MP4: Virtual Memory Management and Memory Allocation

Cheng-Yun Cheng
UIN: 633002216
CSCE611: Operating System

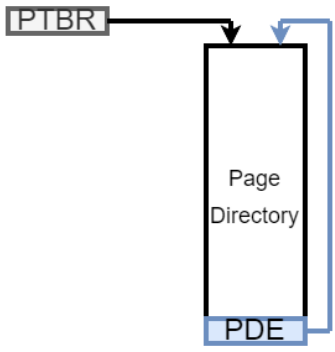
Assigned Tasks

Main (Part1, Part2, Part3): Completed.

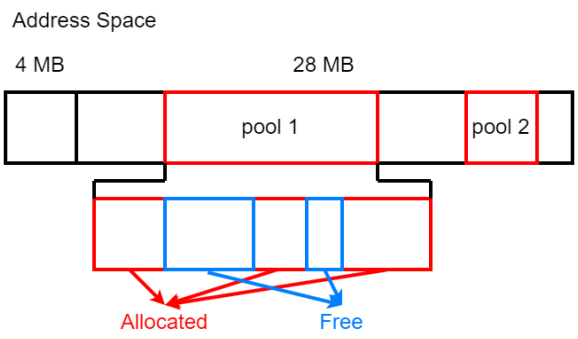
System Design

The goal of the machine problem is to complete a memory manager, including recursive page table look-up and virtual-memory allocator. The memory within the first 4MB is direct-mapped and is reserved for the kernel. The memory beyond 4 MB is freely mapped.

- To support for large address space, the page table pages are allocated in mapped memory. Therefore, recursive page table look-up is needed.



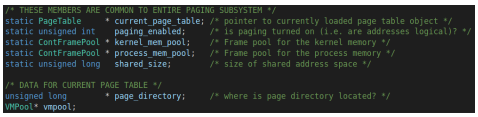
- Virtual-memory pool is implemented and is used by new and delete operators. An address space can have multiple virtual memory pools, and a virtual memory pool can have multiple allocated regions.



Code Description

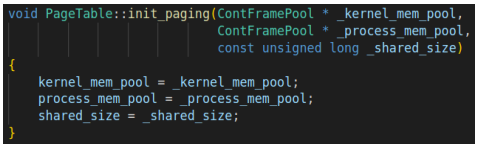
I changed page_table.h, page_table.c, vm_pool.h and vm_pool.c to complete the PageTable class and the VMPool class for this machine problem.

page_table.c: data member : A VMPool pointer was added to record registered pools.



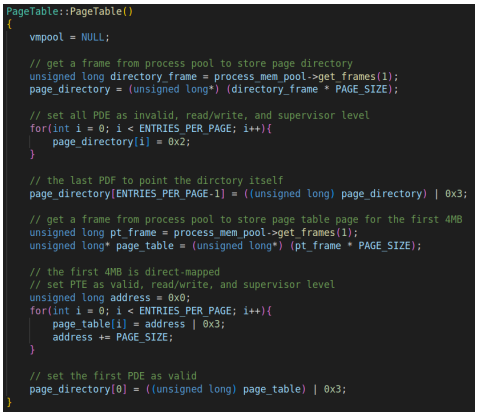
```
/* THESE MEMBERS ARE COMMON TO ENTIRE PAGING SUBSYSTEM */
static PageTable     * current_page_table; /* pointer to currently loaded page table object */
static unsigned int    paging_enabled;     /* is paging turned on (i.e. are addresses logical)? */
static ContFramePool * kernel_mem_pool;    /* 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* vmpool;
```

page_table.c: init_paging : Set the global parameters, kernel_men_pool, process_men_pool, and shared_size, for the paging system.



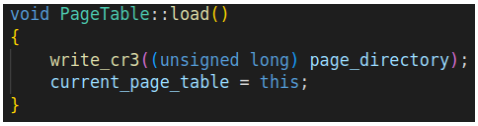
```
void PageTable::init_paging(ContFramePool * _kernel_mem_pool,
                            ContFramePool * _process_mem_pool,
                            const unsigned long _shared_size)
{
    kernel_mem_pool = _kernel_mem_pool;
    process_mem_pool = _process_mem_pool;
    shared_size = _shared_size;
}
```

page_table.c: PageTable constructor : Initialize a page directory and a page table for the first 4 MB. First, get a frame from the process pool to store the page directory. It is important that all the page directory entries should be set as invalid. Furthermore, to implement recursive page table look-up, set the last PDE to point to the page directory itself. Secondly, get another frame from the process pool to store a page table. This page table is used for the first 4 MB memory. Because these memories are direct-mapped, put physical address into page table entries and set them as valid. In the end, update the first PDE and set it as valid.



```
PageTable::PageTable()
{
    vmpool = NULL;

    // get a frame from process pool to store page directory
    unsigned long directory_frame = process_mem_pool->get_frames(1);
    page_directory = (unsigned long*) (directory_frame * PAGE_SIZE);

    // set all PDE as invalid, read/write, and supervisor level
    for(int i = 0; i < ENTRIES_PER_PAGE; i++){
        page_directory[i] = 0x2;
    }

    // the last PDF to point the dirctory itself
    page_directory[ENTRIES_PER_PAGE-1] = ((unsigned long) page_directory) | 0x3;

    // get a frame from process pool to store page table page for the first 4MB
    unsigned long pt_frame = process_mem_pool->get_frames(1);
    unsigned long* page_table = (unsigned long*) (pt_frame * PAGE_SIZE);

    // the first 4MB is direct-mapped
    // set PTE as valid, read/write, and supervisor level
    unsigned long address = 0x0;
    for(int i = 0; i < ENTRIES_PER_PAGE; i++){
        page_table[i] = address | 0x3;
        address += PAGE_SIZE;
    }

    // set the first PDE as valid
    page_directory[0] = ((unsigned long) page_table) | 0x3;
}
```

page_table.c: load : Load the page table into the processor context. There are two steps. The first is storing the address of page directory into the CR3 register, and the second is set the page directory as the current table.



```
void PageTable::load()
{
    write_cr3((unsigned long) page_directory);
    current_page_table = this;
}
```

page_table.c: enable_paging : Enable the paging to switch the kernel from physical addressing to logical addressing. First, set a particular bit in the CR0 register as 1. Secondly, set paging_enabled as 1.

```
void PageTable::enable_paging()
{
    write_cr0(read_cr0() | 0x80000000);
    paging_enabled = 1;
}
```

page_table.c: handle_fault : The page fault handler. Read the address that caused the page fault from the CR2 register, and then call the page_fault function.

```
void PageTable::handle_fault(REGS * _r)
{
    unsigned long logical_address = read_cr2();
    current_page_table->page_fault(logical_address);
}
```

page_table.c: PDE_address : Given a logical address, return the address of its PDE. The PDE address is $\| 1023 : 10 \| 1023 : 10 \|$ page table number $: 10 \| 00 \|$

```
unsigned long* PageTable::PDE_address(unsigned long addr)
{
    return (unsigned long*) (0xFFFFF000 + (addr >> 22 << 2));
}
```

page_table.c: PTE_address : Given a logical address, return the address of its PTE. The PTE address is $\| 1023 : 10 \|$ page table number $: 10 \|$ page number $: 10 \| 00 \|$

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

page_table.c: page_fault : Deal with page fault. First, check whether the logical address is legitimate or not by calling the is_legitimate function for each registered pool. If the address is legitimate, process the page fault just like MP3. Check whether the PDE is valid or not. If the PDE is invlid, get a frame to create a new page table. Update the PDE and set all PTE as invalid. And then, for invalid PTE, get a frame, store its address into PTE, and set PTE as valid. The PDE and PTE can be gotten by the PDE_address function and the PTE_address function.

```
void PageTable::page_fault(unsigned long logical_address)
{
    bool legitimate = false;
    VMPool *vm = vmpool;
    // check the logical_address is legitimate or not
    while(vm){
        legitimate = vm->is_legitimate(logical_address);
        if(legitimate) break;
        vm = vm->next;
    }

    if(legitimate){
        unsigned long* pde = PDE_address(logical_address);
        // if PDE is invalid, create a new page table
        if((*pde & 0x1) == 0){
            // get a frame from process pool and store it in page directory
            unsigned long pt_frame = vmpool->get_frame_pool()->get_frames(1);
            *pde = (pt_frame * PAGE_SIZE) | 0x3;

            // initial all PTE as invalid
            unsigned long* new_pte = PTE_address(logical_address & 0xFFC00000);
            for(int i = 0; i < ENTRIES_PER_PAGE; i++){
                new_pte[i] = 0x0 | 0x2;
            }
        }

        // get a frame from process pool
        // store it in page table and set it as valid
        unsigned long* pte = PTE_address(logical_address);
        if((*pte & 0x1) == 0){
            unsigned long p_frame = vmpool->get_frame_pool()->get_frames(1);
            *pte = (p_frame * PAGE_SIZE) | 0x3;
        }
    }
```

page_table.c: register_pool : Register a virtual memory pool with the page table. PageTable has a VMPool pointer (vmpool) and the VMPool has a VMPool pointer (next) too. Link the page table and the vm pool like a linked list.

```
void PageTable::register_pool(VMPool * _vm_pool)
{
    _vm_pool->next = vmpool;
    vmpool = _vm_pool;
}
```

page_table.c: free_page : If the page is valid (check the valid bit in PTE), call ContFramePool::release_frame function to release the frame and set the PTE as invalid. Furthermore, reload the page table base register (CR3) to flush the entire TLB.

```
void PageTable::free_page(unsigned long _page_no) {
    unsigned long* pte = PTE_address(_page_no);
    if((*pte & 0x1) == 1){
        ContFramePool::release_frames(*pte >> 12);
        *pte = 0x0 | 0x2;
        write_cr3((unsigned long) page_directory);
    }
}
```

vm_pool.h: data member :

- unsigned long base_address: The logical start address of the pool
- unsigned long size: The size of the pool in bytes
- ContFramePool* frame_pool: A pointer points to the frame pool that provides the virtual memory pool with physical memory frames.
- PageTable* page_table: A pointer points to the page table that maps the logical memory references to physical addresses.
- Region* allocated_list: An array of allocated regions
- unsigned long allocated_size: The size of the allocated list
- Region* free_list: An array of free regions
- unsigned long free_size: The size of the free list
- VMPool* next: A pointer points to the next VMPool
- const unsigned int PAGE_SIZE: 4 KB
- Region: A class contains base_address and the size (number of pages)

```
private:
    /* -- DEFINE YOUR VIRTUAL MEMORY POOL DATA STRUCTURE(s) HERE. */
    unsigned long base_address;
    unsigned long size;
    ContFramePool* frame_pool;
    PageTable* page_table;
    Region* allocated_list;
    unsigned long allocated_size;
    Region* free_list;
    unsigned long free_size;

public:
    VMPool* next;
    static const unsigned int PAGE_SIZE = Machine::PAGE_SIZE;
```

```
struct Region
{
    unsigned long base_address;
    unsigned long size;
};
```

vm_pool.c: constructor : There are three parts in this function. First, set the data member, base_address, size, frame_pool, page_table, and next. Second, call the page_table's register_pool function to register the vm pool. Third, initialize the allocated list and the free list. The first page in the pool is used to store these two array, the first half (base_address to base_address + PAGE_SIZE/2) for the allocated list and the other (base_address + PAGE_SIZE/2 to base_address + PAGE_SIZE) for the free list. Also, the first element in the allocated list is the first page, and the first element in the free list is the rest of memory in 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;
    next = NULL;

    page_table->register_pool(this);

    // using the first page to store allocated list and free list
    // the first half for allocated list and the other for free list
    // allocated_list[0] stores these two lists
    allocated_list = (Region*) _base_address;
    allocated_size = 1;
    allocated_list[0].base_address = base_address;
    allocated_list[0].size = 1;

    // free_list[0] stores rest of the pool
    free_list = (Region*) (_base_address + PAGE_SIZE/2);
    free_size = 1;
    free_list[0].base_address = base_address + PAGE_SIZE;
    free_list[0].size = size/PAGE_SIZE - 1;
}
```

vm_pool.c: allocate : Allocates a region of _size bytes of memory from the virtual memory pool. Check whether the size of the allocated list is less than 256 or not first, because only half page is used for the allocated list. Calculate how many pages are needed to allocate and search free regions in the free list. If the size of the free region is equal to the demand, remove it from the free list; if the size of free region is greater than the demand, split the region and update the free list. Finally, add the region into the allocated list. On the other hand, if there is no proper region, return 0.

```
unsigned long VMPool::allocate(unsigned long _size) {
    // the size of allocated list must be less than 256
    assert(allocated_size < 256);

    // calculate how many pages are needed
    unsigned long alloc_size = _size / PAGE_SIZE + (_size % PAGE_SIZE > 0 ? 1 : 0);

    // search proper memory region in free list
    for(int i = 0; i < free_size; i++){
        if(free_list[i].size >= alloc_size){
            unsigned long ret_address = free_list[i].base_address;
            free_list[i].size -= alloc_size;

            // if the free region is greater than allocated size, update it
            if(free_list[i].size){
                free_list[i].base_address += alloc_size*PAGE_SIZE;
            }
            // if the free region is equal to allocated size, delete it
            else{
                free_list[i] = free_list[--free_size];
            }

            // update allocated list
            allocated_list[allocated_size].base_address = ret_address;
            allocated_list[allocated_size].size = alloc_size;
            allocated_size++;

            return ret_address;
        }
    }

    return 0;
}
```

vm_pool.c: release : Releases a region of previously allocated memory. Check whether the size of the free list is less than 256 or not first, because only half page is used for the free list. Search the region whose base_address is equal to the input in the allocated list, and then call free_page function to free all the pages in this region. Finally, remove the region from the allocated list and add it into the free list.

```
void VMPool::release(unsigned long _start_address) {
    // the size of free_list must be less than 256
    assert(free_size < 256);

    for(int i = 0; i < allocated_size; i++){
        // find the region needed to release
        if(allocated_list[i].base_address = _start_address){
            // free all the pages in the region
            for(int j = 0; j < allocated_list[i].size; j++){
                page_table->free_page(_start_address + j * PAGE_SIZE);
            }

            // update allocated list and free list
            free_list[free_size] = allocated_list[i];
            free_size++;
            allocated_list[i] = allocated_list[--allocated_size];
            break;
        }
    }
}
```

vm_pool.c: is_legitimate : Return true if the given address is in an allocated region. There are two parts in the function. The first page in the pool is always legitimate because the allocated list and the free list are stored in this page. For other page, check whether it is in an allocated region (alloc_region.base_address <= _address < alloc_region.base_address + alloc_region.size). If so, return true.

```
bool VMPool::is_legitimate(unsigned long _address) {
    // The first page is used for allocated list and free list
    if( base_address <= _address && _address < base_address + PAGE_SIZE)
        return true;

    for(int i = 0; i < allocated_size; i++){
        Region r = allocated_list[i];
        // if _address is in allocated region, return true
        if(r.base_address <= _address && _address < (r.base_address + r.size*PAGE_SIZE))
            return true;
    }
    return false;
}
```

vm_pool.c: get_frame_pool : Return the frame pool that provides the virtual memory pool with physical memory frames.

```
ContFramePool* VMPool::get_frame_pool(){
    return frame_pool;
}
```

Testing

For the machine problem, I only use the provided test, because it covers all functions which I implemented. The result shows below.