {
            return nullptr;
        }

        page_table = (unsigned long *)page_address;
        init_page_table(page_table, 0, 2);
        page_address = page_address | 3;
        page_directory[pd_entry] = page_address;
    }
    else // If the page director entry is valid, get the frame address of the page table
    {
        unsigned long page_address = page_directory[pd_entry];
        page_address = page_address >> 12;
        page_table = (unsigned long *)(page_address << 12);
    }

    return page_table;
}
```

Figure 3: Page table get page table addr

- PageTable *page_table: Variable to hold reference of page table.
- Node *allocated_list: Variable to hold reference to list that contains address of assigned memory slots
- unsigned long allocated_list_count: Holds number of entries in the allocated list.
- Node *free_list: Variable to hold reference to list that contains address of free memory slots.

- unsigned long find_free_space(unsigned long size): Method to find free space within the memory pool
- unsigned long delete_node(Node *head, unsigned long addr, int dir, unsigned long end): Method to delete a node in Linked List
- unsigned long allocate(unsigned long _size): Allocates a region of _size bytes of memory from the virtual memory pool. If successful, returns the virtual address of the start of the allocated region of memory. If fails, returns 0.
- void release(unsigned long _start_address): Releases a region of previously allocated memory. The region is identified by its start address, which was returned when the region was allocated.
- bool is_legitimate(unsigned long _address): Returns false if the address is not valid. An address is not valid if it is not part of a region that is currently allocated.

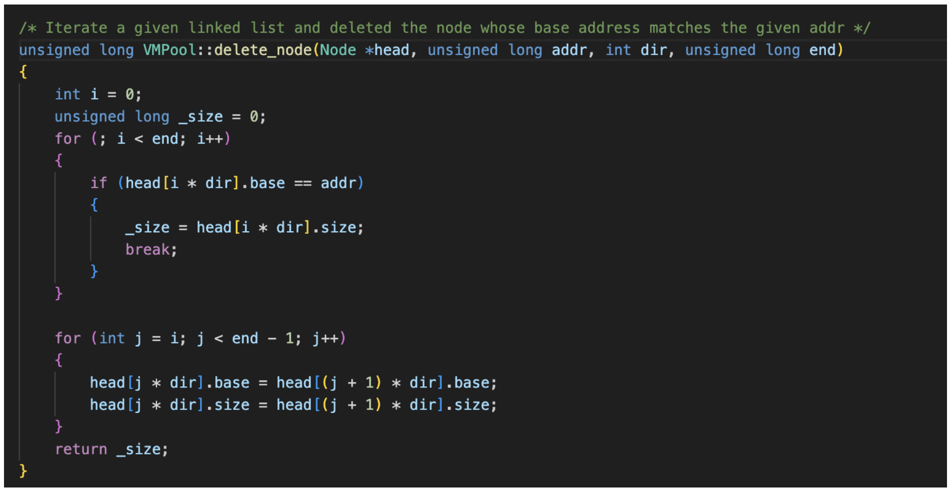


```
/* Structure to hold free list and allocated list*/
struct node
{
    unsigned long base;
    unsigned long size;
};

typedef struct node Node;

class VMPool
{ /* Virtual Memory Pool */
private:
    /* -- DEFINE YOUR VIRTUAL MEMORY POOL DATA STRUCTURE(s) HERE. */

    // Variable to hold the base address
    unsigned long base_address;

    // Variable to hold the size
    unsigned long size;

    // Variable to hold the frame pool
    ContFramePool *frame_pool;

    // Variable to hold reference of page table
    PageTable *page_table;

    // Variable to hold reference to list that contains address of assigned memory slots
    Node *allocated_list;
    unsigned long allocated_list_count;

    // Variable to hold reference to list that contains address of free memory slots
    Node *free_list;
    unsigned long free_list_count;

    // Method to find free space within the memory pool
    unsigned long find_free_space(unsigned long size);

    // Method to delete a node in Linked List
    unsigned long delete_node(Node *head, unsigned long addr, int dir, unsigned long end);
```

Figure 4: VM Pool data structures

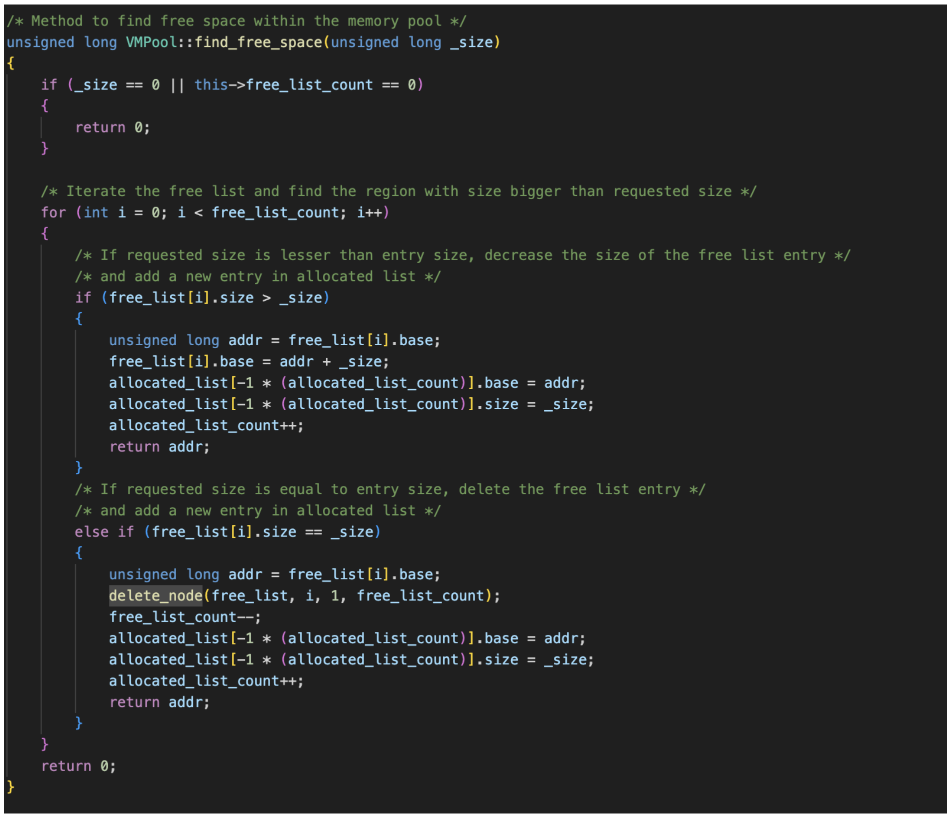
vm_pool.C

- unsigned long VMPool::delete_node(Node *head, unsigned long addr, int dir, unsigned long end): The method iterates through the list and find any nodes that has the base address as addr. The linked list are stored in the first frame of the VM pool. Free list start from base and move in positive direction. Allocated list starts from the end and move in negative direction. Hence, dir gives direction of iteration. END Variable gives the length of the linked list.
- unsigned long VMPool::find_free_space(unsigned long _size)
  - Checks whether the count of nodes in the free list is empty.
  - Iterate through the list and find the region that is bigger than the requested size.
  - Update the list accordingly and return the starting virtual address.
- unsigned long VMPool::allocate(unsigned long _size): The method initialises the free list for the first time. It finds a free region using free space method. It also updates the page table entries. If anything is failed, the whole process is rolled back.



```
/* Iterate a given linked list and deleted the node whose base address matches the given addr */
unsigned long VMPool::delete_node(Node *head, unsigned long addr, int dir, unsigned long end)
{
    int i = 0;
    unsigned long _size = 0;
    for (; i < end; i++)
    {
        if (head[i * dir].base == addr)
        {
            _size = head[i * dir].size;
            break;
        }
    }

    for (int j = i; j < end - 1; j++)
    {
        head[j * dir].base = head[(j + 1) * dir].base;
        head[j * dir].size = head[(j + 1) * dir].size;
    }
    return _size;
}
```

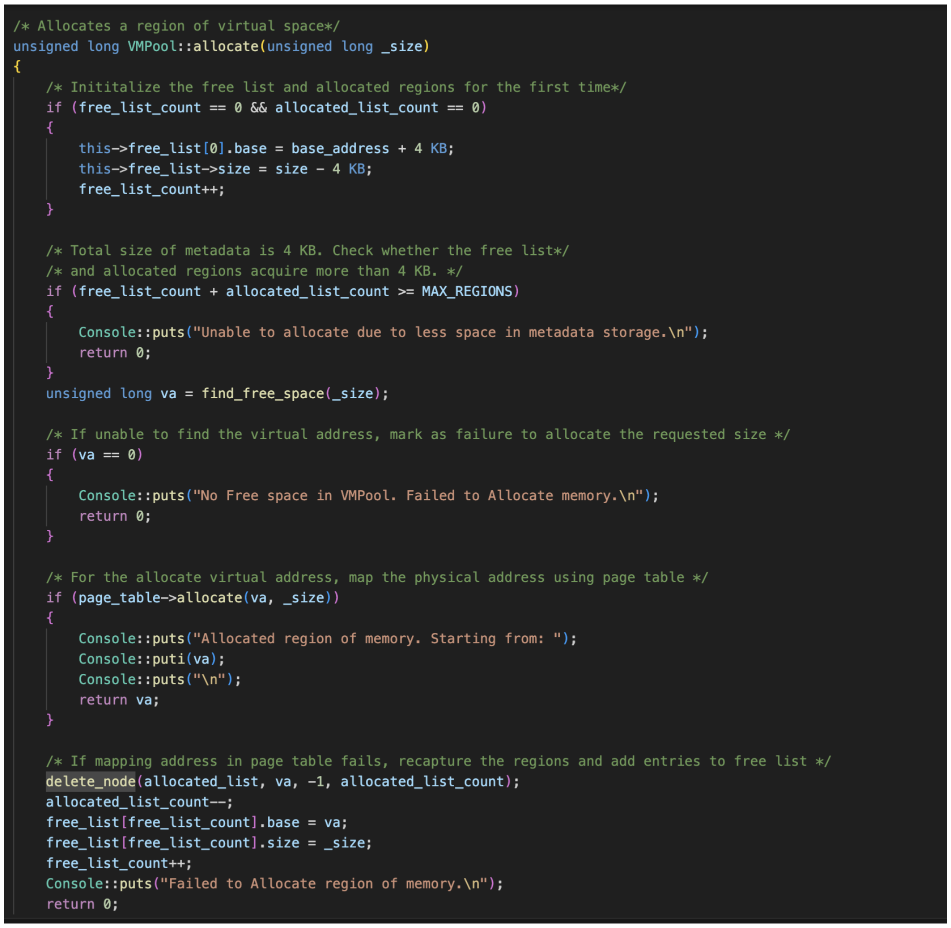
Figure 5: VM Pool Delete Node



```
/* Method to find free space within the memory pool */
unsigned long VMPool::find_free_space(unsigned long _size)
{
    if (_size == 0 || this->free_list_count == 0)
    {
        return 0;
    }

    /* Iterate the free list and find the region with size bigger than requested size */
    for (int i = 0; i < free_list_count; i++)
    {
        /* If requested size is lesser than entry size, decrease the size of the free list entry */
        /* and add a new entry in allocated list */
        if (free_list[i].size > _size)
        {
            unsigned long addr = free_list[i].base;
            free_list[i].base = addr + _size;
            allocated_list[-1 * (allocated_list_count)].base = addr;
            allocated_list[-1 * (allocated_list_count)].size = _size;
            allocated_list_count++;
            return addr;
        }
        /* If requested size is equal to entry size, delete the free list entry */
        /* and add a new entry in allocated list */
        else if (free_list[i].size == _size)
        {
            unsigned long addr = free_list[i].base;
            delete_node(free_list, i, 1, free_list_count);
            free_list_count--;
            allocated_list[-1 * (allocated_list_count)].base = addr;
            allocated_list[-1 * (allocated_list_count)].size = _size;
            allocated_list_count++;
            return addr;
        }
    }
    return 0;
}
```

Figure 6: VM Pool find free space

- void VMPool::release(unsigned long _start_address): Releases a region of previously allocated memory. The region is identified by its start address, which was returned when the region was allocated.
- bool VMPool::is_legitimate(unsigned long _address): By using the base address and size of the given VM pool, the function verifies whether a given address belongs to the VMPool.



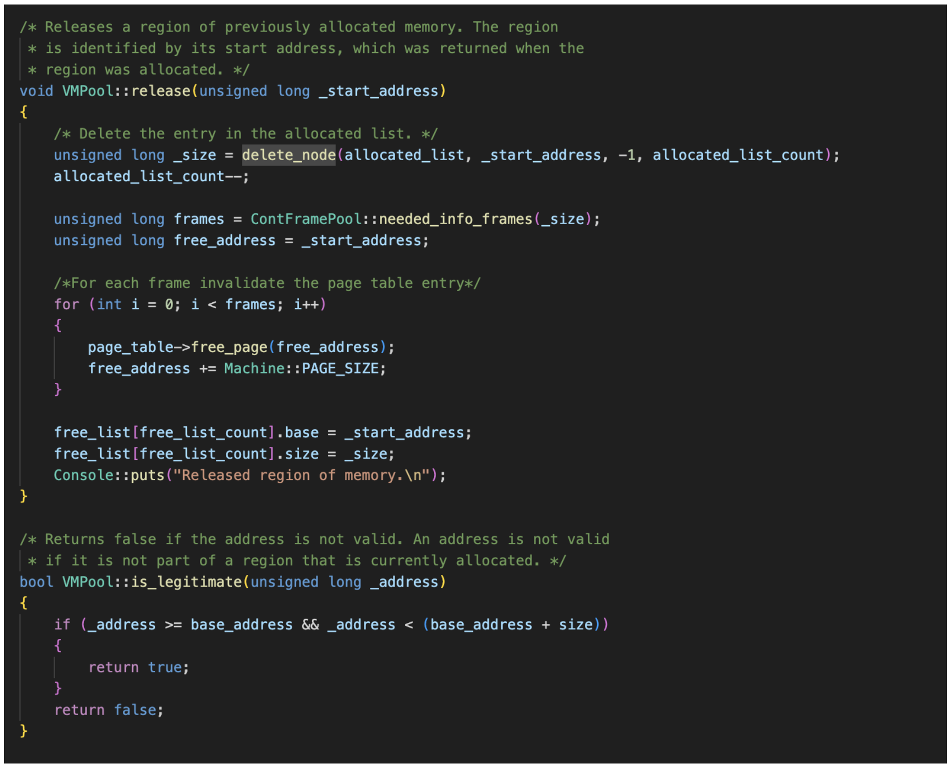
```
/* Allocates a region of virtual space*/
unsigned long VMPool::allocate(unsigned long _size)
{
    /* Inititalize the free list and allocated regions for the first time*/
    if (free_list_count == 0 && allocated_list_count == 0)
    {
        this->free_list[0].base = base_address + 4 KB;
        this->free_list->size = size - 4 KB;
        free_list_count++;
    }

    /* Total size of metadata is 4 KB. Check whether the free list*/
    /* and allocated regions acquire more than 4 KB. */
    if (free_list_count + allocated_list_count >= MAX_REGIONS)
    {
        Console::puts("Unable to allocate due to less space in metadata storage.\n");
        return 0;
    }
    unsigned long va = find_free_space(_size);

    /* If unable to find the virtual address, mark as failure to allocate the requested size */
    if (va == 0)
    {
        Console::puts("No Free space in VMPool. Failed to Allocate memory.\n");
        return 0;
    }

    /* For the allocate virtual address, map the physical address using page table */
    if (page_table->allocate(va, _size))
    {
        Console::puts("Allocated region of memory. Starting from: ");
        Console::puti(va);
        Console::puts("\n");
        return va;
    }

    /* If mapping address in page table fails, recapture the regions and add entries to free list */
    delete_node(allocated_list, va, -1, allocated_list_count);
    allocated_list_count--;
    free_list[free_list_count].base = va;
    free_list[free_list_count].size = _size;
    free_list_count++;
    Console::puts("Failed to Allocate region of memory.\n");
    return 0;
```

Figure 7: VM Pool Allocate

Testing

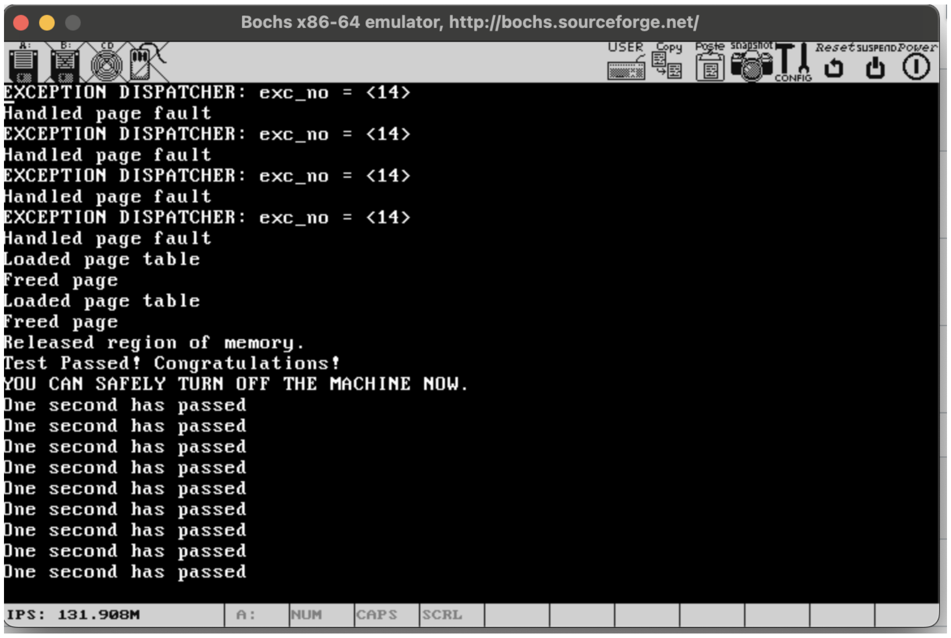
- Testing VM pool functionality: This test whether VM pool is able to assign and release regions of memory.
- Testing new and delete operators: The functions tests whether the operator binding works without any issues.

Both the tests are passing.



```
/* Releases a region of previously allocated memory. The region
 * is identified by its start address, which was returned when the
 * region was allocated. */
void VMPool::release(unsigned long _start_address)
{
    /* Delete the entry in the allocated list. */
    unsigned long _size = delete_node(allocated_list, _start_address, -1, allocated_list_count);
    allocated_list_count--;

    unsigned long frames = ContFramePool::needed_info_frames(_size);
    unsigned long free_address = _start_address;

    /*For each frame invalidate the page table entry*/
    for (int i = 0; i < frames; i++)
    {
        page_table->free_page(free_address);
        free_address += Machine::PAGE_SIZE;
    }

    free_list[free_list_count].base = _start_address;
    free_list[free_list_count].size = _size;
    Console::puts("Released region of memory.\n");
}

/* Returns false if the address is not valid. An address is not valid
 * if it is not part of a region that is currently allocated. */
bool VMPool::is_legitimate(unsigned long _address)
{
    if (_address >= base_address && _address < (base_address + size))
    {
        return true;
    }
    return false;
}
```

Figure 8: VM Pool Release and isLegitimate



```
EXCEPTION DISPATCHER: exc_no = <14>
Handled page fault
EXCEPTION DISPATCHER: exc_no = <14>
Handled page fault
EXCEPTION DISPATCHER: exc_no = <14>
Handled page fault
EXCEPTION DISPATCHER: exc_no = <14>
Handled page fault
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
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
```

Figure 9: Testing Results