MP3: Page Table Management

Naveen Babu Krishnasamy Jeyakumar
UIN: 335007146
CSCE611: Operating System

Assigned Tasks:

Main: Page Table Management - Completed.
Files Modified: page_table.H page_table.C cont_frame_pool.H and cont_frame_pool.C

System Design:

The objective of this machine problem is to set up and initialize the paging system and page table infrastructure on the x86 architecture for a single address space. The x86 architecture employs a two-level hierarchical paging scheme. A 32-bit logical address is divided into a 10-bit page table number, a 10-bit page number, and a 12-bit offset. The page table base register points to the start of the page directory. The page table number indexes into the page directory to locate the page directory entry, which contains a pointer to the start of the page table page associated with the page table number. The page number indexes into the page table page to find the page table entry. Finally, the offset is combined with the frame number stored in the page table entry to produce the physical address. This process maps the logical address to a physical address. The memory layout of the machine is as follows:

Total memory in the machine: 32 MB
Memory reserved for Kernel: 4 MB (Direct-mapped to physical memory)
Process memory pool: 28 MB (Freely-mapped to physical memory)
Frame size: 4 KB
Inaccessible memory region: 15 MB – 16 MB

Page faults occur when physical memory cannot be linked to the page directory or page table. In such cases, the frame manager must allocate a new frame, and the corresponding entries in the page directory or page table must be updated.

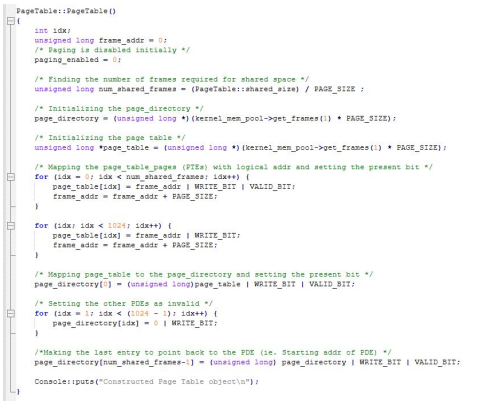
Code Description:

init_paging: Configure the parameters for the paging subsystem.

```
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("Paging System is Initialized\n");
}
```

Figure 1: Init Paging

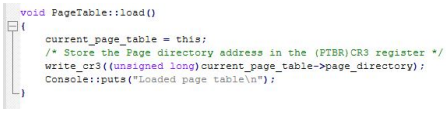
PageTable constructor: Initializes the entries in the page directory and page table for the shared memory segment. The page directory points to each page table in the shared memory and is marked as valid using the last bit of the entry. The page table entries for the shared memory segment (i.e., the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



```
PageTable::PageTable()
{
    int idx;
    unsigned long frame_addr = 0;
    /* Paging is disabled initially */
    paging_enabled = 0;

    /* Finding the number of frames required for shared space */
    unsigned long num_shared_frames = (PageTable::shared_size) / PAGE_SIZE ;

    /* Initializing the page_directory */
    page_directory = (unsigned long *)(kernel_mem_pool->get_frames(1) * PAGE_SIZE);

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

    /* Mapping the page_table_pages (PTEs) with logical addr and setting the present bit */
    for (idx = 0; idx < num_shared_frames; idx++) {
        page_table[idx] = frame_addr | WRITE_BIT | VALID_BIT;
        frame_addr = frame_addr + PAGE_SIZE;
    }

    for (idx; idx < 1024; idx++) {
        page_table[idx] = frame_addr | WRITE_BIT;
        frame_addr = frame_addr + PAGE_SIZE;
    }

    /* Mapping page_table to the page_directory and setting the present bit */
    page_directory[0] = (unsigned long)page_table | WRITE_BIT | VALID_BIT;

    /* Setting the other PDEs as invalid */
    for (idx = 1; idx < ((1024 - 1)); idx++) {
        page_directory[idx] = 0 | WRITE_BIT;
    }

    /*Making the last entry to point back to the PDE (ie. Starting addr of PDE) */
    page_directory[num_shared_frames-1] = (unsigned long) page_directory | WRITE_BIT | VALID_BIT;

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

Figure 2: Page Table Constructor API Code

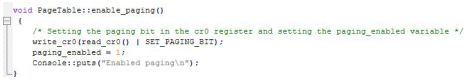
load() function: Loads the page table into memory by storing the address of the page directory into the CR3 register (PTBR).



```
void PageTable::load()
{
    current_page_table = this;
    /* Store the Page directory address in the (PTBR)CR3 register */
    write_cr3((unsigned long)current_page_table->page_directory);
    Console::puts("Loaded page table\n");
}
```

Figure 3: Page Load Function

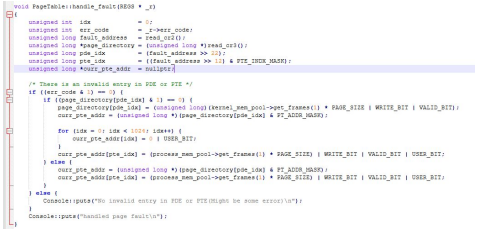
enable_paging() function: Enables the kernel to switch from physical addressing to logical addressing by setting the Most Significant Bit (MSB) in the CR0 register.



```
void PageTable::enable_paging()
{
    /* Setting the paging bit in the cr0 register and setting the paging_enabled variable */
    write_cr0(read_cr0() | SET_PAGING_BIT);
    paging_enabled = 1;
    Console::puts("Enabled paging\n");
}
```

Figure 4: Enable Paging Function

handle_fault(): This method reads the logical address from the CR2 register and the error word from the error code in the REGS object. It then looks up the appropriate entry in the page directory and page table for faulty entries using the valid bit (0th bit). If there is no physical memory (frame) associated with the page, an available frame is allocated, and the page table entry is updated. If a new page table page needs to be initialized, a frame is allocated for it, and the new page table page and directory are updated accordingly.



```
void PageTable::handle_fault(REGS * _r)
{
    unsigned int  idx             = 0;
    unsigned int  err_code        = _r->err_code;
    unsigned long fault_address   = read_cr2();
    unsigned long *page_directory = (unsigned long *)read_cr3();
    unsigned long pde_idx         = (fault_address >> 22);
    unsigned long pte_idx         = ((fault_address >> 12) & PTE_INDX_MASK);
    unsigned long *curr_pte_addr  = nullptr;

    /* There is an invalid entry in PDE or PTE */
    if ((err_code & 1) == 0) {
        if ((page_directory[pde_idx] & 1) == 0) {
            page_directory[pde_idx] = (unsigned long)(kernel_mem_pool->get_frames(1) * PAGE_SIZE | WRITE_BIT | VALID_BIT);
            curr_pte_addr = (unsigned long *)(page_directory[pde_idx] & PT_ADDR_MASK);

            for (idx = 0; idx < 1024; idx++) {
                curr_pte_addr[idx] = 0 | USER_BIT;
            }
            curr_pte_addr[pte_idx] = (process_mem_pool->get_frames(1) * PAGE_SIZE) | WRITE_BIT | VALID_BIT | USER_BIT;
        } else {
            curr_pte_addr = (unsigned long *)(page_directory[pde_idx] & PT_ADDR_MASK);
            curr_pte_addr[pte_idx] = (process_mem_pool->get_frames(1) * PAGE_SIZE) | WRITE_BIT | VALID_BIT | USER_BIT;
        }
    } else {
        Console::puts("No invalid entry in PDE or PTE(Might be some error)\n");
    }
    Console::puts("handled page fault\n");
}
```

Figure 5: Page Fault Handling

Frame Manager System Design:

The goal of Machine Problem 2 is to create a frame pool manager that handles the allocation and release of frames for both the kernel frame pool (2MB) and the process frame pool (28MB).

Additionally, it should mark certain frames as inaccessible (15MB), making them off-limits to users.

Contiguous Frame Pool Design: The contiguous frame pool manages frames for the kernel pool between 2MB and 4MB, and frames for the process pool from 4MB to 32MB. It uses a bitmap methodology where each frame is represented by two bits indicating its state (11 - Frame is used, 00 - Frame is free, 01 - Frame is HoS and allocated). Since two bits are used per frame, one byte of the bitmap can store information for 4 frames. Given that the frame size is 4KB, a bitmap can store a total of 16K frames (4KB * 4).

Code Description:

Contiguous Frame Pool Constructor: Initializes the data structures for the contiguous frame pool from the starting base frame number to base frame number + number of frames. It also uses an info frame number to store the state of the frame in a bitmap. If the info frame number is 0, the first base frame number is used as the info frame.

```
ContFramePool::ContFramePool(unsigned long _base_frame_no,
                             unsigned long _nframes,
                             unsigned long _info_frame_no)
{
    assert(_nframes <= FRAME_SIZE * 4);

    int num_bytes;
    base_frame_no = _base_frame_no;
    nframes = _nframes;
    nFreeFrames = _nframes;
    info_frame_no = _info_frame_no;

    /* If _info_frame_no is zero then we keep management info in the first
     * frame, else we use the provided frame to keep management info */
    if(info_frame_no == 0) {
        bitmap = (unsigned char *) (base_frame_no * FRAME_SIZE);
    } else {
        bitmap = (unsigned char *) (base_frame_no + (info_frame_no * FRAME_SIZE));
    }

    assert ((nframes % 4) == 0);

    /* Everything ok. Proceed to mark all frame as free and set bitmap to 0
    which indicates that all the frames are free */
    for(unsigned long fno = 0; fno < _nframes; fno++) {
        set_state(fno, FrameState::Free);
    }

    // Mark the first frame as being used if it is being used
    if(_info_frame_no == 0) {
        set_state(0, FrameState::HoS);
        nFreeFrames--;
    } else {
        set_state(0, FrameState::HoS);
        nFreeFrames--;
        for (int i = 0; i < info_frame_no; i++) {
            set_state(i, FrameState::Used);
            nFreeFrames--;
        }
    }

    if (ContFramePool::frame_pool_head == NULL) {
        ContFramePool::frame_pool_head = this;
        ContFramePool::frame_pool_tail = this;
    } else {
        ContFramePool::frame_pool_tail->frame_pool_next = this;
        ContFramePool::frame_pool_tail = this;
    }
    frame_pool_next = NULL;

    Console::puts("ContframePool::Constructor initialized\n");
}
```

Figure 6: Contiguous frame pool constructor API code

get_frames API: Takes the number of frames requested as an argument and returns the head frame number. This function searches for free frames in the bitmap, marks the first free frame as the head frame, and the rest as used (allocated) frames. It then returns the head frame number. This API allocates only contiguous frames and skips non-contiguous frames even if they are available.

```
unsigned long ContFramePool::get_frames(unsigned int _n_frames)
{
    // Any frames left to allocate?
    if(_n_frames > nFreeFrames) {
        Console::puts("These many free frames are not available");
        Console::puts("nFreeFrames = "); Console::puti(nFreeFrames);Console::puts("\n");
        Console::puts("_nframes = "); Console::puti(_n_frames);Console::puts("\n");
    }

    unsigned int i = 0, count = 0, cont_n_avail = 0;
    while ((i < ContFramePool::nframes) && (count < _n_frames)) {
        if (get_state(i) == FrameState::Free) {
            count++;
        } else {
            count = 0;
        }

        if (count == _n_frames) {
            cont_n_avail = 1;
            break;
        }
        i++;
    }

    if(cont_n_avail == 0) {
        Console::puts("Continuous memory not available\n");
        return 0;
    }

    unsigned long first_free_frame = i - _n_frames + 1;
    for (i = first_free_frame; i < (first_free_frame + _n_frames); i++) {
        if (i == first_free_frame) {
            set_state(i, FrameState::HoS);
        } else {
            set_state(i, FrameState::Used);
        }
        nFreeFrames --;
    }
    return (first_free_frame + base_frame_no);
}
```

Figure 7: get_frames code

release_frames API: Takes the head frame number as input and returns nothing. This function

first checks whether the frame is inside the process pool or kernel pool, marks the head frame as free, and marks the rest of the allocated frames as free until it reaches another head frame or free frame.

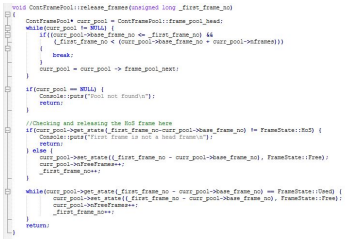


Figure 8: release_frames API code

mark_inaccessible API: Takes the base frame number and number of frames as input and returns nothing. This function marks the frames from the base frame number to base frame number + number of frames as head frames to make them inaccessible.

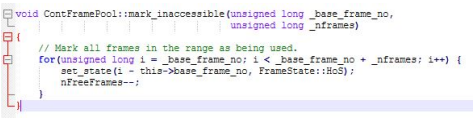


Figure 9: mark_inaccessible API code

needed_info_frames API: Takes the total number of frames as an argument and returns the number of info frames required to store the total frames. In a simple pool, 1 bit is used for addressing the frame, so the total number of frames that can be stored in the bitmap is 32K. Similarly, in a contiguous pool, 2 bits per frame are used, so the total frames that can be stored in the bitmap are 16K. Hence, the total number of info frames is: total frames / 16K + (total frames

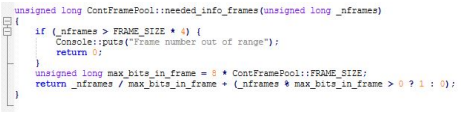


Figure 10: needed_info_frames API code

Testing and Output:

The testing is conducted using Bochs, where the test case invokes page faults and outputs were observed to validate the paging functionality. The test case is tested by modifing NACCESS values in Kernel.c (Values tested = 1, 15, 27, 28) For NACCESS = 28 We will get an error as Requested number of Frames not available. For the remaining values of NACCESS the test case is passed.

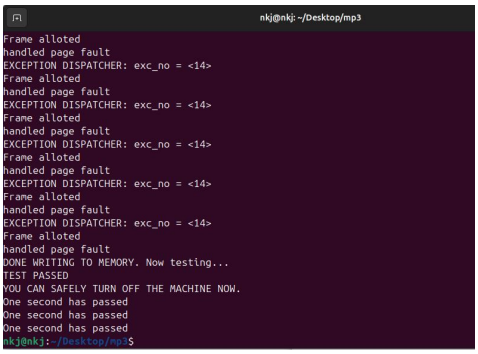


Figure 11: Test Output