MP4: Virtual Memory Management and Memory Allocation

Nachiket Umesh Naganure
UIN: 532008698
CSCE611: Operating System

Assigned Tasks

Part I: Support for Large Address Spaces Completed.
Part II: Preparing class PageTable to handle Virtual Memory Pools Completed.
Part III: An Allocator for Virtual Memory Completed.

System Design

The goal of the machine problem was to design a virtual memory manager with support to for allocating memory using new and delete keywords. We made changes to our previous mp3 page table implementation to move the page tables from kernel space to process space to support large address spaces.

1. For this task, we've changed the page fault logic to support virtual memory pools
2. Added VMPool Class to manage memory pools and to support allocate and deallocation of memory segments
3. We use the first page of each new pool to store the struct array which stores information about the allocated regions in the pool

Code Description

I mainly made changes in "page_table.c", "vm_pool.c" and "vm_pool.h" while importing code from previous machine problem (mainly "cont_frame_pool.c" and "cont_frame_pool.h"). To compile the code, you need to run the make file. To run the simulator, use copykernel.sh script and run the command "bochs -f bochsrc.bxrc" to start the bochs simulator.

Part I: Support for Large Address Spaces Recursive Page Lookup. To implement Recursive page lookup, we make the last entry of page directory to point to the page directory itself. In addition, we index into the PDE and PTE values of given address using 1023—1023—X—offset and 1023—X—Y—offset as mentioned in the handout.

Part II: Preparing class PageTable to handle Virtual Memory Pools The PageTable class was modeified for management of virtual memory. Register Pool API aids in maintaining record of all memory pools in an array. Page fault handler initially checks if the address is valid ie does it belong to one the pools. If it does then frame allocation is done.

Part III: An Allocator for Virtual Memory VMPool class has function for allocation which checks if there is a free region to allocate the memory segment, it also had deallocate which frees up memory by calling page table class to free pages.

PageTable Constructor :

```
PageTable::PageTable()
{
    page_directory = (unsigned long *) (kernel_mem_pool->get_frames( n_frames: 1) * PAGE_SIZE);
    auto* page_table = (unsigned long *) (kernel_mem_pool->get_frames( n_frames: 1) * PAGE_SIZE);

    unsigned long address = 0;
    for(int i=0; i<1024; i++){
        page_table[i] = address | 3;
        address += PAGE_SIZE;
    }

    page_directory[0] = (unsigned long) page_table | 3;
    for(int i=1; i<1024; i++)
    {
        if(i == 1023)
            page_directory[i] = (unsigned long) page_directory | 3;
        else
            page_directory[i] = 2; // attribute set to: supervisor level, read/write, not present(010 in binary)
    };

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

Figure 1: Constructor of Page Table

Page Fault handler :

```
void PageTable::handle_fault(REGS * _r)
{
    Console::puts( s: "ENTERED handle_fault\n");
    unsigned int error = _r->err_code;
    if(error & 1){
        Console::puts( s: "Protection Fault, NOT a page fault");
        return;
    }
    unsigned long page_fault_address = read_cr2();

    bool valid_pool_page = false;
    int num_of_valid_pools = 0;

    for(int i=0; i<512; i++){
        if(vm[i] != nullptr){
            if(vm[i]->is_legitimate( address: page_fault_address)){
                valid_pool_page=true;
                break;
            }
            else{
                num_of_valid_pools++;
            }
        }
    }
    if(!valid_pool_page && num_of_valid_pools){
        Console::puts( s: "No valid pool associated to address\n");
        assert(false);
    }

    unsigned long *pde_pointer = PDE_address( addr: page_fault_address);
    unsigned long *pte_pointer = PTE_address( addr: page_fault_address);

    if(*pde_pointer & 1){
        *pte_pointer = (unsigned long) (process_mem_pool->get_frames( n_frames: 1) * PAGE_SIZE) | 3;
    }
    else{
        *pde_pointer = (unsigned long) (process_mem_pool->get_frames( n_frames: 1) * PAGE_SIZE) | 3;

        *pte_pointer = (unsigned long) (process_mem_pool->get_frames( n_frames: 1) * PAGE_SIZE) | 3;

    }
```

Figure 2: Page Fault Handler of Page Table

```
void PageTable::register_pool(VMPool * _vm_pool)
{
    if(pool_number == 512){
        Console::puts( s: "No free pools available");
        assert(false);
    }
    vm[pool_number] = _vm_pool;
    pool_number++;
    Console::puts( s: "registered VM pool\n");
}

void PageTable::free_page(unsigned long _page_no) {
    unsigned long *pte = PTE_address( addr: _page_no);
    if(*pte & 1){
        process_mem_pool->release_frames( first_frame_no: *pte>>12);
    }
    *pte = 2;
    write_cr3( val: (unsigned long) page_directory);
    Console::puts( s: "freed page\n");
}
```

Figure 3: Register VM Pool function and Free page function of Page Table

PDE and PTE Addresses :

```
unsigned long * PageTable::PDE_address(unsigned long addr){
    unsigned long pde_bits = (addr >> 20);
    pde_bits |= 0xFFFFF000;
    pde_bits &= 0xFFFFFFFC;
    return (unsigned long*) pde_bits;
}

unsigned long * PageTable::PTE_address(unsigned long addr){
    unsigned long pte_bits = (addr >> 10);
    pte_bits |= 0xFFC00000;
    pte_bits &= 0xFFFFFFFC;
    return (unsigned long*) (pte_bits);

}
```

Figure 4: Fucntion to calculate the pde and pte values from given address

```
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( vm_pool: this);
//    Console::puts("Here after registering");
    memRegion = (mem_region*)(base_address);
    num_of_allocated_regions=1;

    memRegion[0].start_addr = base_address;
    memRegion[0].size = Machine::PAGE_SIZE;
//    Console::puts("something wrong?");

    for(int i=1; i<512; i++){
        memRegion[i].start_addr = 0;
        memRegion[i].size = 0;
    }

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

Figure 5: Fucntion to calculate the pde and pte values from given address

```
unsigned long VMPool::allocate(unsigned long _size) {
    Console::puts( s: "In allocate\n");
    if(num_of_allocated_regions>512){
        Console::puts( s: "Ran out of regions to allocate\n");
        assert(false);
    }
    unsigned long num_of_pages = _size/Machine::PAGE_SIZE;
    unsigned long offset = _size%Machine::PAGE_SIZE;
    if(offset) num_of_pages++;

    unsigned long new_start_address = memRegion[num_of_allocated_regions-1].start_addr + memRegion[num_of_allocated_regions-1].size;

    memRegion[num_of_allocated_regions].start_addr = new_start_address;
    memRegion[num_of_allocated_regions].size = num_of_pages * Machine::PAGE_SIZE;
    num_of_allocated_regions++;

    Console::puts( s: "Allocated region of memory.\n");
    return new_start_address;
}

void VMPool::release(unsigned long _start_address) {
    int mem_region_index=-1;
    for(int i=0;i<512;i++){...}
    if(mem_region_index == -1){
        Console::puts( s: "No region found with given start address\n");
        assert(false);
    }
    unsigned long num_pages_in_region = memRegion[mem_region_index].size/Machine::PAGE_SIZE;
    unsigned long page_address = _start_address;
    for(int i = mem_region_index;i<num_of_allocated_regions-1;i++){...}
    memRegion[num_of_allocated_regions].start_addr = 0;
    memRegion[num_of_allocated_regions].size = 0;
    while(num_pages_in_region){
        page_table->free_page( page_no: page_address);
        page_address += Machine::PAGE_SIZE;
        num_pages_in_region--;
    }
    num_of_allocated_regions--;

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

Figure 6: Fucntion to allocate and release memory regions from vm pool

```
bool VMPool::is_legitimate(unsigned long _address) {
    if(_address == base_address) return true;
    for(int i=0;i<num_of_allocated_regions;i++){
        if(_address >= memRegion[i].start_addr && _address <= (memRegion[i].start_addr + memRegion[i].size)){
            return true;
        }
    }
    Console::puts( s: "Checked whether address is part of an allocated region.\n");

    return false;
}
```

Figure 7: Fucntion to check if given address is legitimate

Testing

For testing, I used the given test function from kernel.C. I ran two scenarios where the _TEST_PAGE_TABLE_ macro was set and unset. All the tests are passing, for given and additional scenarios.

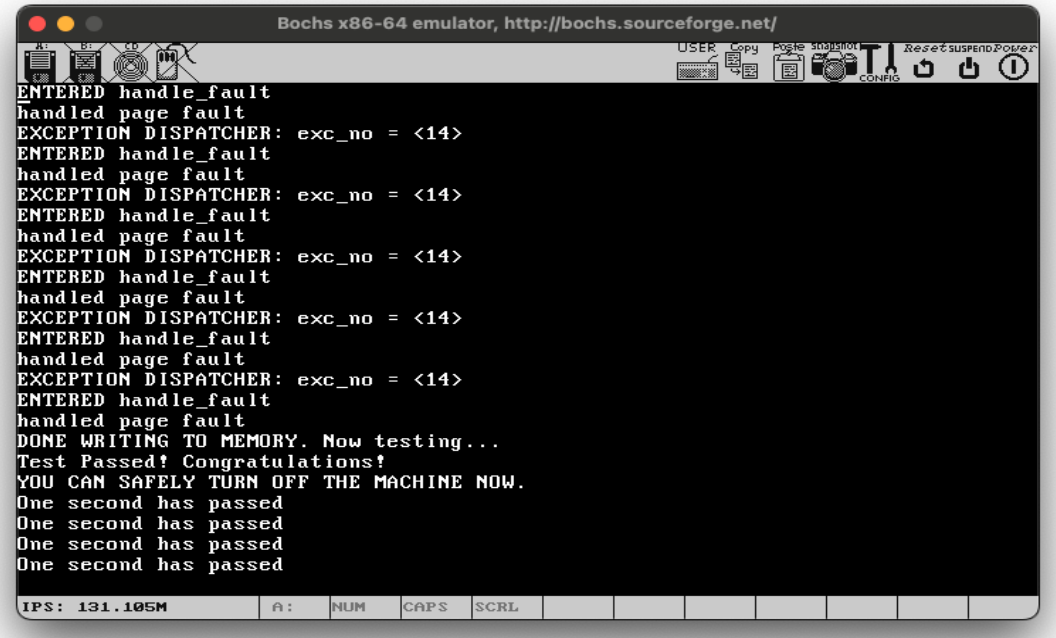

Figure 8: Testing the recursive page table implmentation

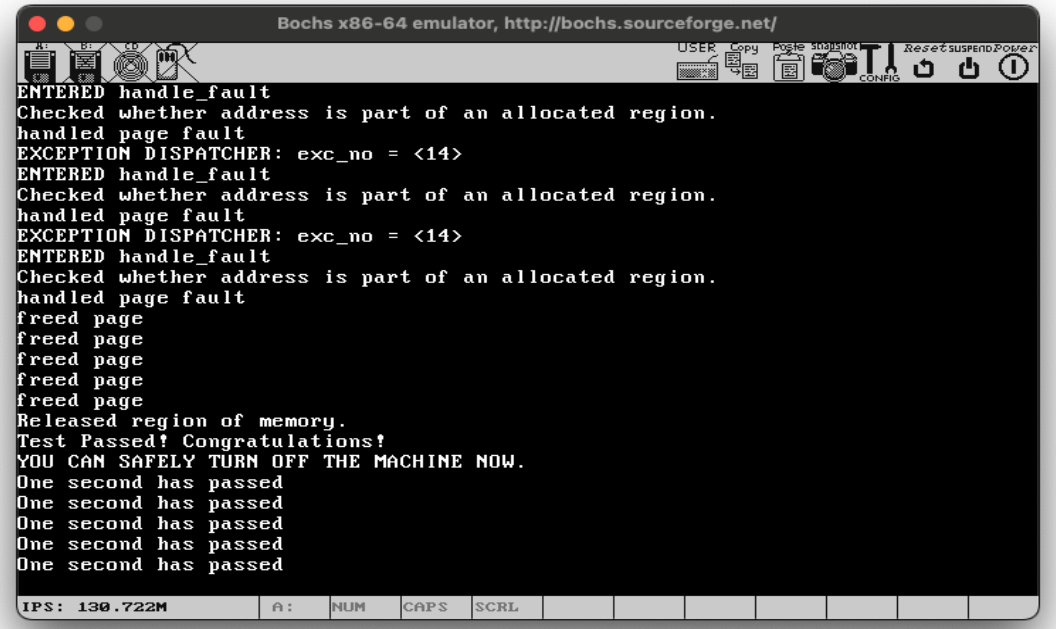

Figure 9: Testing the vm pool implmentation