MP3: Page Table Management

By: Mehul Varma

Part 1 Introduction

In this assignment, we were tasked to implement a two-level lookup Page Table mechanism that will help us start implementing an on-demand paging-based virtual memory system for our kernel.

The two levels of the paging system we will be implementing will utilize a Page Directory which keeps track of all the Page Table pages, and the Page Table pages themselves that contain the lookups for actual physical frames.

In our system configuration, we will make sure that the first 4MB of every process' logical memory is directly mapped to the same first 4MB of physical memory.

To implement the paging system we implement a Page Table class that takes care of developing a Page Table for each process and takes care of necessary actions like loading, enabling paging and page fault handling.

Part 2 Data Design

We make use of only one instance variable and a few static variables that help in our implementation of the Page Table class. The following are the class variables and data structures we make use of -

Private Variables:

- `static PageTable* current_page_table`
  Pointer to currently loaded page table object
- `static unsigned int paging_enabled`
  Flag to check if paging is enabled
- `static ContFramePool* kernel_mem_pool`
  Pointer to kernel memory pool
- `static ContFramePool* process_mem_pool;`
  Pointer to kernel memory pool
- `static unsigned long shared_size;`
  Size of shared memory among processes. In our case, it is 4 MB.

Part 3 Architectural and component-level design

The majority of implementation for this Machine problem was the implementation of the Page Table Constructor and Page Fault handler.

Page Table Constructor:

The page table constructor is responsible for setting up the shared 4MB in the paging directory and the page table pages that would map that segment of memory. Since we know that the first 4MB is reserved for OS functionality, we make sure to manually map the page table to the physical addresses so the logical memory from 0-4MB maps directly to the physical memory of 0-4MB.

The following points describe the implementation of the constructor.

- 1 frame for Page Directory and 1 frame for 1 Page Table Page gotten from Kernel Memory Pool. 1 Page Table page is enough because each Page Table Page handles 4MB of memory.
- First Page Directory Entry initialized to the first Page Table Page that was just created. Attributes are set so the only first page is present and the rest are not present.
- All Page Table Entries in the first Page Table Page are initialized to present since all the 1024 memory frames are being used. The bits are set to present and read/write.
- For each Page Table Entry, the physical address is incremented started from address 0x00000.
- The instance variable page_directory is set to the pointer pointing to the Page Directory page that was just created.

Page Fault Handler:

The implementation of the fault handler follows the following steps.

- First, we load the virtual address that caused the page fault using the CR2 register.
- Using the virtual address, we break down the 32 bits into directory index, table index, and offset bits that will help us navigate through the Page Directory and Page Table Page.
- Next, we check if the page directory index we extracted is valid. If it isn't valid we have to allocate a page table page and then use that page to allocate a memory frame. We do this by getting a single frame from the Kernel Memory Pool and

initialize a new Page Directory Entry for this newly created Page Table Page. We also initialize the entries in this Page Table Page to be user mode.
- Once we know that the Page Directory Entry exists for the extracted index, we go on to get a frame from the Process Memory Frame Pool which is the new frame we will make to address this page fault.
- Once we get this frame, we use it to create a new Page Table Entry so that the frame number is in the entry along with the created and write bit set. This PTE is set at the page index we read from the virtual address in the start.

In this function, we also utilize some bit manipulation functions that help us extract the different parts of the virtual address to create a new physical memory translation for it.

Part 4 Testing & Modifications

For this assign the modifications were only made to page_table.H and page_table.C files.

Testing:

- For testing the program, the same memory creation and allocation test was ran on a larger address space of 5MB which tested the creation of a Page Directory Entry twice.