Introduction

In this machine problem 2, we have implemented a paging mechanism on the x86 architecture by setting up the paging system and the page table infrastructure for a single address space. The section that follow describes the implementation of the page table, the frame pool, and the page-fault handler.

Page Table implementation

The page_table.c implements the functionality defined in the page_table.H to initialize and load the page table and to enable paging. In the page table constructor, we set up the directory by allocating first free frame (at 2MB location) via a *get_frame()* call. Next we set up the page table by allocating the next free frame (at 2MB location) to directory via another *get_frame()* call.

To directly map the first 4MB of physical memory (1024 frames) to the virtual address, we make the first 1024 entries in the page table as present by setting the LSB bit of the address to 1(address | 0x03). We then put the page table address into the page directory entry zero and mark it as present. We also mark the next 1023 page directory entries as not present by clearing the LSB bit.

In the *PageTable_load()* functions we assign the current object to *current_page_table*. In the enable_paging() function we write the page directory address to register CR3 and set the paging bit(MSB bit) in CR0 to 1 which enable paging.

Page Fault handler

In the handle_fault (REGS * _r) function, we read the 32-bit address that caused the page fault from CR2 register. We also read the lower 3 bits of error code in which bit0 indicates whether the page is present or not. We extract the page directory index (MSB 10 bits) from the address and page table index from the next 10 bits.

We check whether the page directory entry is zero (empty). If empty, we get 1024 frames from the frame pool and mark that page entries as present/valid. The base address of the page table is then stored in the page directory entry and marked as valid. If the page directory entry is present and is valid, we index the page table entry and if the page entry is not present, we get a free frame from the frame pool and put the frame address into page table entry and mark that as valid.

Frame pool implementation

In the frame pool constructor, we have defined the bit mapping registers and initialized all the 256 registers to zero. The 256 register (256*32 = 8192 bits) maps the 8192 frames (32MB address space). Mark_Frames() API has been used which marks the corresponding frame as “USED” when allocated and “AVAILABLE” when freed. One advantage of the bitmap_register implementation is its space efficiency, and its simplicity. Each page only needs one bit of control information: either the page is allocated (USED), or it isn't (AVAILABLE).

The get_frame() function scans the bitmap_regsiter() to find a free_frame (first bit zero). If found, it allocates the free frame and marks the frame (page) as used. If none of the frames are available, this API returns a zero indicating “out of memory”.

The mark_inaccessible() API marks the marks the area of physical memory as inaccessible. In the current implementation, we have made the region between 15MB and 16MB as non-available by using this API.

Changes made to the header files:

Frame_pool.H changes:

MARK_USED and MARK_AVAILABLE have been declared as macro with values 1 and 0 respectively.
Bitmap_regsiter has been declared as a static unsigned long 256 byte array.
Mark_Frames() function prototype is added. void Mark_Frames(unsigned long frame_no, int flag);