P3 DESIGN

Operating Systems – CSCE 613

ABSTRACT

Design for Paging Structure and Memory Management and API discussion

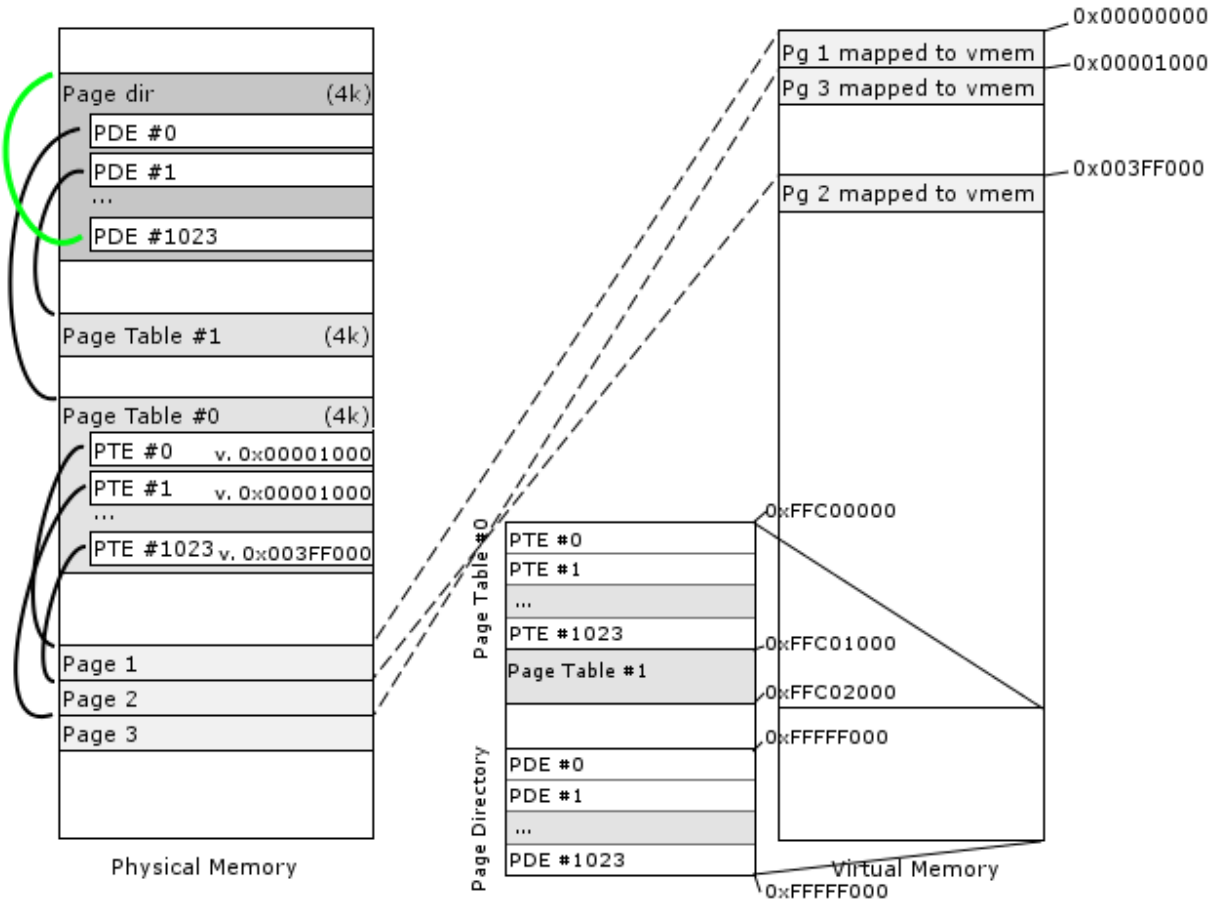
Tharun Battula

82400197

DESIGN

- The specifications mentioned in the assignement are followed
    - 32 MB of memory layout
    - Kernel Pool 2MB to 4MB
    - Process Pool beyond 4MB
    - Inaccessible region 15MB to 16 MB
- The P2 assignment structure with frame pool class is very useful in getting frame spotlessly. I followed frame allocation as below.
    - Kernel Memory Pool
        - Used for allocation of Page Frames
        - Used for first PDE - First Page Table - for direct mapping of 4MB (kernel Memory) – This will be used directly even if paging is enabled
    - Process Memory Pool
        - Used get_frame() for allocation of first level and actual data tables based on memory referencing and whenever page fault occurred after paging enabled is called.
- Recursive Page table lookup
    - After the paging is enabled, it becomes difficult to access a page table with physical address. The get_frame() function gives page table location in terms of physical address space. For process pool obtained frames, Its corresponding virtual mapping might not be have been mapped in the page directory lookup.
    - The Idea to mitigate the above challenge is using recursive page table, where when we create Page directory we point the page directory last entry to page directory itself.
    - This way for the virtual addresses above 0xFFC0 0000 looked up last entry of PDE .i.e., Page directory itself recursively. Hardware will map PDE last entry to PDE as page table. Now 0th entry of PDE(i.e.) page table maps to physical page table. Similarly if we know directory index of any page table we can access the contents of page table by virtual address by looking up (0xFFC0 0000 +(dir_index<<12))

- For more understanding on details refer to this below diagram and reference



Reference: http://www.rohitab.com/discuss/topic/31139-tutorial-paging-memory-mapping-with-a-recursive-page-directory/

- Protection Levels
  - Kernel Memory Pool Frames – (e.g., Page Directory, PDE#0-> first page table)
    - Supervisor- bit2 - (1)
    - Read/Write – bit1-(0)
    - Present- bit0 - (1) if created otherwise non present(0)
  - Process Memory Pool Frames-
    - User level- bit2 - (0)
    - Read/Write- bit1 - (0)
    - Present- bit 0 - (1) if created otherwise non present(0)

Functionality

The above mentioned design with assignment mentioned guidelines is coded.

- Corner Cases
  - If get_frame() requested return 0
    - Prints on Console that no new frames available, physical memory is fully accessed
  - If protection fault occurs
    - Prints the Console type of protection fault that occurred
- The regular expected API functionality as per specification are implemented
  - FramePool class is implemented with the necessary functions.
  - To debug the code → page_table.c file make debug enable flag to 1.
    `#define DEBUG_ENABLE 1`
- Not Modified Page_Table.H

The main recursive page table implementation is used page fault handler function

P2 – New Work

Improved the frame pool class functionality by handling corner cases as per Grader's feedback. I made sure that my new frame pool passes all the test cases in Kernel.C for P2.

1. Handled corner case for not releasing the kernel memory pools info frame
   i.e., first frame of the entire data

2. Mark Inaccessible function Added boundary check if data is accessed from one pool to another pools or beyond the region pool (Illegal access of frame b
3. Fixed the corner case where boundary of get_frame function after allocating every frame should not allocate anymore frames. By changing inequality and fixing an iteration

Assumptions

- First kernel FramePool will be called, It keeps its bitmap info in the first frame residing inside the FramePool
- Second Process FramePool will be called.
- PageTable::init_paging will be called before any page table functions.
- Page Table constructor will be called, then load, and enabled_paging will be called in the same order

Testing

- Tested basic features asked in the assignment by individually checking page table values and its assignment levels
- Tested Recursive page table with prints
- Stress testing by creating one more object for Page Table and allocation of virtual memory for it
- Increased NACCESS value to more than 32 MB and such and tested
    - For very high memory references I am facing the print of No frames available in the pool.

Efficiency

Time Profiling

Profiled with the timer function available in Kernel.C

Average time displayed in Seconds = 0

Average Ticks = 65

Memory Analysis

1Page Directory Entry → Page Table → 1024 PT.Entries → 1024 * 4KB → 4MB
e.g., first direct mapping 4MB is mapped like this
We have 1024 PDE→ 1024 * 4MB = 4GB of virtual memory can be accessed in theory.
Since the last entry of PDE is used for accessing page tables physical address as part of recursive page tables. (4GB – 8MB) can be virtual memory.

Hence our systems 32MB physical memory will be able to be mapped by 8 PDE in virtual paging.

- Total available frames from our P2 discussion
    - In Kernel pool in 2MB to 4MB → 2MB = 512 frames
        - First two frames are maintenance info for pool structures
        - Hence 510 frames in kernel pool
    - In process pool 4MB to 32 MB → 28 MB
        - Deduct the inaccessible region 1MB
        - 27 MB → 6912 frames

- In P3
    - PDirectory – 1kernel frame
    - PDE#0 –Direct mapping – 1 kernel frame
    - In process pool
        - 1022 frames are needed for maintenance(page tables)
        - 5890 frames for storing the data = 23.5 MB

References:

1. Wikipedia
2. osdev.net – tutorials and wiki pages
3. http://www.rohitab.com/discuss/topic/31139-tutorial-paging-memory-mapping-with-a-recursive-page-directory/