MP4: Page Manager II

Ashutosh Punyani
UIN: 834006613
CSCE611: Operating System

Assigned Tasks

- **Part 1:** Support for Large Address Spaces - Completed.

  I have utilized the recursive page table lookup method to map the logical address space to the physical frames. When a logical address is in the following format:

  $$\begin{aligned} X &: 10 \\ Y &: 10 \\ \text{offset} &: 12 \end{aligned}$$

  The page directory can be accessed as follows:

  $$\begin{aligned} 1023 &: 10 \\ 1023 &: 10 \\ \text{page directory} &: 10 \\ \text{offset} &: 2 \end{aligned}$$

  Similarly, the page table can be accessed as follows:

  $$\begin{aligned} 1023 &: 10 \\ \text{page directory} &: 10 \\ \text{page table} &: 10 \\ \text{offset} &: 2 \end{aligned}$$

- **Part 2:** Preparing class `PageTable` to handle Virtual Memory Pools - Completed.

  The function `PageTable::register_pool()` establishes a connection between the Page Table and the Virtual Memory Pool by maintaining a linked list of Virtual Memory Pools. This linked list contains information about the regions available for virtual memory allocation, including the base address, size of each region, Page Table, and Frame Pool to which they belong. When a Page Fault occurs in a Page Table, it checks if the fault address falls within the range of the base address and base address + size to ensure its validity in the Virtual Memory Pool to which the Page Table belongs. If the fault address satisfies this condition, then page fault is handled. The Virtual Memory Pool keeps track of regions used for memory allocation, and the Page Table utilizes this information to validate and manage Page Faults.

- **Part 3:** An Allocator and De-allocator for Virtual Memory - Completed.

Virtual memory allocates and de-allocates memory in integral multiples of pages using `VMPool::allocate` and `VMPool::release` respectively in the specific virtual memory pool. I have used the array `regions` of data type `region_data` which stores the base address in `base_addr` and size in `size` for each region.

System Design

The main objective of machine problem 4 is the implementation of a virtual memory manager and a basic virtual memory allocator, with the goal of supporting a large address space. We use a recursive page table lookup approach in this machine problem.

Code Description

During the implementation of this machine problem, I made changes to the following six files:

1. **cont_frame_pool.H**
2. **cont_frame_pool.C**
3. **page_table.H**
4. **page_table.C**
5. **vm_pool.H**
6. **vm_pool.C**

I used the same code for `cont_frame_pool.C`, `cont_frame_pool.H` that I implemented for MP2. I made changes to my existing `page_table.C` and `page_table.H` that I have implemented in MP3. Additionally, I wrote definitions for the functions defined in `vm_pool.H` in `vm_pool.C`.

1. **page_table.H**

In `page_table.H`, I have added two static variables `vm_pool_head` and `vm_pool_current` both of type `VMPool*` to store the head and current element of linked list of virtual memory pools. It also includes two functions: `PDE_address(addr)` returns the address of the Page Directory Entry (PDE) location for a given address, and `PTE_address(addr)` returns the address of the Page Table Entry (PTE) location for a given address.

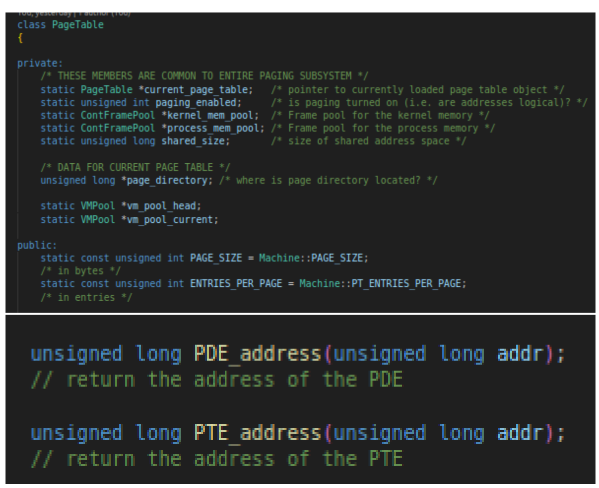


Figure 1: **page_table.H**

2. **page_table.C**

   (a) **init_paging** : This function is used to initialize static private data members in a class, including the kernel frame pools, memory frame pools, and the shared size for the page table.

```
void PageTable::init_paging(ContFramePool *_kernel_mem_pool,
                            ContFramePool *_process_mem_pool,
                            const unsigned long _shared_size)
{
    // initlizaing basic data structure for paging
    Console::puts("Initialized Paging System Start\n");
    kernel_mem_pool = _kernel_mem_pool;
    process_mem_pool = _process_mem_pool;
    shared_size = _shared_size;
    Console::puts("Initialized Paging System End\n");
}
```

Figure 2: **init_paging**

   (b) **PageTable (Class Constructor)** : The constructor is used to construct the page table object. It initializes the first page directory by assigning the free frame from the kernel frame pool and marking it as valid (present). It also initializes the first page table by assigning the free frame from the process frame pool. The first directory entry holds the first page table, while all the remaining directories are marked as invalid (not present), except for the last page directory entry, which points to the first page directory entry for the recursive page table lookup. Finally, it initializes variables for virtual memory pool management, such as `vm_pool_head` and `vm_pool_current`.

```
PageTable::PageTable()
{
    // Constructor for PageTable class
    Console::puts("Constructed Page Table object Start\n");
    unsigned long page_directory_frame_number = kernel_mem_pool->get_frames(1);
    page_directory = (unsigned long *)(page_directory_frame_number * PAGE_SIZE);
    // last pde as valid and pointing to first pde for recurive table look up
    page_directory[ENTRIES_PER_PAGE - 1] = (((unsigned long)page_directory) | 0x3);

    unsigned long page_table_frame_number = process_mem_pool->get_frames(1);
    unsigned long *page_table = (unsigned long *)(page_table_frame_number * PAGE_SIZE);

    // mappng the first 4MB of memory
    for (unsigned long i = 0, physical_address = 0; i < ENTRIES_PER_PAGE; i++, physical_address += PAGE_SIZE)
    {
        // attribute set to: supervisor level,
        // read/write, present(011 in binary)
        page_table[i] = physical_address | 0x3;
    }

    // attribute set to: supervisor level,
    // read/write, present(011 in binary)
    // first pde as valid pointing to page table
    page_directory[0] = (((unsigned long)page_table) | 0x3);
    // page_directory[ENTRIES_PER_PAGE - 1] = (((unsigned long)page_directory) | 0x3);

    for (unsigned int i = 1; i < ENTRIES_PER_PAGE - 1; i++)
    {
        // attribute set to: supervisor level,
        // read/write, not present(010 in binary)
        page_directory[i] = 0 | 0x2;
    }

    vm_pool_head = NULL;
    vm_pool_current = NULL;
    Console::puts("Constructed Page Table object End\n");
}
```

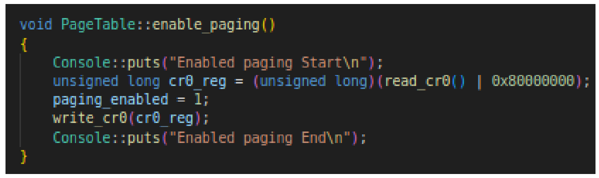
Figure 3: **PageTable (Class Constructor)**

   (c) **load** : This function sets the `current_page_table` pointer to the current instance of the `PageTable` object. Then, it loads the current page directory into register CR3 using `write_cr3()`. The page table is then loaded.

```
void PageTable::load()
{
    // Load the page table into the CR3 register
    Console::puts("Loaded page table Start\n");
    current_page_table = this;
    write_cr3((unsigned long)current_page_table->page_directory);
    Console::puts("Loaded page table End\n");
}
```

Figure 4: **load**

   (d) **enable_paging** : This function is used to enable paging by setting the paging bit (bit 31) of CR0 to 1 using `read_cr0` to read the contents of CR0 and `write_cr0` to write the contents to CR0. It also sets the boolean `paging_enabled` to true. Before enabling paging, the page directory and page table should be set up and loaded correctly.



```
void PageTable::enable_paging()
{
    Console::puts("Enabled paging Start\n");
    unsigned long cr0_reg = (unsigned long)(read_cr0() | 0x80000000);
    paging_enabled = 1;
    write_cr0(cr0_reg);
    Console::puts("Enabled paging End\n");
}
```

Figure 5: **enable_paging**

(e) **handle_fault** : This function is utilized to handle raised faults. The function retrieves the error code from the provided REGS structure. It examines the error code received from the register. If the least significant bit of the error code is 0, indicating a page fault, the function proceeds to handle the fault. Subsequently, it retrieves the faulty address from the CR2 register using the `read_cr02` function. Additionally, it reads the current page directory list and directory location from CR3 using the `read_cr3` function and the faulty address, respectively. It checks whether the faulty address is a legitimate virtual memory address by iterating over the virtual memory pool list. If it is not legitimate, then the execution is halted with a message, 'Not a legitimate address'. If the page directory entry for the directory location is not found, indicating a problem with the directory, the function creates a page table and allocates a new page table from the process memory pool. It updates the page directory entry accordingly and sets all the page table entries as not being present. If the page directory entry is found, indicating an existing issue with the page table, the function allocates a frame from the process memory pool and updates the corresponding page table entry. In the event of a different scenario than the aforementioned, the function halts execution with a message stating, 'Something went wrong'.

```
void PageTable::handle_fault(REGS *_r)
{
    // Handle page faults
    Console::puts("handle_fault Start\n");
    unsigned long err_code = _r->err_code;
    if ((err_code & 0x1) == 0x0)
    {
        Console::puts("handle_fault err_occuured\n");
        unsigned long faulty_address = (unsigned long)(read_cr2());
        unsigned long *page_directory_list = (unsigned long *)(read_cr3());
        unsigned long directory_location = (faulty_address) >> 22;
        bool is_legitimate_vm_address = false;
        VMPool *iterartor;
        for (iterartor = vm_pool_head; iterartor != NULL; iterartor = iterartor->next)
        {
            bool temp_bool = iterartor->is_legitimate(faulty_address) == true;
            if (temp_bool)
            {
                is_legitimate_vm_address = true;
                break;
            }
        }
        if (!is_legitimate_vm_address && iterartor != NULL)
        {
            Console::puts("Not Legitimate address\n");
            assert(false);
        }
        if ((page_directory_list[directory_location] & 0x1) == 0x0)
        {
            Console::puts("directory issue and new page table");
            Console::puts("\n");
            unsigned long new_page_table_frame_number = process_mem_pool->get_frames(1);
            unsigned long *new_page_table = (unsigned long *)((new_page_table_frame_number * PAGE_SIZE));

            unsigned long *page_directory_entry_addr = (unsigned long *)(0xFFFFF << 12);
            // attribute set to: supervisor level,
            // read/write, not present(010 in binary)
            page_directory_entry_addr[directory_location] = (unsigned long)new_page_table | 0x3;

            // initializing the page table enteries
            for (unsigned int i = 0; i < ENTRIES_PER_PAGE; i++)
```

```
        {
            Console::puts("directory issue and new page table");
            Console::puts("\n");
            unsigned long new_page_table_frame_number = process_mem_pool->get_frames(1);
            unsigned long *new_page_table = (unsigned long *)((new_page_table_frame_number * PAGE_SIZE));

            unsigned long *page_directory_entry_addr = (unsigned long *)(0xFFFFF << 12);
            // attribute set to: supervisor level,
            // read/write, not present(010 in binary)
            page_directory_entry_addr[directory_location] = (unsigned long)new_page_table | 0x3;

            // initializing the page table enteries
            for (unsigned int i = 0; i < ENTRIES_PER_PAGE; i++)
            {
                // attribute set to: supervisor level,
                // read/write, not present(010 in binary)
                new_page_table[i] = 0 | 0x2;
            }
            unsigned long physical_frame_number = process_mem_pool->get_frames(1);
            unsigned long page_table_entry_location = (faulty_address & (0x03FF << 12)) >> 12;
            new_page_table[page_table_entry_location] = (unsigned long)(physical_frame_number * PAGE_SIZE) | 0x3;
        }
        else
        {
            Console::puts("existing page table issue");
            Console::puts("\n");
            unsigned long *existing_page_table = (unsigned long *)((0x3FF << 22) | (directory_location << 12));
            unsigned long physical_frame_number = process_mem_pool->get_frames(1);
            unsigned long page_table_entry_location = (faulty_address & (0x03FF << 12)) >> 12;
            existing_page_table[page_table_entry_location] = (unsigned long)(physical_frame_number * PAGE_SIZE) | 0x3;
        }

        Console::puts("resolved page fault\n");
    }
    else
    {
        Console::puts("Something went wrong\n");
        assert(false);
    }
    Console::puts("handle_fault End\n");
}
```

Figure 6: **handle_fault**

(f) **register_pool** : This function is used to register a VMPool instance into a linked list of virtual memory pools. It assigns the instance to the head of the linked list of virtual memory pools if the list is empty or appends it to the end of the list if it's not.

```
void PageTable::register_pool(VMPool *_vm_pool)
{
    // Register a VMPool instance
    Console::puts("registered VM pool - start\n");
    if (vm_pool_head == NULL)
    {
        Console::puts("Empty Head.\n");
        // if list is empty assign it to head
        vm_pool_head = _vm_pool;
    }
    else
    {
        Console::puts("Non Empty Head.\n");
        // if list is not empty make next point to it
        vm_pool_current->next = _vm_pool;
    }
    // in both cases current next should be null and curent should be point to cureent _vm_pool
    vm_pool_current = _vm_pool;
    vm_pool_current->next = NULL;
    Console::puts("registered VM pool - end\n");
}
```

Figure 7: **register_pool**

(g) **free_page** : This function is responsible for freeing a page and flushing the Translation Lookaside Buffer (TLB). It determines the frame number associated with the page based on its address and invokes the corresponding `release_frame` function from the process pool to release that frame. Additionally, it marks the page as not present in the page table entry. Finally, it ensures that the TLB is updated to reflect the changes in memory mapping.

```cpp
void PageTable::free_page(unsigned long _page_no)
{
    // Free a page and flush the TLB
    Console::puts("freed page - start\n");
    unsigned long page_directory_location = PDE_address(_page_no);
    unsigned long page_table_location = PTE_address(_page_no);
    unsigned long *page_directory_entry = (unsigned long *)((0x000003FF << 22) | (page_directory_location * PAGE_SIZE));
    unsigned long frame_no = (page_directory_entry[page_table_location] & 0xFFFFF000) / PAGE_SIZE;
    process_mem_pool->release_frames(frame_no);
    page_directory_entry[page_table_location] = page_directory_entry[page_table_location] | 2;
    // Flushing the TLB
    load();
    Console::puts("freed page - end\n");
}
```

Figure 8: **free_page**

(h) **PDE_address** : This function returns the Page Directory Entry (PDE) Location.

```cpp
// Return the address of the Page Directory Entry (PDE) Location
unsigned long PageTable::PDE_address(unsigned long addr)
{
    unsigned long page_directory_location = (addr & 0xFFC00000) >> 22;
    return page_directory_location;
}
```

Figure 9: **PDE_address**

(i) **PTE_address** : This function returns the Page Table Entry (PTE) Location.

```cpp
// Return the address of the Page Table Entry (PTE) Location
unsigned long PageTable::PTE_address(unsigned long addr)
{
    unsigned long page_table_location = (addr & 0x003FF000) >> 12;
    return page_table_location;
}
```

Figure 10: **PTE_address**

3. **vm_pool.H**

In `vm_pool.H`, a data type called `region_data` is defined with attributes `base_addr` and `size` for specifying the base address and size of each region in a virtual memory pool. Other variables include `base_address`, `size`, `frame_pool`, `page_table`, and a list of memory pools using `region_data`. There are variables for storing the available virtual memory size in `available_size` and the total number of regions in `total_count`. The `next` attribute of data type `VMPool *` is used to store the address of the next virtual memory pool region and is initialized as `NULL`.

```cpp
class VMPool
{ /* Virtual Memory Pool */
        You, 3 days ago • method definitions for vm pool added
private:
    /* -- DEFINE YOUR VIRTUAL MEMORY POOL DATA STRUCTURE(s) HERE. */

    You, yesterday | 1 author (You)
    class region_data
    {
    public:
        unsigned long base_addr;
        unsigned long size;
    };

    unsigned long base_address;
    unsigned long size;
    ContFramePool *frame_pool;
    PageTable *page_table;

    region_data *regions;
    unsigned long available_size;
    unsigned long total_count;

public:
    VMPool *next = NULL;
    VMPool(unsigned long _base_address,
```

Figure 11: **vm_pool.H**

4. **vm_pool.C**

(a) **VMPool (Class Constructor)** : The constructor is used to initialize private data members such as `base_address`, `size`, `page_table`, `frame_pool` of virtual memory pool in a class. Additionally, I have initialized the pointer to the next virtual memory pool as `NULL`, `available_size` to the size of virtual memory pool, `total_count` to number of region in a virtual memory pool, and reserved the first region for storing the region data.

```
VMPool::VMPool(unsigned long _base_address,
               unsigned long _size,
               ContFramePool *_frame_pool,
               PageTable *_page_table)
{
    // Constructor for the VMPool class
    Console::puts("Constructed VMPool object - start.\n");
    base_address = _base_address;
    size = _size;
    frame_pool = _frame_pool;
    page_table = _page_table;

    next = NULL;
    available_size = _size;
    total_count = 0;

    page_table->register_pool(this);

    // Using the first pool to store region data
    region_data *temp_region = (region_data *)base_address;
    temp_region[0].base_addr = base_address;
    temp_region[0].size = PageTable::PAGE_SIZE;
    regions = temp_region;
    available_size -= PageTable::PAGE_SIZE;
    total_count += 1;

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

Figure 12: **VMPool (Class Constructor)**

(b) **allocate** : This function checks for sufficient memory and allocates an area of the requested size if available; otherwise, it stops the execution with the message 'No free size available'. The available memory size is updated to reflect the allocation of the remaining size and increments the number of regions. Finally, it returns the location address of the given region.

```
unsigned long VMPool::allocate(unsigned long _size)
{
    // Allocate a region of memory
    Console::puts("Allocated region of memory - start. \n");
    if (available_size < _size)
    {
        Console::puts("No free size available.\n");
        assert(false);
    }
    unsigned number_of_pages = _size / PageTable::PAGE_SIZE;
    number_of_pages = _size % PageTable::PAGE_SIZE > 0 ? number_of_pages + 1 : number_of_pages;
    regions[total_count].base_addr = regions[total_count - 1].base_addr + regions[total_count - 1].size;
    regions[total_count].size = number_of_pages * PageTable::PAGE_SIZE;
    total_count += 1;
    available_size -= number_of_pages * PageTable::PAGE_SIZE;

    Console::puts("Allocated region of memory - end.\n");
    return regions[total_count - 1].base_addr;
}
```

Figure 13: **allocate**

(c) **release** : This function releases a memory area based on the given start address. It first checks whether the given start address is legitimate or not. If it is, the function searches for the start address among all the available used regions. If it is not found, the program stops execution with the message 'No such region found starting with this start address.' If it is found, the function frees the corresponding pages in the page table. The function adjusts the array by shifting subsequent regions, updates the count and available memory size, and finally flushes the TLB. If the address is not legitimate, the function halts execution with the message 'Not Legitimate - start address'.

```cpp
void VMPool::release(unsigned long _start_address)
{
    // Release a region of memory
    Console::puts("Released region of memory - start.\n");
    if (!is_legitimate(_start_address))
    {
        Console::puts("Not Legitimate - start address");
        assert(false);
    }
    unsigned long region_relase_index = -1;
    for (unsigned long i = 1; i < total_count; i++)
    {
        if (regions[i].base_addr == _start_address)
        {
            region_relase_index = i;
            break;
        }
    }
    if (region_relase_index < 0)
    {
        Console::puts("No such region found starting with this start address");
        assert(false);
    }
    else
    {
        unsigned long curr_addr = _start_address;
        unsigned long number_of_pages = regions[region_relase_index].size / PageTable::PAGE_SIZE;
        for (unsigned long i = 0; i < number_of_pages; i++)
        {
            page_table->free_page(curr_addr);
            curr_addr += PageTable::PAGE_SIZE;
        }

        for (unsigned long i = region_relase_index; i < total_count; i++)
        {
            regions[i] = regions[i + 1];
        }

        total_count -= 1;
        available_size += regions[total_count].size;
```

```cpp
        assert(false);
    }
    unsigned long region_relase_index = -1;
    for (unsigned long i = 1; i < total_count; i++)
    {
        if (regions[i].base_addr == _start_address)
        {
            region_relase_index = i;
            break;
        }
    }
    if (region_relase_index < 0)
    {
        Console::puts("No such region found starting with this start address");
        assert(false);
    }
    else
    {
        unsigned long curr_addr = _start_address;
        unsigned long number_of_pages = regions[region_relase_index].size / PageTable::PAGE_SIZE;
        for (unsigned long i = 0; i < number_of_pages; i++)
        {
            page_table->free_page(curr_addr);
            curr_addr += PageTable::PAGE_SIZE;
        }

        for (unsigned long i = region_relase_index; i < total_count; i++)
        {
            regions[i] = regions[i + 1];
        }

        total_count -= 1;
        available_size += regions[total_count].size;

        regions[region_relase_index].base_addr = regions[total_count].base_addr;
        regions[region_relase_index].size = regions[total_count].size;
    }
    // Flushing the TLB
    page_table->load();
    Console::puts("Released region of memory - end.\n");
}
```

Figure 14: **release**

(d) **is_legitimate** : This function checks if a given address is within any allocated memory region. It iterates through the memory regions and compares the address with the base address and size of each region. If a match is found, it returns true, indicating the address is legitimate. If no match is found, it returns false. The function outputs messages indicating the start and end of the address legitimacy check.

```cpp
bool VMPool::is_legitimate(unsigned long _address)
{
    // Checking whether the address is part of an allocated region
    Console::puts("Checked whether address is part of an allocated region - start.\n");
    for (unsigned long i = 0; i < total_count; i++)
    {
        if (regions[i].base_addr <= _address && regions[i].base_addr + regions[i].size >= _address)
        {
            Console::puts("Legitimate.\n");
            Console::puts("Checked whether address is part of an allocated region - end.\n");
            return true;
        }
    }
    Console::puts("Not Legitimate.\n");
    Console::puts("Checked whether address is part of an allocated region - end.\n");
    return false;
}
```

Figure 15: **is_legitimate**

Testing

During the development of the code, I wrote several `Console:puts()` and `Console:putui()` statements to pinpoint where my code was breaking and to understand if the logic was incorrect or not performing as expected. During testing, the program was running infinitely. Additionally, test case 0 was failing. After making necessary changes and analyzing the console debug logs, the code ran successfully. I removed all Console statements and reverted Kernel.C to its original state. Initially, the page table implementation was tested to ensure its functionality. Subsequently, the macro known as `_TEST_PAGE_TABLE_` was commented for testing the virtual memory (VM) pools. The following screenshots illustrate this process.

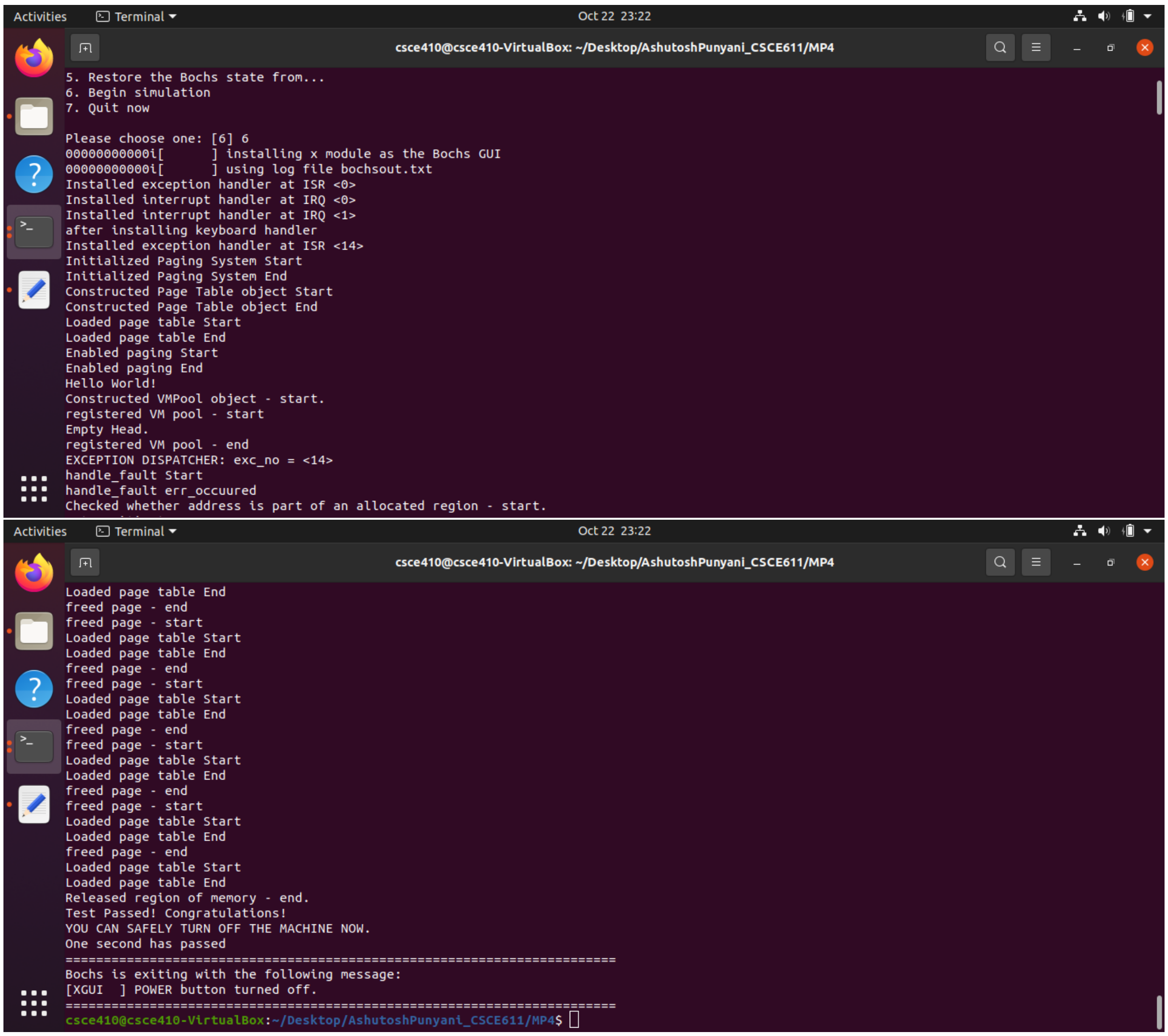


Figure 16: **Testing (with commented _TEST_PAGE_TABLE_ define)**

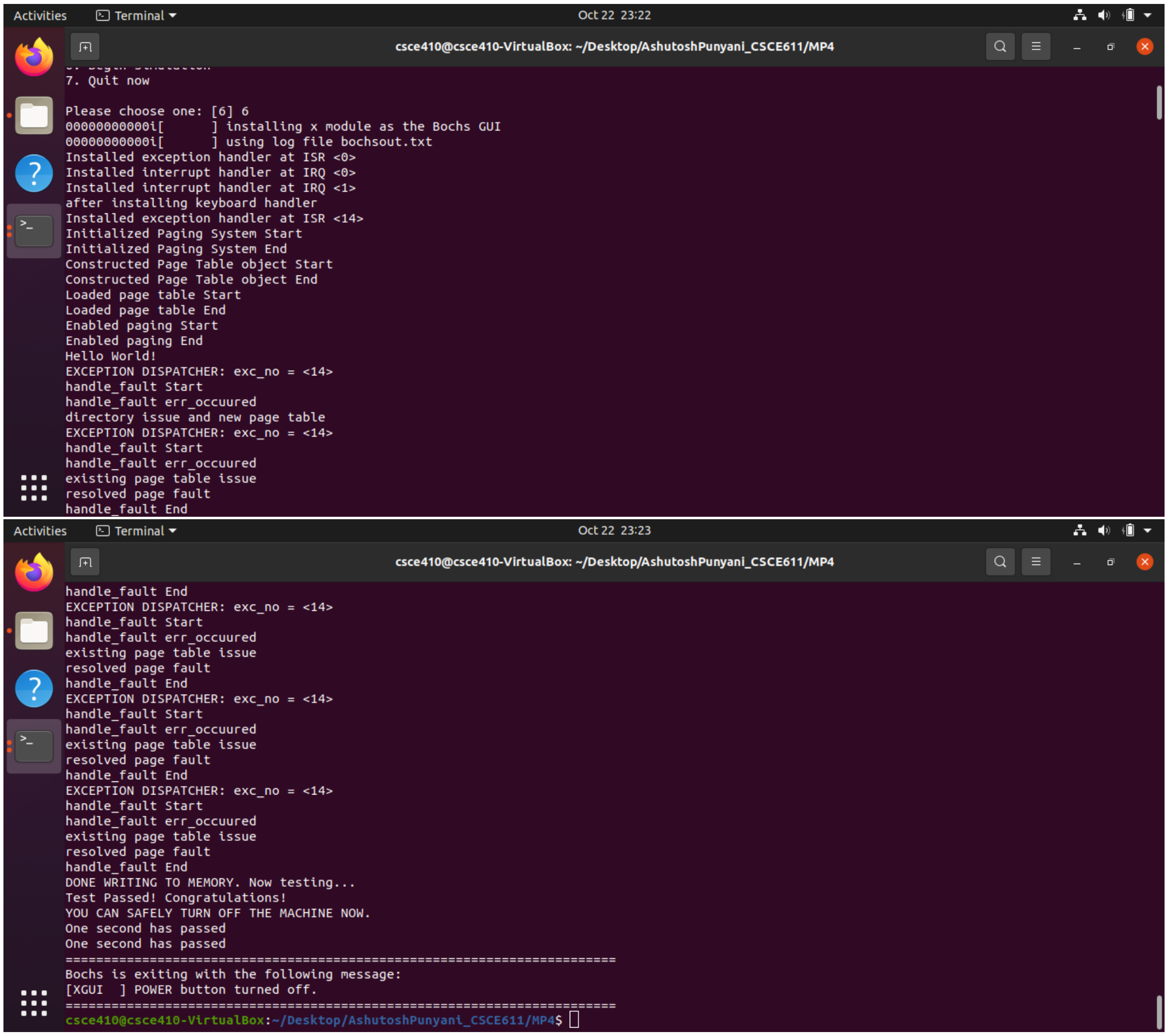


Figure 17: **Testing (without commented _TEST_PAGE_TABLE_ define)**