CSCE 611: Operating Systems
MP4: Design Document

Shweta Sharma
UIN: 433003780

Design Idea:

Machine Problem 4: Virtual Memory Management and Memory Allocation

Multiple virtual memory pools can be created using the base address and size of each pool, along with the frame_pool and page_tables they belong to.

- The virtual memory pool is divided into multiple regions. Each region is a multiple of the Page Size and stores the base address and the size of that region. Regions can be allocated and released using the VMPool::allocate() and VMPool::release() functions.

- The implementation is such that any VMPool must fall within a Page. So, the number of regions allowed in a VM Pool = PAGE_SIZE/(size of VMPoolObject)

- The VMPool follows lazy loading, i.e., the pages are not assigned at Virtual Memory pool creation, but rather on PageFault.

- On calling VMPool::release(), pages belonging to the given region are freed using PageTable::freePage(). Finally, the page_directory and page_table are reloaded to make the TLB consistent.

- The VMPool also provides an API to check if an address is valid or not. This is done using the base address and the size of the VMPool. If the given address falls in the range [base address , base address+ size], the address is legitimate, otherwise not.

The following functions have been implemented in vm_pool.c to facilitate creation of virtual memory:

1. **VMPool** : Initializes the data structures needed for the management of this virtual - memory pool .

```
47
48  VMPool::VMPool(unsigned long  _base_address,
49                 unsigned long  _size,
50                 ContFramePool *_frame_pool,
51                 PageTable     *_page_table) {
52
53      // Initialising variables
54      this->_base_address = _base_address;
55      this->_size = _size;
56      this->_frame_pool = _frame_pool;
57      this->_page_table = _page_table;
58      this->region_iterator = 0;
59
60      allocated_region = (struct allocated_vm_region*) (_base_address);
61      this->_page_table->register_pool(this);
62
63      Console::puts("Constructed VMPool object.\n");
64  }
65
```

2. **allocate**: 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 it fails , return 0.

```
unsigned long VMPool::allocate(unsigned long _size) {
    // _size cannot be zero.
    if(_size == 0) {
        Console::puts("Invalid size for allocate\n");
        assert(false);
        return 0;
    }
    // Virtual memory is full
    if(region_iterator == MAX_VM_REGIONS) {
        Console::puts("VM full\n");
        assert(false);
        return 0;
    }

    //Total number of pages( n_pages_needed ) calculated
    unsigned int n_pages_needed = _size/Machine::PAGE_SIZE;
    if(_size % Machine::PAGE_SIZE > 0){
        n_pages_needed++;
    }

    // Calculating final_mem_size of the new segment along with _base_address for the segment
    unsigned long final_mem_size = n_pages_needed*Machine::PAGE_SIZE;
    if(region_iterator == 0){
        allocated_region[region_iterator]._base_address = _base_address;
    }
    else{
        allocated_region[region_iterator]._base_address = allocated_region[region_iterator-1]._base_address + allocated_region[region_iterator-1]._size;
    }

    // Setting _size variable to final_mem_size
    allocated_region[region_iterator]._size = final_mem_size;
    region_iterator++;

    Console::puts("Allocated region of memory.\n");
    return allocated_region[region_iterator-1]._base_address;
}
```

3. **release**: 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) {
    // Finding indx for the memory pool
    unsigned int indx=0;
    for(;indx<region_iterator;indx++){
        if(allocated_region[indx]._base_address == _start_address){
            break;
        }
    }

    // Calculating number of pages alloted
    unsigned int n_pages_allocated = allocated_region[indx]._size/Machine::PAGE_SIZE;
    // Freeing the alloted pages
    for(unsigned int i=0;i<n_pages_allocated;i++){
        unsigned long address = allocated_region[indx]._base_address+i*Machine::PAGE_SIZE;
        _page_table->free_page(address);
    }

    // Updating the allocated_region array
    for(;indx<region_iterator;indx++){
        allocated_region[indx] = allocated_region[indx+1];
    }
    region_iterator--;

    // Flusing the TLB: We know that loading the page table also flushes the TLB
    _page_table->load();

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

4. **is_legitimate**: 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) {

    unsigned long last_address = _base_address + _size;
    if(_address>=_base_address && _address<=last_address)
        return true;
    return false;
    Console::puts("Checked whether address is part of an allocated region.\n");
}
```

Page Table Logic Changes – MP4:

Apart from the below page management details (implemented in MP3),

- The page now also stores the number of VMPools (=10) and an array of all the VMPools it supports.

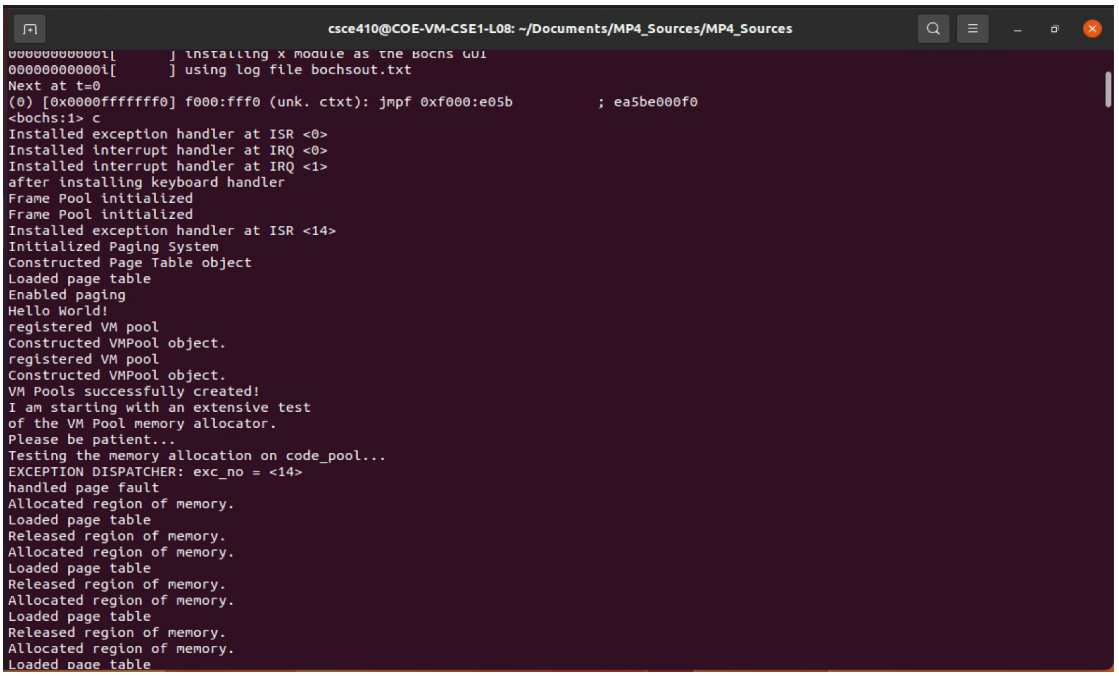
- Also, to enhance the space of the Virtual Memory, the page table and the page directory are now created in the process_mem_pool and not the kernel_mem_pool.

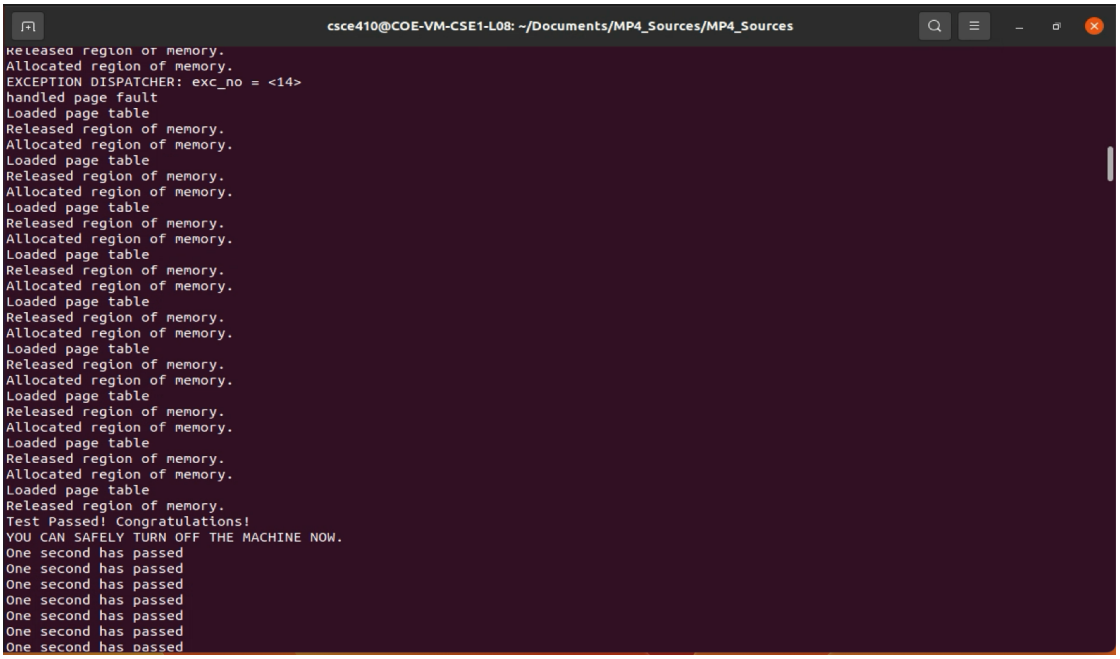
- As the PageTable now does not directly map a logical address to physical address, Recursive Address Resolution is implemented to get the physical addresses from the logical addresses for legitimate pages, checked using VMPool::is_legitimate().

- Two new functions , PageTable::register_pool() and PageTable:: free_page() functions are implemented to register a VMPool to a Page Table and free unused pages respectively.

- An important part of the code is the TLB flush, done by reloading the page directory in the CR3 register.

Test Screenshots / Output:





Page Table Management (implemented in page_table.c) - MP3:
The entire memory (32 MB) is divided among two frame pools:

- **kernel_mem_pool:** Memory in the first 4MB is directly mapped to physical memory, i.e., physical and logical addresses are identical. The first 2 MB is reserved for system use. The kernel_mem_pool, between 2MB and 4MB, manages frames in the shared portion of memory, typically for use by the kernel for its data structures.

- **process_mem_pool:** Above 4MB, the process_mem_pool manages frames in the non-shared memory portion (28MB), i.e., the freely mapped memory, where logical addresses are not the same as physical addresses. Frames in the logical address space are mapped to their physical address space using 2-level paging. Every page in this address range is mapped to whatever physical frame was available when the page was allocated.

After the pools are initialized, the following functions help the kernel to set-up the overall paging experience.

- **init paging:** set the parameters for the paging subsystem.

```
22  void PageTable::init_paging(ContFramePool * _kernel_mem_pool,
23                              ContFramePool * _process_mem_pool,
24                              const unsigned long _shared_size)
25  {
26      PageTable::kernel_mem_pool = _kernel_mem_pool;
27      PageTable::process_mem_pool = _process_mem_pool;
28      PageTable::shared_size = _shared_size;
29      Console::puts("Initialized Paging System\n");
30  }
```

- **PageTable constructor:** sets up the entries in the page directory and the page table for the shared portion of memory. The page directory points to each page table in the shared memory and is set as valid using the last bit of the entry. The page table entries for the shared portion of the memory (i.e., the first 4MB) are marked valid using the last bit in each entry.

```
32  PageTable::PageTable()
33  {
34      page_directory = (unsigned long *)(process_mem_pool-> get_frames(1)* PAGE_SIZE);
35      unsigned long * page_table = (unsigned long *)(process_mem_pool->get_frames(1) * PAGE_SIZE);
36
37      // Calculating size of the page table
38      unsigned long n_shared_frames = (PageTable::shared_size)/PAGE_SIZE;
39
40      unsigned long i, frame_address =0, n_entries=PAGE_SIZE/4; // since each entry is 4 bytes long.
41      //shared memory space -> 4MB/4KB = 1024
42
43      // Storing Main Memory addresses
44      for(i=0;i<n_shared_frames;i++){
45          page_table[i] = frame_address | WRITE_BIT | VALID_BIT;
46          frame_address += PAGE_SIZE;
47      }
48
49      //remaining memory space = 1024-1024 = 0
50      // Valid Bit is not set
51      for(i;i<n_entries;i++){
52          page_table[i] = frame_address | WRITE_BIT;
53          frame_address += PAGE_SIZE;
54      }
55
56
57      //page directory setup
58      page_directory[0] = (unsigned long) page_table | WRITE_BIT | VALID_BIT;
59
60      // For now, our page directory has just one page table reference
61      for(i=1;i<n_entries-1;i++){
62          page_directory[i] = 0 | WRITE_BIT;
63      }
64
65      //Implementing recursive page table lookup: Last entry to point to the start of page_directory
66      page_directory[n_entries-1] = (unsigned long) page_directory | WRITE_BIT | VALID_BIT;
67
68      //setting the vmpools to NULL
69      n_registered_vmpools = 0;
70      for(unsigned int i=0;i<VM_POOLS_MAX;i++) {
71          registered_vmpools[i]=NULL;
72      }
73
74
75      Console::puts("Constructed Page Table object\n");
76  }
```

- **load() function**: The page table is loaded into memory by storing the address of the page directory into the CR3 register (PTBR).

```
90
91  void PageTable::load()
92  {
93      current_page_table = this;
94      write_cr3((unsigned long)(current_page_table-> page_directory));
95      Console::puts("Loaded page table\n");
96  }
```

- **enable_paging() function:** It helps the kernel switch from physical addressing to logical addressing. The paging is enabled by setting the Most Significant Bit (MSB) in the CR0 register.

```
97
98  void PageTable::enable_paging()
99  {
100     paging_enabled = 1;
101     //setting the MSB of cr3 to enable paging.
102     write_cr0(read_cr0() | MSB_MASK);
103     Console::puts("Enabled paging\n");
104 }
105
```

- **handle_fault():** This method first reads the logical_address from the CR2 register and the error_word from the error_code present in the REGS object. It then looks up the appropriate entry in the page directory and the page table for faulty entries using the valid bit (0^{th} bit). If there is no physical memory (frame) associated with the page, an available frame is brought in, and the page-table entry is updated. If a new page-table page has to be initialized, a frame is brought in for that, and the new page table page and the directory are updated accordingly.

```
107 void PageTable::handle_fault(REGS * _r)
108 {
109     // Storing reason for page fault
110     unsigned long faulty_logical_address = read_cr2();
111     unsigned long error_word = _r->err_code;
112     if ((error_word & VALID_BIT) == 1) {
113         Console::puts("Protection fault\n");
114         assert(false);
115         return;
116     }
117
118     unsigned int vm_pool_index = 0;
119     unsigned int curr_vm_pool_count = current_page_table -> n_registered_vmpools;
120
121     for(;vm_pool_index<curr_vm_pool_count;vm_pool_index++){
122         if(current_page_table->registered_vmpools[vm_pool_index]!=NULL && current_page_table->registered_vmpools[vm_pool_index]->is_legitimate(faulty_logical_address)){
123             break;
124         }
125     }
126
127     if(vm_pool_index==curr_vm_pool_count){
128         Console::puts("Illegitimate Page\n");
129         assert(false);
130         return;
131     }
132
133     VMPool* curr_vm_pool = current_page_table->registered_vmpools[vm_pool_index];
134
135     unsigned long pde_indx = faulty_logical_address >> (12+10) ;
136     unsigned long pte_indx = (faulty_logical_address >> 12) & PTE_INDX_MASK;
137     unsigned long* pde = PageTable::PDE_address(faulty_logical_address);
138     unsigned long* pte_base_index = (unsigned long*)((pde_indx << 12) | PT_ADDR_MASK);
139
140     if((*pde & VALID_BIT) == 0){ //*curr_pd_address = pde
141         // pde invalid
142         *pde = ((curr_vm_pool->_frame_pool->get_frames(1)) << 12 ) | WRITE_BIT | VALID_BIT;
143         // get_frames returns a 20 bit value, which is the index of the start frame. Hence, << 12 to make it 32 bit.
144
145         // setting up new page table, and all its entries
146         for(unsigned int pd_offset=0;pd_offset<PAGE_SIZE/4;pd_offset++){
147             *(pte_base_index+pd_offset) = WRITE_BIT; //pd_offset*4 not needed as int is 4 bytes long.
148         }
149     }
150
151     // setting up new frame for the given faulty_logical_address
152     unsigned long new_frame_address = (curr_vm_pool->_frame_pool->get_frames(1)) << 12 ;
```

Every entry in a frame takes 32 bits, where:

- **Bits 31-12:** represent the address (of Page Table / Frame) stored in the entry.
- **Bits 11-9:** Available bits
- **Bits 8-3:** Reserved bits for various purposes.
- **Bit 2:** User/Supervisor Bit (0 -> Supervisor Bit, 1-> User Bit)
- **Bit 1**: Read/Write Bit (0-> Read-Only, 1-> can be written to)
- **Bit 0:** Valid Bit (0 -> Invalid entry, 1-> Valid Entry)

Test Screenshots / Output:

```
DONE WRITING TO MEMORY. Press keyboard to continue testing...
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
One second has passed
TEST PASSED.
YOU CAN SAFELY TURN OFF THE MACHINE NOW.
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.
========================================================================
```

```
Installing handler in IDT position 47
Installed exception handler at ISR <0>
Installed interrupt handler at IRQ <0>
Installed interrupt handler at IRQ <1>
Frame Pool initialized
Frame Pool initialized
Installed exception handler at ISR <14>
Initialized Paging System
Constructed Page Table object
Loaded page table
Enabled paging
WE TURNED ON PAGING!
If we see this message, the page tables have been
set up mostly correctly.
Hello World!
```

Frame Manager (implemented in cont_frame_pool.c) – MP2:

I use a bitmap to store the three states of a frame – Free, Used, Head of Sequence. 2 bits are used to represent the state of a frame as below.

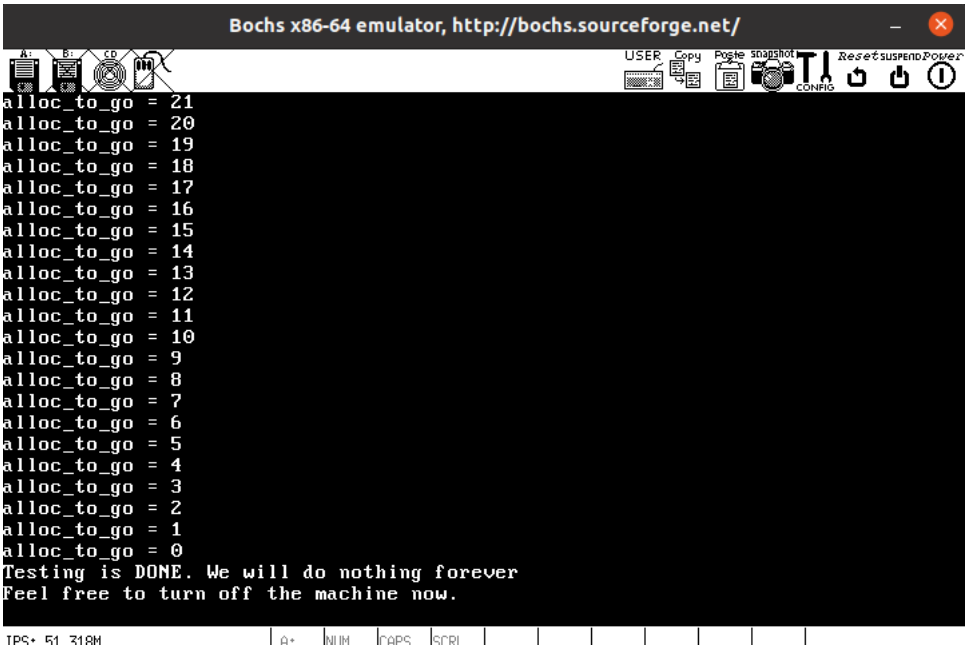
- 00 -> represents Free Frame
- 11 -> represents Used Frame
- 10 -> represents HoS (Head of Sequence) Frame

When a new frame pool is created, the pool gets appended to a static pool list, which is helpful for releasing any frame from any pool. New frames can be requested, and freed at any time, using two pools: kernel_mem_pool and process_mem_pool.

When requesting new frames, it is checked if contiguous space is available, otherwise an error is raised. If the space is available, status of the corresponding frames are updated to Used, and the first frame is marked as HoS. The address of the first frame number is returned.

When releasing frames, its pool, and base frame number are fetched first. If not found, an error is raised. Otherwise, the bitmap state of the frames in the pool are freed starting from the first frame until the state of a frame matches Used. (This is because all the frames are allocated memory in a contiguous fashion.)

Test Screenshots / Output:



<u>Documents changed:</u>

I have only changed the following files, as mentioned in the instructions:

- vm_pool.c
- vm_pool.h
- page_table.c
- page_table.h
- cont_frame_pool.c
- cont_frame_pool.h