Machine Problem 3
Pranami Bhattacharya
UIN: 223004200

In this problem, we have implemented virtual memory management and memory allocation.

Frame Manager and Frame Pool:
This is taken care of in the files "frame_pool.C and frame_pool.H"
A bitmap has been made use of to maintain the availability information of the frames. If the base frame number is 512 it is a kernel memory pool and the bitmap is assigned a start address of 2MB. If the base frame number is 1024, it is a process memory pool and the bitmap is assigned a start address by computing its offset using the parameter “info_frame_no”.

The following data members have been defined inside the class FramePool:

frame_bitmap: The bitmap that maintains the frame pools
nframes: The number of frames in the frame pool, 512 for the kernel pool and 7168 for the process pool
info_frame_no: Frame number of the frame in which this frame pool's bitmap is stored
base_frame_no: 512 for the kernel pool and 1024 for the process pool

When a request is made for a frame, we scan through the frame bitmap and assign the first frame whose corresponding bit is 0. After allocating this frame, we assign a value of 1 to this bit.
When a frame is released, the corresponding bit in the relevant bitmap is made 0.
A frame is marked inaccessible by setting the corresponding bit to -1.

Page Table Directory and Page Table:
This is taken care of in the files "page_table.C" and "page_table.H"
The page directory is set up on a 4KB aligned boundary. Since the memory space between 0 - 2MB is reserved by the kernel , hence we take the next 4 KB aligned address above 2 MB and assign it to the page directory.The next 4 KB aligned address is assigned to the page table. The page directory and page table are allocated from the kernel memory pool.
The bytes 0 - 4MB in the page table are set up for direct mapping. We mark the entries in this page table as valid. The corresponding bit in the page directory is also marked as present.
We assume that all the other page directories are invalid and when a new page table is created, we update the entry in the page directory by pointing it to the newly created page table.

The recursive trick has been made use of for implementing the memory mapping. We make the last entry in the page directory to point back to the first entry in the page directory and validate the respective flags by masking with 3.
Since each page table covers 4MB of the address space, the last page directory entry will point to a page table that covers the top 4MB of the address space – that’s 0xFFC00000 and above. Now, since we have the page directory acting as a page table, when we write to the address 0xFFC00000, the processor first finds the page table that corresponds to that address which is our

page directory. It then looks for the first entry in the supposed “page table” when it is actually looking at the first entry in the page directory. Conveniently, the top 4KB of the address space is mapped to the page directory. This way, by accessing memory at offsets from 0xFFFFF000, we modify the PDEs in the page directory.

Page Fault Handling:
When a page fault occurs (exception 14), we find the error code corresponding to the page fault. The faulting address is written to the control register CR2 and using this faulting address we can reach the corresponding page table directory and the page table. This is done by masking and shifting operations on the fault address. The first 10 bits will give the index of the page directory and the next 10 bits will give the index to the page table.
The page directory obtained through the index explained above is checked for validity.
If valid, it means that the page table containing the faulting address has already been created. We allocate a frame from the process frame pool and set the entry in the page table as "present".
If not valid, we need to create a page table. This is done by allocating a frame from the kernel frame pool for the page table and a frame from the process frame pool. We make the page directory point to this newly created page table and also set the relevant fields. We have made use of the recursive trick as explained above.

Virtual Memory Pool
The virtual memory pool will allocate requested size of pages.
The VM pool is maintained as a structure and we have a pointer to the structure for all the allocation and deallocation requests. Any allocation is always taken as multiple of page size.
We assign a frame to the virtual memory bitmap.
For each allocation, we keep a count of the number of regions allocated inside the virtual memory pool. If the count is 0, then the start address of the region is allocated else the start address is the end address of the previous allocated region. After allocating, we set the valid bit to 1 implying that the region is taken. The start address of the region is returned.
For each de-allocation, we find the corresponding index to the region in the VM pool that has a start address match. The entire size in that region is deallocated by calling free_page. The respective region is marked as 0. After every de-allocation, we call the load function that will flush the TLB so that we do not end up with any wrong address translations.

The is_legitimate function simply checks if the address requested falls within the bounds of the region, returns false if not.

Files written:
page_table.C
frame_pool.C
vm_pool.C

Machine Problem 3
Pranami Bhattacharya
UIN: 223004200

Files changed:
page_table.H
vm_pool.H
frame_pool.H

page_table.H

Added headers “console.H” and paging_low.H. The class “Console” in the header “console.H” encapsulates the output operations to the console screen and the header “paging_low.H” is needed for the low level register operations for the X86 paging subsystem. We have made use of the control registers in our C file.

Added “*page_table” ,“address” and “vm_pools” members to the class PageTable. These are relevant members which would be needed in the file page_table.C for initializing the paging subsystem. The number of virtual memory pools will be needed for registering the VM Pool with the Page Table.

Modified page size as 4KB and entries per page as 1024. In our design, we maintain a page size of 4KB and accordingly 12 bits would be needed to address such a page (offset bits), Of the 32 bits, the MSB 10 bits would be needed to index the page directory and the next 10 bits to index the page table.

frame_pool.H:

Declared the members “nframes”, “info_frame_no” and “base_frame_no”. Their utility has already been explained in the first section.

Defined an array of bitmaps “_frame_bitmap” for the allocation and de-allocation of frames.

Declared a function “get_frame_address”

vm_pool.H

Declared a structure VMPool with parameters like size, valid and start_address.

Declared members like count and capacity for the pool. A pointer to the structure VMPool has also been declared

Machine Problem 3
Pranami Bhattacharya
UIN: 223004200

Declared private members “base_address”, “size”, “*frame_pool” and “*page_table”