MP4: Page Table Manager 2
Chonglin Zhang
UIN: 833003072
CSCE611: Operating System

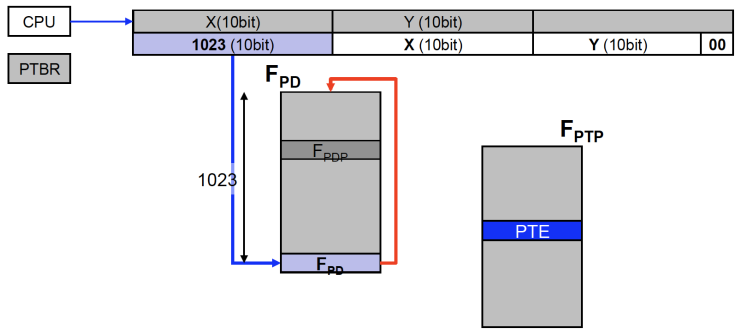
Assigned Tasks

To accommodate broader address spaces, we'll upgrade our page table management system. Due to the constraints of directly-mapped memory where our current page directory and table pages are housed, expansion is restricted. It's vital to shift the page table pages and perhaps the directory from kernel memory to process memory.

System Design

We allocate memory for the page table and directory from the process memory pool due to direct mapping limitations. A recursive page table is utilized for addressing, given the CPU's random logical address generation. Using figure 1 from the lecture note, this figure illustrates how in a recursive setup, the page directory's final entry references itself. In this setup, the virtual memory-stored directory is represented as |1023 : 10|1023 : 10|offset : 12|, and the table as |1023 : 10|X : 10|Y : 10|0 : 2|.

Figure 1: recursive table lookup



Code Description

page table.c: constructor: The page table constructor assigns addresses and activates the present bits, even as page directory entries remain unmarked. Moreover, the recursive configuration ensures the final entry references itself.

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

    for (int i = 0; i < shared_frames; i++) {
        unsigned long mask = i * PAGE_SIZE;
        page_table[i] = mask | 3;
        page_directory[i] = (i == 0) ? (unsigned long) page_table | 3 : mask | 2;
    }
    page_directory[shared_frames - 1] = (unsigned long) (page_directory) | 3;
   //assert(false);
   Console::puts("Constructed Page Table object\n");
}
```

page_table.c: load(): After the configuration of the page table is completed, the page directory index is deposited in the CR3 register.

```
void PageTable::load()
{
    current_page_table = this;
    write_cr3((unsigned long) page_directory);
    //assert(false);
    Console::puts("Loaded page table\n");
}
```

page_table.c: enable_paging() : The method initiates paging by configuring the CR0 register.

```
void PageTable::enable_paging()
{
     paging_enabled = 1;
     write_cr0(read_cr0() | 0x80000000);
    //assert(false);
    Console::puts("Enabled paging\n");
}
```

page_table.c: handle_fault() : This approach addresses page faults by identifying and managing the faulty addresses:

1. Start by extracting the address associated with the page fault.
2. Identify an available frame from the FramePool for address mapping in the page table and directory. Use the recursive strategy for logical addresses and frame mapping to resulting offsets.
3. Conclude the Page Fault handler.

```
void PageTable::handle_fault(REGS *_r)
{
    if (!(_r->err_code & 1)) {

        // Add VMPool legitimacy check
        VMPool *temp = PageTable::HEAD;
        bool foundLegitimate = false;
        do {
            if (temp && temp->is_legitimate(read_cr2())) {
                foundLegitimate = true;
                break;
            }
            if(temp) temp = temp->next;
        } while (temp);

        assert(foundLegitimate || !temp);

        unsigned long *dir = (unsigned long *) 0xFFFFF000;
        unsigned long dirIndex = (read_cr2() >> 22) & 0x3FF;
        unsigned long pageIndex = (read_cr2() >> 12) & 0x3FF;
        unsigned long *tbl = (unsigned long *) (0xFFC00000 | (dirIndex << 12));

        if (!(dir[dirIndex] & 1)) {
            dir[dirIndex] = (process_mem_pool->get_frames(1) * PAGE_SIZE) | 3;
            for (unsigned int i = 0; i < shared_frames; i++) {
                tbl[i] = 0 | 4;
            }
        }
        tbl[pageIndex] = (process_mem_pool->get_frames(1) * PAGE_SIZE) | 3;
    }
    //assert(false);
    Console::puts("handled page fault\n");
}
```

VM Pool Registration: Virtual memory allocation sources its regions from the virtual memory pool. The page table recognizes VM pools via the PageTable::register pool() function and keeps a LinkedList of these pools. When a page fault arises, the validity of the faulty address is assessed by determining if it falls within the base address to base address + limit range.

```
void PageTable::register_pool(VMPool * _vm_pool)
{
    VMPool **indirect = &PageTable::HEAD;
    while (*indirect) {
        indirect = &((*indirect)->next);
    }
    *indirect = _vm_pool;

    Console::puts("registered VM pool\n");
}

bool VMPool::is_legitimate(unsigned long _address) {
    bool init_limit = _address >= base_addr;
    bool max_limit = _address < (base_addr + size);
    return init_limit && max_limit;
}
```

page_table.c: free page: Each page determines its frame number from the address and invokes the release function from the Process pool. Subsequently, the page is set as invalid, and the TLB is cleared.

```
void PageTable::free_page(unsigned long _page_no) {
    unsigned long pt_offset = (_page_no >> 22) << 12;
    auto * page_table = (unsigned long *) (0xFFC00000 | pt_offset);
    unsigned long idx = _page_no >> 12 & 0x3FF;

    unsigned long frame_addr = page_table[idx];
    unsigned long frame = frame_addr / PAGE_SIZE;
    process_mem_pool->release_frames(frame);

    page_table[idx] = 2;
    //assert(false);
    //Console::puts("freed page\n");
}
```

vm pool.c: allocate:Regions reside in an array, with the ith index representing the ith region. Each array entry consists of a structure detailing the base address and region size. When a new region is requested, it's added to the array. The size of the region is determined by multiplying the needed pages by the Page Size.

```
unsigned long VMPool::allocate(unsigned long _size) {
    if (!_size) {
        return 0;
    }

    assert(region_no < MAX_REGIONS);

    unsigned long framesRequired = (_size + PageTable::PAGE_SIZE - 1) /
        PageTable::PAGE_SIZE;

    unsigned long prevRegionEnd = (region_no > 0) ? allocation[region_no -
        1].base_addr + allocation[region_no - 1].size : base_addr;

    allocation[region_no].base_addr = prevRegionEnd;
    allocation[region_no].size = framesRequired * PageTable::PAGE_SIZE;
    region_no++;

    return prevRegionEnd;
}
```

vm pool.c: release: Upon a request to free a region, the relevant region is located in the array, revealing its base address and page count. Each page undergoes the freeing process. The function pinpoints the initial frame and releases the frames tied to that page. Subsequently, the page's entry is set to invalid, and the TLB is refreshed.

```
void VMPool::release(unsigned long _start_address) {
    int regionIndex = 0;
    while (regionIndex < MAX_REGIONS && allocation[regionIndex].base_addr !=
        _start_address) {
        ++regionIndex;
    }

    assert(regionIndex < MAX_REGIONS);

    unsigned long total_pages = allocation[regionIndex].size / PageTable::PAGE_SIZE;

    for (unsigned long pg = 0; pg < total_pages; pg++) {
        page_table->free_page(_start_address + pg * PageTable::PAGE_SIZE);
    }

    // Shift the allocations after the removed one
    for (int j = regionIndex; j < region_no - 1; j++) {
        allocation[j] = allocation[j + 1];
    }

    --region_no;
    page_table->load();
    //assert(false);
    Console::puts("Memory region has been released.\n");
}
```

Testing

The figure 2 and figure 3 display the kernel output and console output.

Figure 2:kernel output

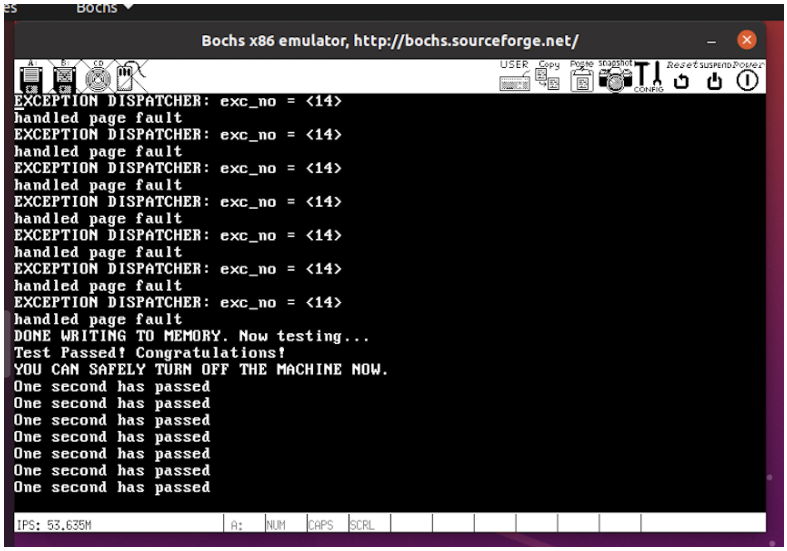
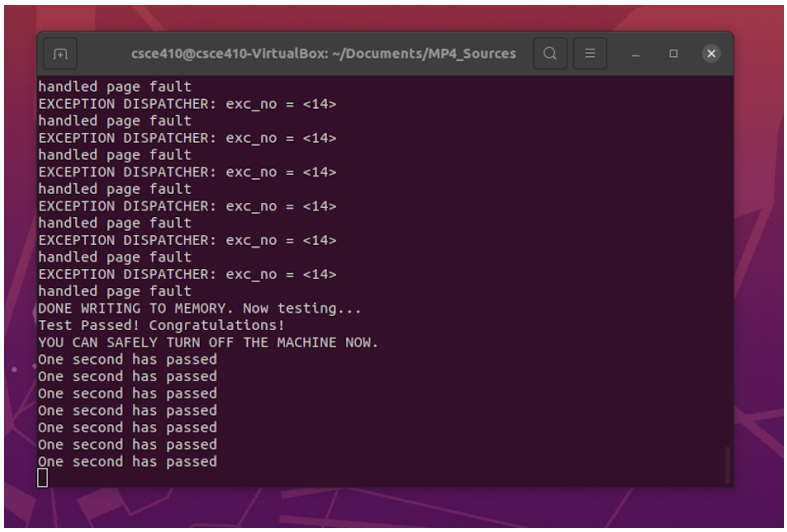


Figure 3: console output



Test: passed
1. Case 01: When registering a pool, the FAULT ADDRESS 4MB is within the limits. The validation function confirms it, processes the fault, and records the related address pool.
2. Case 02: When registering a pool, the FAULT ADDRESS 4MB is outside the limits. The validation check doesn't pass.

Test(fault handler): passed
We evaluated the fault handler using the 4MB and 8MB fault addresses.