CSCE 410 P2 Report

Gustavo Estrela de Matos 925003559

1 Description

In this project we developed a simplified Memory Manager based on the x86 architecture. This manager implements paging and, therefore permits us to create a system in which processes only see virtual addresses instead of physical addresses.

In our system we have 32 MB of memory, 4 MB reserved for the kernel and system. The physical memory is divided in blocks of 4 KB, the frames. The virtual memory is divided in blocks of the same size and they are mapped to physical blocks. To do this mapping we need two structures: page directory and page table.

To develop this manager we implemented two classes. The first one is the class FramePool and its responsibility is to control a contiguous set of frames. The second one is the class PageTable, responsible for creating and managing pages and frames correspondences.

3 FramePool Class

This class is responsible to manage contiguous sets of frames, a “pool” of physical memory blocks.

3.1 FramePool Attributes

- *base_n* stores the number of the first frame the pool manages
- *frames_n* stores the number of frames the pool manages
- *info_frame_n* stores the number of the frame that stores information about this frame pool
- *free_frames* is a vector of unsigned ints responsible for mapping every frame in the pool to determine if it is available or not. To access information of the $i-th$ frame we should simply access the $i \bmod \text{LONG\_SIZE}$ index of $\text{free\_frames}[i/\text{LONG\_SIZE}]$ where LONG_SIZE depends on the system and is defined at *frame_pool.H* file.
- *previous_frame* stores the last FramePool object created before the creation of “this” object
- *head_frame* is a static variable that stores the last FramePool object addresses

3.2 FramePool Methods

- *FramePool* is the default constructor
- *get_frame* returns the number of a free frame inside the frame pool. This is the method responsible to give physical addresses to the PageTable.
- *mark_inaccessible* makes a subset of the framepool frames inaccessible. At first our implementation just set these frames as unavailable at the *free_frames* bitmap, which is not correct because any user could release this frames and work with them normally.
- *release_frame* is a static method that “free” frames, this is, set a frame as available at the bitmap variable.

3.3 FramePool Information Storage

A FramePool object in our system is created at the stack and because of that, since the stack has a limited size, its recommended to store all the FramePool information inside

frames.

In our implementation we decided that storing only the *free_frames* inside the frame and the other variables in the stack was enough. Changing this implementation should not be hard if necessary.

3.4 How to Release a Frame

To release a frame you should first determine which frame pool object is responsible for that frame. To do this we created a simple linked list of frame pools objects using the attributes *previous_frame* (which should have been named previous_frame_pool) and *head_frame* (same name problem).

4 PageTable Class

This class is responsible for creating and managing associations between virtual and physical addresses, in other words, links pages to frames.

4.1 PageTable Attributes

- *current_page_table* is a static variable that stores the address of the current page directory being used
- *paging_enabled* is a static boolean that says if paging was enabled
- *kernel_mem_pool* is a static pointer to the Kernel FramePool object
- *process_mem_pool* is a static pointer to the Process FramePool object
- *shared_size* determines the size of the memory that has direct mapping from virtual do physical, also static
- *page_directory* stores the address of the page directory

4.2 PageTable Methods

- *init_paging* is a simple static method to initialize static members
- *PageTable* is the constructor of the class. This method requests frames from the kernel frame pool to store the page directory and its first page table. After that the page table have its entries direct mapped to physical addresses. The other entries of the page directory are marked as not present.
- *load* is a method that loads a page directory to map virtual addresses to physical addresses. To make this change, the method updates the *current_page_table* variable and loads to the CR3 register the address of the page directory. The CR3 register stores the page directory that must be used to map addresses.
- *handle_fault* is responsible for deciding what the system should do whenever a not mapped (or not accessible) virtual address is referenced.

4.3 How to handle with Page Faults

In our memory manager, whenever a page fault happens we first see if the address is already mapped, and if it is we are in the case that a protected page was accessed by someone who has not the proper rights to do it, then nothing more than showing a warning message is needed (for now).

If the address referenced is still not mapped, the fault handler should create a new association. If the page table referenced is still not present, we should create a new page table and initialize it with not present pages. Then, we should ask for a new frame and associate it to the virtual address that caused the fault.

4.3 How to Determine the Virtual Addresss Referenced

To determine the virtual address that was referenced in a page fault we can read the CR2 register, which has the following architecture:

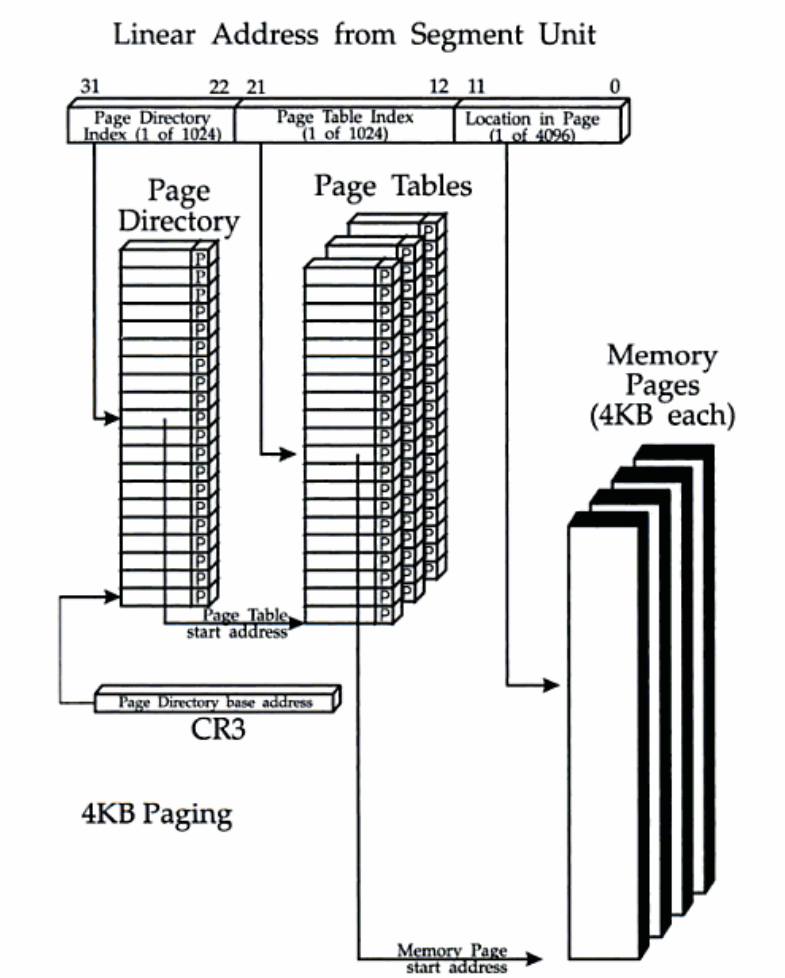

Illustration 1: Fig1 - The structure of the CR2 register is the same as seen in the top 32 bits. Removed from the book in the referrences

5 Results

This project was developed with use of a version control program, git, and the working log can be viewed at the GitHub portal through this link. As it is possible to see at the log, most of the time of the development was bug finding, which is can be very difficult but makes you really think about what you are developing.

6 References

Shanley, Tom. *Protected mode software architecture*. Reading, Mass: Addison-Wesley, 1996