Kwangkyu Hwang

MP4 design document

Editted file:

page_table.H
page_table.C
vm_pool.H
vm_pool.C

Flow in kernel.C

After operate timer, exception handler, interrupt handler, keyboard etc, initiallize frame pools that we implemented last machine problem.
Kernel frame pool is 2MB size and starts from address 2MB.
Process frame pool is 28MB size and starts from address 4MB.
And then register page fault handler to deal with page fault.
Now, the program initialize the page table by calling PageTable::init_paging(), PageTable Constructor, PageTable::load(), PageTable::enable_paging()
In **PageTable Constructor,** in mp3, we made page_directory and page_table in directly mapped kernel space using kernel frame pool. In mp4, however, we make page_directory and page_table in virtual memory using process frame pool.

After that check page table using **GeneratePageTableMemoryReferences** method.
It allocate int values to 4-5MB address. Since we didn't allocate at 4MB address, it causes page fault and calls page fault handler. A page fault is handled in
PageTable::handle_fault

Next, we define the code pool(256MB size) from virtual address 512MB and heap pool(256MB size) from virtual address 1GB. **(VMPool Constructor)**
To check whether VMPool created properly or not, call **GenerateVMPoolMemoryReferences** for code_pool and heap_pool.
In **GenerateVMPoolMemoryReferences**, operator new[], and operator delete[] is called and these operators calls **allocate and release of vm_pool.** Also it checks whether address is legitimate or not using **pool->is_legitimate()**

Thus, implemented and modified methods in this MP4 are

PageTable::PageTable() Constructor
PageTable::handle_fault(REGS * _r)
PageTable::register_pool(VMPool * _vm_pool)
PageTable::free_page(unsigned long _page_no)

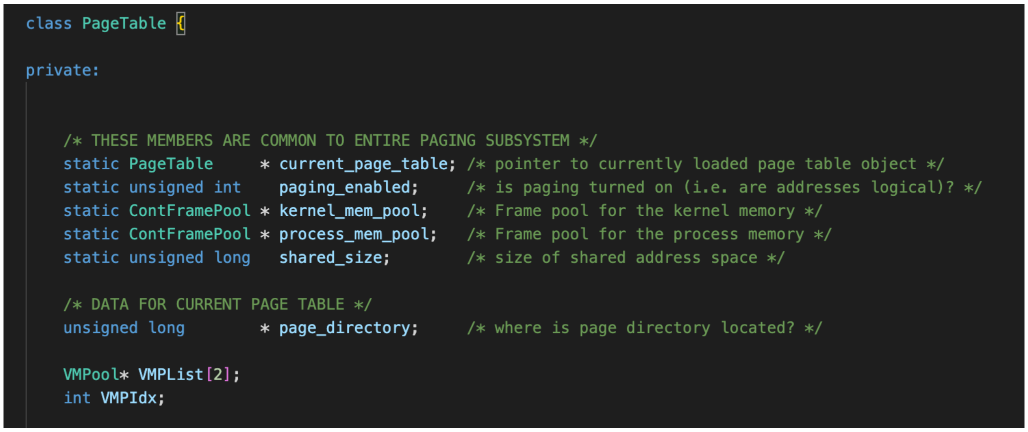
VMPool::VMPool() constructor

Kwangkyu Hwang

VMPool::allocate(unsigned long _size)
VMPool::release(unsigned long _start_address)
VMPool::is_legitimate(unsigned long _address)

Kwangkyu Hwang

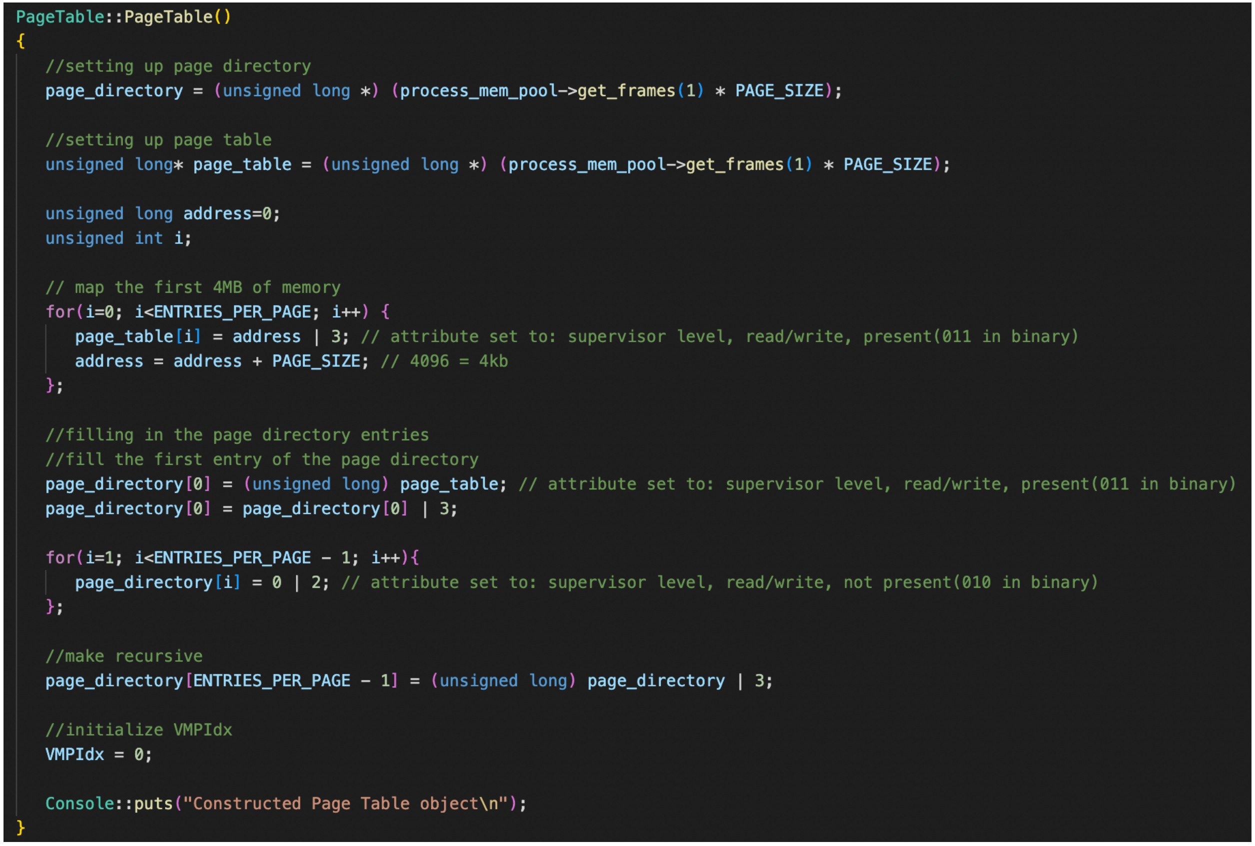
PageTable



```
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? */

   VMPool* VMPList[2];
   int VMPIdx;
```

Since we need to manage VMpools in Page table, I added VMPool* array VMPList and VMPIdx which indicate current VMPList size

PageTable::PageTable() Constructor

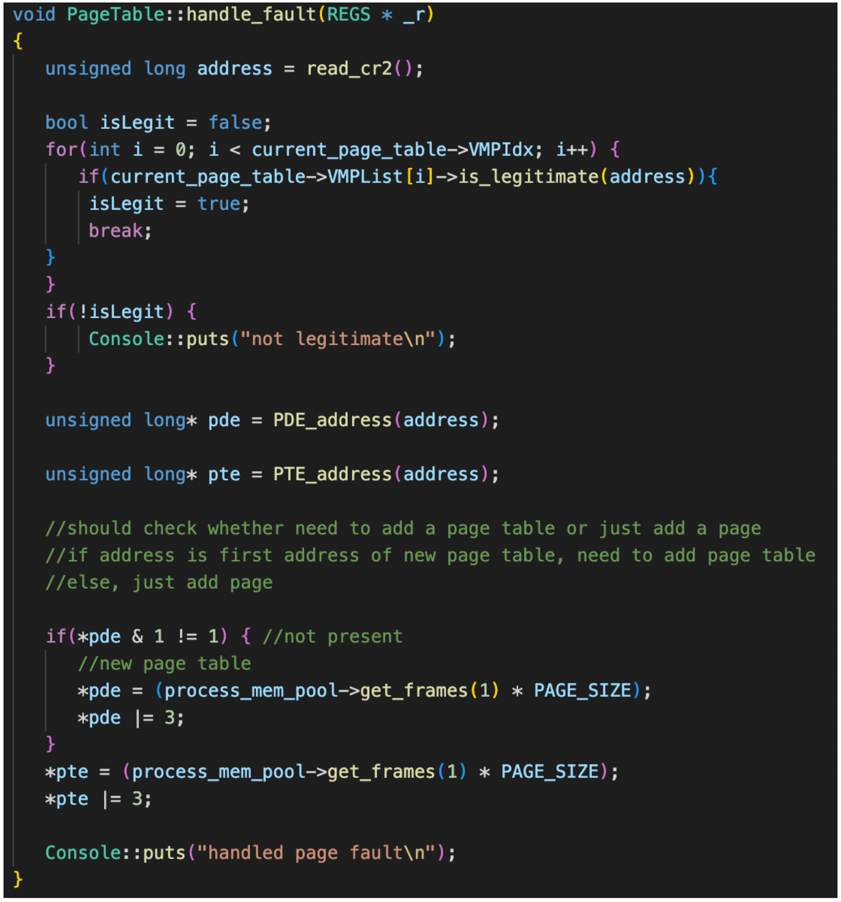


```
PageTable::PageTable()
{
   //setting up page directory
   page_directory = (unsigned long *) (process_mem_pool->get_frames(1) * PAGE_SIZE);

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

   unsigned long address=0;
   unsigned int i;

   // map the first 4MB of memory
   for(i=0; i<ENTRIES_PER_PAGE; i++) {
      page_table[i] = address | 3; // attribute set to: supervisor level, read/write, present(011 in binary)
      address = address + PAGE_SIZE; // 4096 = 4kb
   };

   //filling in the page directory entries
   //fill the first entry of the page directory
   page_directory[0] = (unsigned long) page_table; // attribute set to: supervisor level, read/write, present(011 in binary)
   page_directory[0] = page_directory[0] | 3;

   for(i=1; i<ENTRIES_PER_PAGE - 1; i++){
      page_directory[i] = 0 | 2; // attribute set to: supervisor level, read/write, not present(010 in binary)
   };

   //make recursive
   page_directory[ENTRIES_PER_PAGE - 1] = (unsigned long) page_directory | 3;

   //initialize VMPIdx
   VMPIdx = 0;

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

Constructor has several changes.
First, we get frames from process_mem_pool, not kernel_mem_pool.
And to support recursive page table lookup, set last page_directory entry points to page_directory itself.
And initialize VMPIdx.

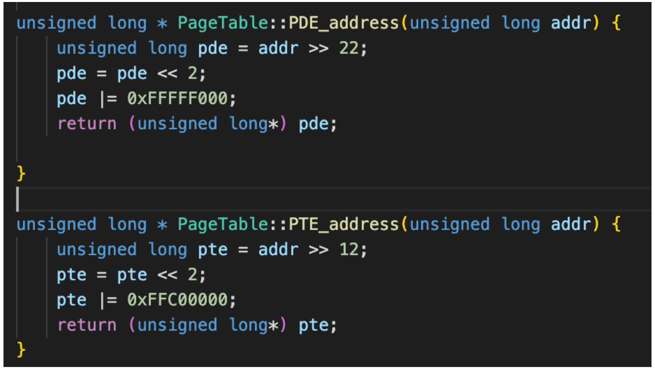
Kwangkyu Hwang

PageTable::handle_fault(REGS * _r)



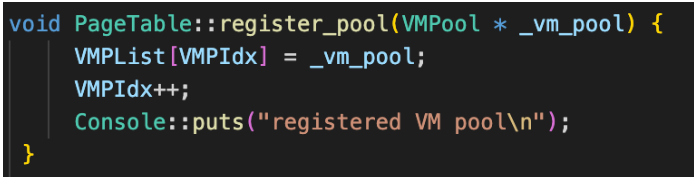
```
void PageTable::handle_fault(REGS * _r)
{
   unsigned long address = read_cr2();

   bool isLegit = false;
   for(int i = 0; i < current_page_table->VMPIdx; i++) {
      if(current_page_table->VMPList[i]->is_legitimate(address)){
       isLegit = true;
       break;
   }
   }
   if(!isLegit) {
       Console::puts("not legitimate\n");
   }

   unsigned long* pde = PDE_address(address);

   unsigned long* pte = PTE_address(address);

   //should check whether need to add a page table or just add a page
   //if address is first address of new page table, need to add page table
   //else, just add page

   if(*pde & 1 != 1) { //not present
      //new page table
      *pde = (process_mem_pool->get_frames(1) * PAGE_SIZE);
      *pde |= 3;
   }
   *pte = (process_mem_pool->get_frames(1) * PAGE_SIZE);
   *pte |= 3;

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

At first we check the address is legitimate using implemented method VMPool::is_legitimate().
After that, convert address to PDE_address, and PTE_address using methods below.
Check attribute from PDE and if it require new page table, make it using process_mem_pool



```
unsigned long * PageTable::PDE_address(unsigned long addr) {
    unsigned long pde = addr >> 22;
    pde = pde << 2;
    pde |= 0xFFFFF000;
    return (unsigned long*) pde;

}

unsigned long * PageTable::PTE_address(unsigned long addr) {
    unsigned long pte = addr >> 12;
    pte = pte << 2;
    pte |= 0xFFC00000;
    return (unsigned long*) pte;
}
```

PageTable::register_pool(VMPool * _vm_pool)

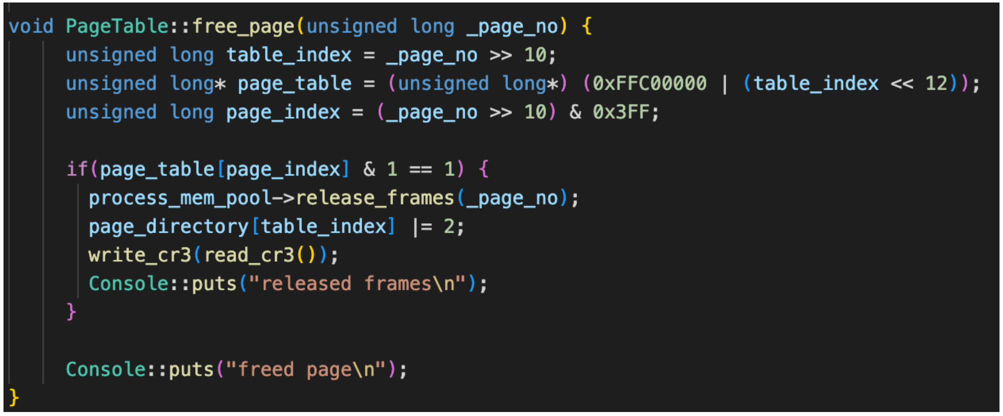


```
void PageTable::register_pool(VMPool * _vm_pool) {
     VMPList[VMPIdx] = _vm_pool;
     VMPIdx++;
     Console::puts("registered VM pool\n");
 }
```

Add _vm_pool to VMPList and increase VMPIdx

Kwangkyu Hwang

PageTable::free_page(unsigned long _page_no)

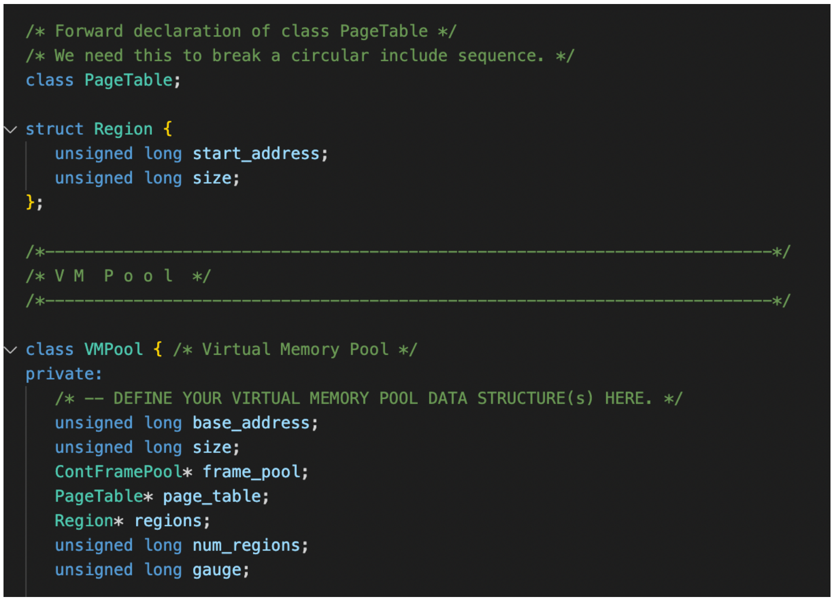


```
void PageTable::free_page(unsigned long _page_no) {
    unsigned long table_index = _page_no >> 10;
    unsigned long* page_table = (unsigned long*) (0xFFC00000 | (table_index << 12));
    unsigned long page_index = (_page_no >> 10) & 0x3FF;

    if(page_table[page_index] & 1 == 1) {
      process_mem_pool->release_frames(_page_no);
      page_directory[table_index] |= 2;
      write_cr3(read_cr3());
      Console::puts("released frames\n");
    }

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

When delete[] is called following methods are called
delete[] → current_pool->release → page_table->free_page(page_no)
To check PTE whether it’s occupied, we can calculate page table in index.
By using release_frames method in process_mem_pool, we can release it and set attribute as not present. Also we should flush TLB here.

Kwangkyu Hwang

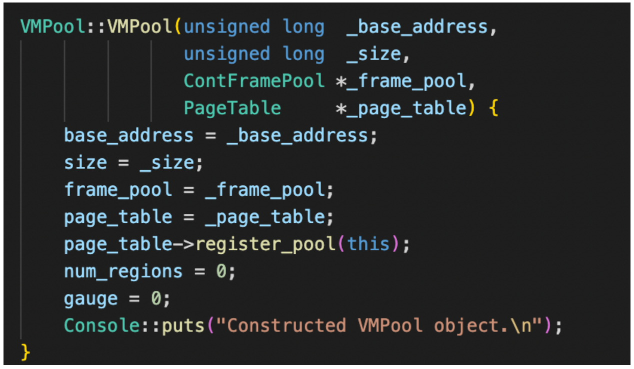
VMPool



```
/* Forward declaration of class PageTable */
/* We need this to break a circular include sequence. */
class PageTable;

struct Region {
    unsigned long start_address;
    unsigned long size;
};

/*--------------------------------------------------------------------------*/
/* V M  P o o l  */
/*--------------------------------------------------------------------------*/

class VMPool { /* Virtual Memory Pool */
private:
    /* -- DEFINE YOUR VIRTUAL MEMORY POOL DATA STRUCTURE(s) HERE. */
    unsigned long base_address;
    unsigned long size;
    ContFramePool* frame_pool;
    PageTable* page_table;
    Region* regions;
    unsigned long num_regions;
    unsigned long gauge;
```

Each VMPool can have multiple regions. Thus define struct Region and add private member variable to point regions and number of regions(num_regions). Also add member variables which will set by Constructor(base_address, size, frame_pool, page_table) and we need a variable that indicate what amount of pool is used(gauge).

VMPool::VMPool() constructor

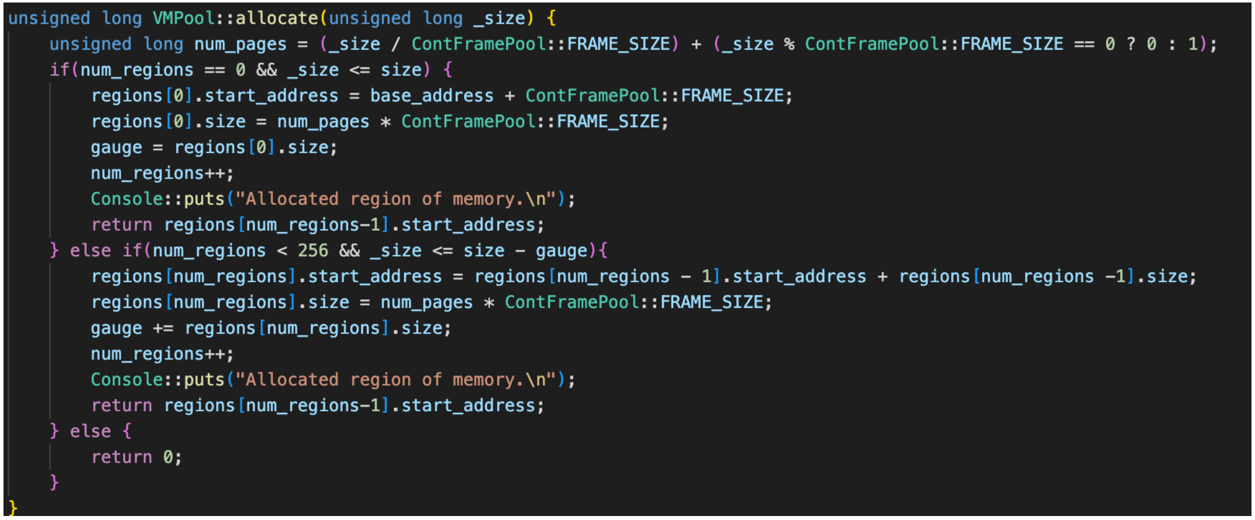


```
VMPool::VMPool(unsigned long  _base_address,
               unsigned long  _size,
               ContFramePool *_frame_pool,
               PageTable     *_page_table) {
    base_address = _base_address;
    size = _size;
    frame_pool = _frame_pool;
    page_table = _page_table;
    page_table->register_pool(this);
    num_regions = 0;
    gauge = 0;
    Console::puts("Constructed VMPool object.\n");
}
```

Set parameters to private member vaiables and initialize regions and gauge. Register this to Page tagle, since it is manager by page table.

Kwangkyu Hwang

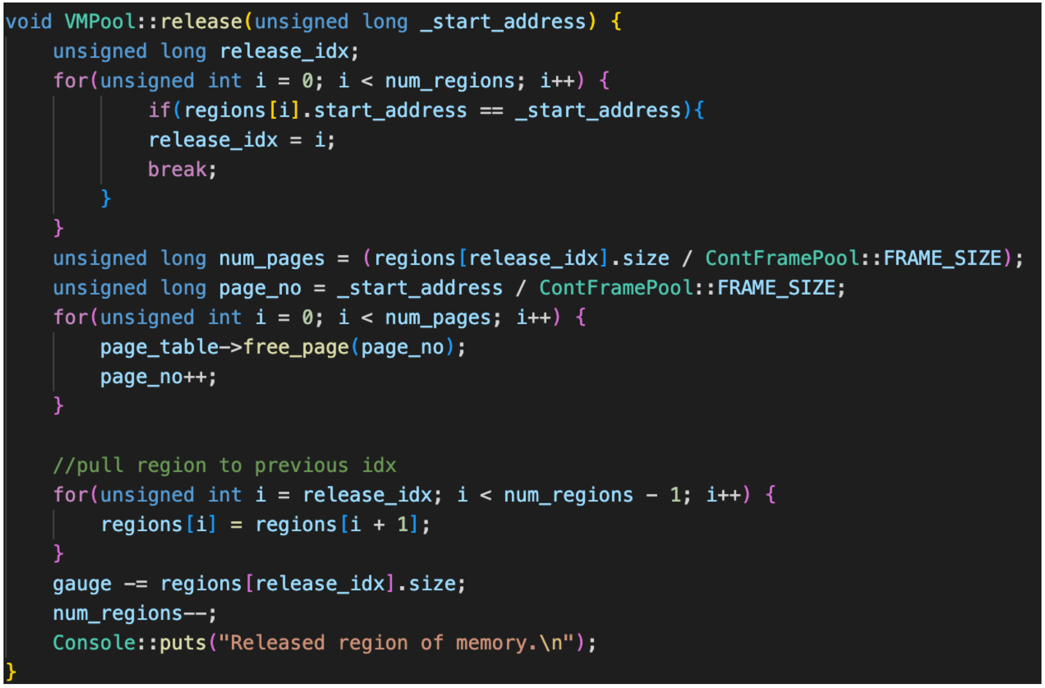
VMPool::allocate(unsigned long _size)



```
unsigned long VMPool::allocate(unsigned long _size) {
    unsigned long num_pages = (_size / ContFramePool::FRAME_SIZE) + (_size % ContFramePool::FRAME_SIZE == 0 ? 0 : 1);
    if(num_regions == 0 && _size <= size) {
        regions[0].start_address = base_address + ContFramePool::FRAME_SIZE;
        regions[0].size = num_pages * ContFramePool::FRAME_SIZE;
        gauge = regions[0].size;
        num_regions++;
        Console::puts("Allocated region of memory.\n");
        return regions[num_regions-1].start_address;
    } else if(num_regions < 256 && _size <= size - gauge){
        regions[num_regions].start_address = regions[num_regions - 1].start_address + regions[num_regions -1].size;
        regions[num_regions].size = num_pages * ContFramePool::FRAME_SIZE;
        gauge += regions[num_regions].size;
        num_regions++;
        Console::puts("Allocated region of memory.\n");
        return regions[num_regions-1].start_address;
    } else {
        return 0;
    }
}
```

We need to allocate space when operator new or new[] called.
At first we need to calculate how many pages needed, and it can easily calculated by FRAME_SIZE
After that assign new regions based on the limit of num_regions and left space(size – gauge). It it is over the limit, retun 0, else return start address of allocated region.

VMPool::release(unsigned long _start_address)

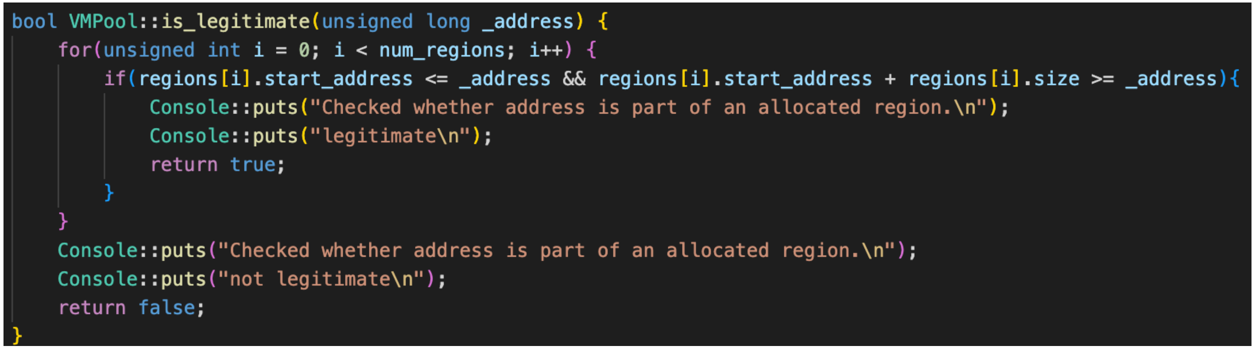


```
void VMPool::release(unsigned long _start_address) {
    unsigned long release_idx;
    for(unsigned int i = 0; i < num_regions; i++) {
            if(regions[i].start_address == _start_address){
            release_idx = i;
            break;
        }
    }
    unsigned long num_pages = (regions[release_idx].size / ContFramePool::FRAME_SIZE);
    unsigned long page_no = _start_address / ContFramePool::FRAME_SIZE;
    for(unsigned int i = 0; i < num_pages; i++) {
        page_table->free_page(page_no);
        page_no++;
    }

    //pull region to previous idx
    for(unsigned int i = release_idx; i < num_regions - 1; i++) {
        regions[i] = regions[i + 1];
    }
    gauge -= regions[release_idx].size;
    num_regions--;
    Console::puts("Released region of memory.\n");
}
```

When release called, we need to find which region in VMPool is releasing now. Thus check all regions until find the same start_address. And than calculate pages that the region have. Free those pages using PageTable::free_page method.
After removing target region, we need to pull next regions of te removed index and decrease the num_regions.

Kwangkyu Hwang

VMPool::is_legitimate(unsigned long _address)



```
bool VMPool::is_legitimate(unsigned long _address) {
    for(unsigned int i = 0; i < num_regions; i++) {
        if(regions[i].start_address <= _address && regions[i].start_address + regions[i].size >= _address){
            Console::puts("Checked whether address is part of an allocated region.\n");
            Console::puts("legitimate\n");
            return true;
        }
    }
    Console::puts("Checked whether address is part of an allocated region.\n");
    Console::puts("not legitimate\n");
    return false;
}
```

Check the address is allocated as a region or not.