MP3: Page Manager I

Vishnuvasan Raghuraman
UIN: 234009303
CSCE611: Operating System

Assigned Tasks

Main: Completed.

System Design

The objective of this machine problem is to set up and initialize the paging system and the page table infrastructure for a single address space using the paging mechanism on the x86 architecture. The x86 architecture follows a two-level hierarchical paging scheme. A multilevel paging scheme consists of two or more page tables in a hierarchical manner. In this scheme, the level 1 page table consists of pointers directing to a level 2 page table. Similarly, entries within the level 2 page tables act as pointers to a level 3 page table, continuing this hierarchical structure. At the last level, the page table entries store information about the actual frame information. Notably, the level 1 page table encompasses a single page table, with its address stored in the Page Table Base Register (PTBR).

For a 32-bit logical address, it's divided into three parts: a 10-bit page table number, a 10-bit page number, and a 12-bit offset. The Page Table Base Register (PTBR) points to the start of the page directory. The page table number serves as an index to locate the corresponding entry in the page directory. This entry holds the reference to the start of the page table page linked with the given page table number. Using the page number, we index into the page table page to locate the page table entry. Eventually, by combining the offset with the frame number stored in the page table entry, we obtain the physical address. This process effectively maps the logical address to its corresponding physical address.

The mapping from logical to physical addresses is essential for understanding how memory is managed in the system. Here's the layout of the memory:

- Total memory available: 32MB
- Kernel memory: 4 MB (directly mapped to physical memory)
- Process memory pool: 28MB (freely mapped to physical memory).
- Frame size: 4 B
- Inaccessible memory region: 15MB to 16MB

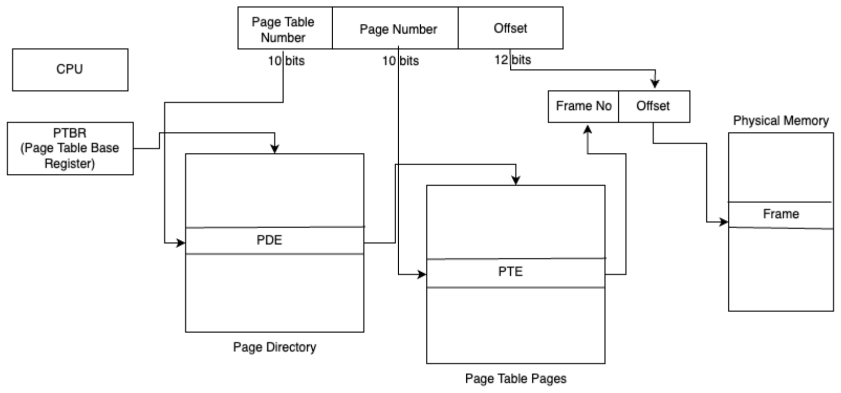


Figure 1: Memory design

A page fault triggers the operating system to load the required page into a frame in physical memory, establishing the necessary mapping between virtual memory and physical memory to allow the program to continue execution seamlessly.

Code Description

A Silicon(M2) Mac was used for this assignment and the native environment setup for such a machine from Piazza was followed. The below files were modified for this particular MP.

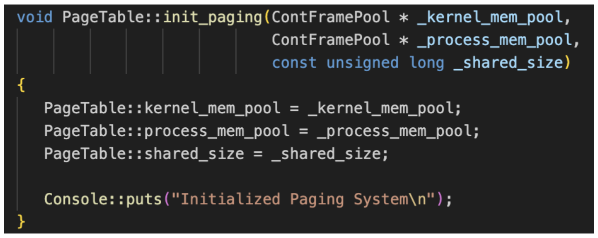
- page_table.C - implemented functionalities responsible for initializing page table, loading page table, handling page faults, etc.,

The below files from the previous MP(MP2) were used for this MP3 as a continuation.

- cont_frame_pool.C
- cont_frame_pool.H

The functions implemented are given as below:

page_table.C: init_paging : This method is responsible for setting up the initial values of the private variables within the PageTable class. Additionally, it configures the global parameters required for the paging subsystem to function properly.

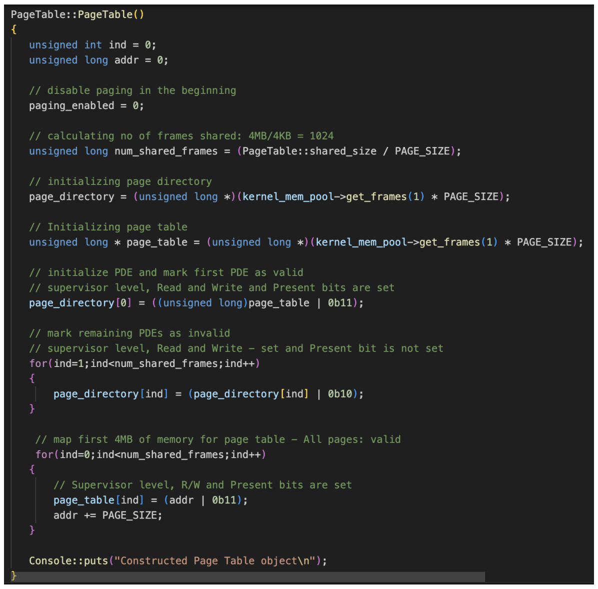


```
void PageTable::init_paging(ContFramePool * _kernel_mem_pool,
                            ContFramePool * _process_mem_pool,
                            const unsigned long _shared_size)
{
    PageTable::kernel_mem_pool = _kernel_mem_pool;
    PageTable::process_mem_pool = _process_mem_pool;
   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

    Console::puts("Initialized Paging System\n");
}
```

page_table.C: PageTable : This method serves as the constructor for the PageTable class. It handles the setup of entries within both the page directory and the page table.

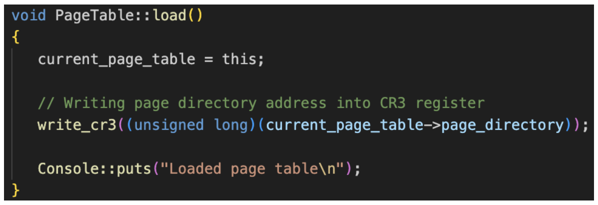
To initialize the page directory, the constructor allocates a single frame from the kernel frame pool. The first entry of the page directory is flagged as 'valid', while the rest are marked as 'invalid' using bitwise operations.

Following this, the constructor proceeds to initialize the page table by allocating another frame from the kernel frame pool. The page table entries corresponding to the shared memory portion (the initial 4MB directly-mapped region) are then marked as 'valid' using bitwise operations.



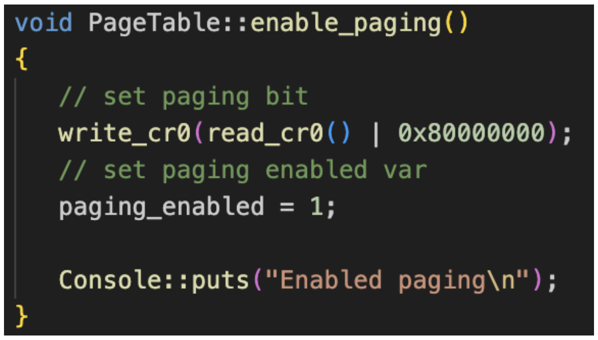
```
PageTable::PageTable()
{
    unsigned int ind = 0;
    unsigned long addr = 0;

    // disable paging in the beginning
    paging_enabled = 0;

    // calculating no of frames shared: 4MB/4KB = 1024
    unsigned long num_shared_frames = (PageTable::shared_size / PAGE_SIZE);

    // initializing page directory
    page_directory = (unsigned long *)(kernel_mem_pool->get_frames(1) * PAGE_SIZE);

    // Initializing page table
    unsigned long * page_table = (unsigned long *)(kernel_mem_pool->get_frames(1) * PAGE_SIZE);

    // initialize PDE and mark first PDE as valid
    // supervisor level, Read and Write and Present bits are set
    page_directory[0] = ((unsigned long)page_table | 0b11);

    // mark remaining PDEs as invalid
    // supervisor level, Read and Write - set and Present bit is not set
    for(ind=1;ind<num_shared_frames;ind++)
    {
        page_directory[ind] = (page_directory[ind] | 0b10);
    }

     // map first 4MB of memory for page table - All pages: valid
     for(ind=0;ind<num_shared_frames;ind++)
    {
        // Supervisor level, R/W and Present bits are set
        page_table[ind] = (addr | 0b11);
        addr += PAGE_SIZE;
    }

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

page_table.C: load : This function is employed to load a page table by storing the address of the page directory in the Page Table Base Register (PTBR), which is achieved by writing the address into the control register CR3. In essence, this process involves loading a new page table and setting it as the currently active table.



```
void PageTable::load()
{
    current_page_table = this;

    // Writing page directory address into CR3 register
    write_cr3((unsigned long)(current_page_table->page_directory));

    Console::puts("Loaded page table\n");
}
```

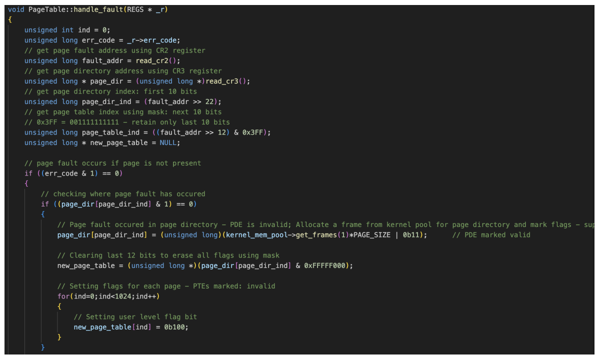
page_table.C: enable_paging : This function is responsible for activating paging on the CPU. It accomplishes this by setting the 32nd bit to 1 and then writing this modified value into register CR0. Additionally, it updates the *paging_enabled* variable to indicate that paging has been enabled with a value of 1.



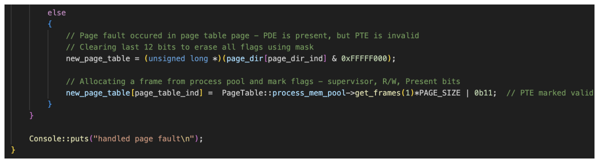
```
void PageTable::enable_paging()
{
    // set paging bit
    write_cr0(read_cr0() | 0x80000000);
    // set paging enabled var
    paging_enabled = 1;

    Console::puts("Enabled paging\n");
}
```

page_table.C: handle_fault : This function is employed to manage page-fault exceptions on the CPU. When a page fault occurs, this method determines the appropriate action to take by consulting the page table. If the page does not currently have a physical memory frame associated with it, the method allocates an available frame and updates the page table entry accordingly. In cases where a new page table needs to be initialized, the method allocates a new frame for it and updates both the new page table page and the directory accordingly. This method follows a structured algorithm(described below) to efficiently handle page faults and ensure proper memory management.

First, we obtain the address that caused the page fault by reading register CR2, followed by retrieving the page directory address from register CR3. Then, we extract the page directory index (first 10 bits) and the page table index (next 10 bits) from the page fault address. We check the error code of Exception 14, verifying whether bit 0 is unset to determine if a *page not present* fault occurred. If so, we examine whether the fault is within the page directory or the page table by checking the *Present* field of the page directory entry. If the fault is in the page directory, we allocate a free frame from the kernel pool for the page directory entry and mark it as *valid.* Subsequently, we initialize the corresponding page table by allocating a free frame from the process pool and marking all entries as *invalid.* If the fault is in the page table, we allocate a free frame from the process pool for the page table entry and mark it as *valid.* After handling the fault, we return from the page fault handler. If a page fault occurred in the page directory previously, another page fault is raised since the corresponding page is not present in the page table. This time, an available free frame is allocated from the process pool, and the page is marked as *valid.* Subsequent page fault exceptions are not raised since the logical address now exists in the paging system.



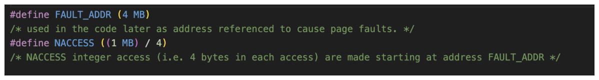
```
void PageTable::handle_fault(REGS * _r)
{
    unsigned int ind = 0;
    unsigned long err_code = _r->err_code;
    // get page fault address using CR2 register
    unsigned long fault_addr = read_cr2();
    // get page directory address using CR3 register
    unsigned long * page_dir = (unsigned long *)read_cr3();
    // get page directory index: first 10 bits
    unsigned long page_dir_ind = (fault_addr >> 22);
    // get page table index using mask: next 10 bits
    // 0x3FF = 001111111111 - retain only last 10 bits
    unsigned long page_table_ind = ((fault_addr >> 12) & 0x3FF);
    unsigned long * new_page_table = NULL;

    // page fault occurs if page is not present
    if ((err_code & 1) == 0)
    {
        // checking where page fault has occured
        if ((page_dir[page_dir_ind] & 1) == 0)
        {
            // Page fault occured in page directory - PDE is invalid; Allocate a frame from kernel pool for page directory and mark flags - sup
            page_dir[page_dir_ind] = (unsigned long)(kernel_mem_pool->get_frames(1)*PAGE_SIZE | 0b11);      // PDE marked valid

            // Clearing last 12 bits to erase all flags using mask
            new_page_table = (unsigned long *)(page_dir[page_dir_ind] & 0xFFFFF000);

            // Setting flags for each page - PTEs marked: invalid
            for(ind=0;ind<1024;ind++)
            {
                // Setting user level flag bit
                new_page_table[ind] = 0b100;
            }
        }
```



```
        else
        {
            // Page fault occured in page table page - PDE is present, but PTE is invalid
            // Clearing last 12 bits to erase all flags using mask
            new_page_table = (unsigned long *)(page_dir[page_dir_ind] & 0xFFFFF000);

            // Allocating a frame from process pool and mark flags - supervisor, R/W, Present bits
            new_page_table[page_table_ind] =  PageTable::process_mem_pool->get_frames(1)*PAGE_SIZE | 0b11;  // PTE marked valid
        }
    }

    Console::puts("handled page fault\n");
}
```

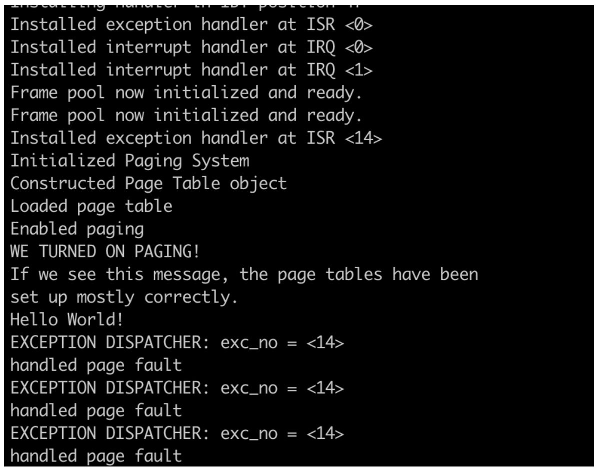
Testing

A few testcases were carried out by making modifications to the *kernel.C* file in the part of the code given below. This code block was changed based on the different test cases.

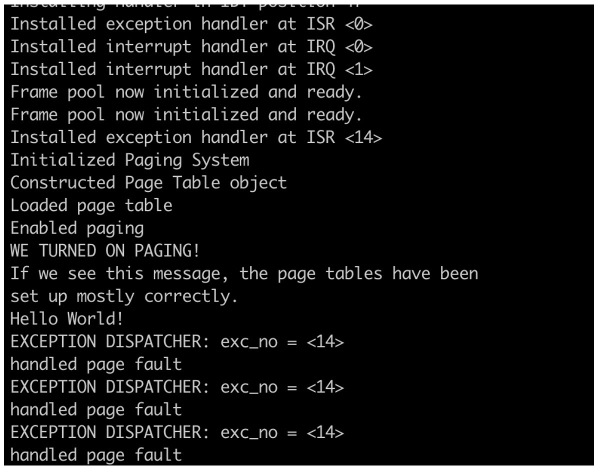


```
#define FAULT_ADDR (4 MB)
/* used in the code later as address referenced to cause page faults. */
#define NACCESS ((1 MB) / 4)
/* NACCESS integer access (i.e. 4 bytes in each access) are made starting at address FAULT_ADDR */
```

- Fault Address: 4MB, Requested Memory: 1MB

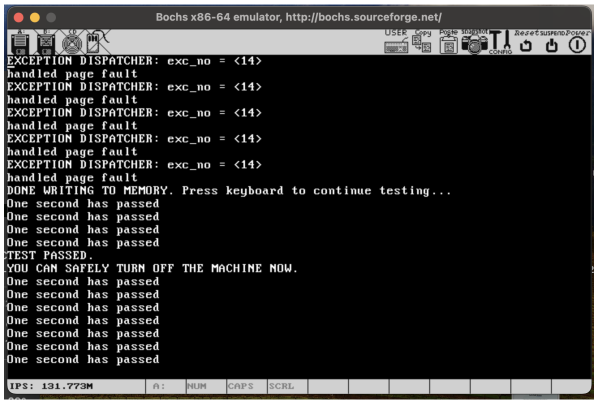


```
Installed exception handler at ISR <0>
Installed interrupt handler at IRQ <0>
Installed interrupt handler at IRQ <1>
Frame pool now initialized and ready.
Frame pool now initialized and ready.
Installed exception handler at ISR <14>
Initialized Paging System
Constructed Page Table object
Loaded page table
Enabled paging
WE TURNED ON PAGING!
If we see this message, the page tables have been
set up mostly correctly.
Hello World!
EXCEPTION DISPATCHER: exc_no = <14>
handled page fault
EXCEPTION DISPATCHER: exc_no = <14>
handled page fault
EXCEPTION DISPATCHER: exc_no = <14>
handled page fault
```



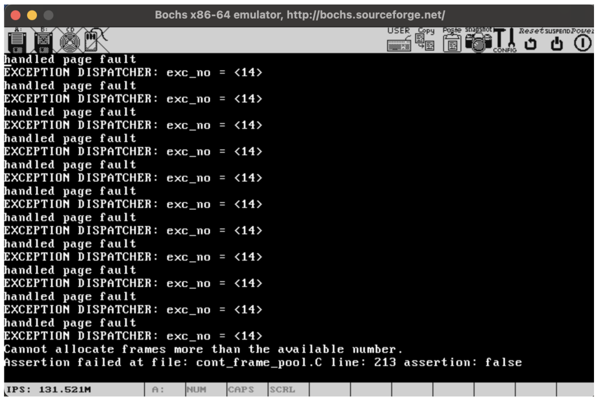
```
Installed exception handler at ISR <0>
Installed interrupt handler at IRQ <0>
Installed interrupt handler at IRQ <1>
Frame pool now initialized and ready.
Frame pool now initialized and ready.
Installed exception handler at ISR <14>
Initialized Paging System
Constructed Page Table object
Loaded page table
Enabled paging
WE TURNED ON PAGING!
If we see this message, the page tables have been
set up mostly correctly.
Hello World!
EXCEPTION DISPATCHER: exc_no = <14>
handled page fault
EXCEPTION DISPATCHER: exc_no = <14>
handled page fault
EXCEPTION DISPATCHER: exc_no = <14>
handled page fault
```

- Fault Address: 4MB, Requested Memory: 12MB



```
EXCEPTION DISPATCHER: exc_no = <14>
handled page fault
EXCEPTION DISPATCHER: exc_no = <14>
handled page fault
EXCEPTION DISPATCHER: exc_no = <14>
handled page fault
EXCEPTION DISPATCHER: exc_no = <14>
handled page fault
EXCEPTION DISPATCHER: exc_no = <14>
handled page fault
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
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
One second has passed
One second has passed
One second has passed
```

- Fault Address: 4MB, Requested Memory: 32MB



```
handled page fault
EXCEPTION DISPATCHER: exc_no = <14>
handled page fault
EXCEPTION DISPATCHER: exc_no = <14>
handled page fault
EXCEPTION DISPATCHER: exc_no = <14>
handled page fault
EXCEPTION DISPATCHER: exc_no = <14>
handled page fault
EXCEPTION DISPATCHER: exc_no = <14>
handled page fault
EXCEPTION DISPATCHER: exc_no = <14>
handled page fault
EXCEPTION DISPATCHER: exc_no = <14>
handled page fault
EXCEPTION DISPATCHER: exc_no = <14>
handled page fault
EXCEPTION DISPATCHER: exc_no = <14>
handled page fault
EXCEPTION DISPATCHER: exc_no = <14>
handled page fault
EXCEPTION DISPATCHER: exc_no = <14>
Cannot allocate frames more than the available number.
Assertion failed at file: cont_frame_pool.C line: 213 assertion: false
```