MP4 Design Document

CSCE 611 – Suma Garuda

UIN: 926009146

1. The implementation of MP4 started with:
    - Correction of MP2 – for release_frames() in MP4
    - Correction of MP3 – I could rectify my error through the TA's suggestion on ecampus, I trimmed the 12 bits of page_table addr and the page fault exception was handled. I was using SimpleFramePool() before for get_frame() and we hadn't used release_frames(). But now replaced with ContFramePool() to use get_frames() and release_frames().

2. Part I – Support for Large Address Spaces

    Firstly, we get our page table and page directory entries from the process_mem_pool and not from kernel_mem_pool. We then need to define our entries so as to support recursive page table lookup. We set the 0^{th} entry of the page directory to point to the page table and the 1023^{rd} entry to point to the page directory itself. We also set the entry bits to 0111 instead of the earlier 011 and we pass the first test within kernel.C

3. Part II – Preparing class PageTable to handle Virtual Memory Pools

    - PageTable::register_pool (VMPool * _pool) – maintains a list of registered pools
      Here we use a fixed size array of max. size 200, the no of pools can be kept track of through vmpool_no (during allocation and deletion).

    - VMPool::is_legitimate – here we add support for region check in page fault handler.
      If the pagefault_address is legitimate, we get a frame pool for the fault, else we go into an infinite loop. This function is implemented in vmpool.C.

    - PageTable::free_page(unsigned long _page_no) – The implementation has the same semantics as before, however the page no is now 20 bits. We check if the page is valid, if it is we call the release_frames() function in ContFramePool.C implemented in MP2. We then mark the following page table entry as invalid and reload the CR3 register again so as to flush the TLB of stale entries.

4. Part III – An allocator for virtual memory

- Initializing class VMPool – I also defined block_info in VMPool.H which keeps block_size and availability information of each block. We also define starting address, size and max_address i.e. base_addess + size of each block. The block_info() pool helps us in allocate() and release().

- Unsigned long allocate(unsigned long _size) - First we check if we have any pre-existing blocks, if not we create them. We then have to find a block of suitable size i.e. greater than _size and allocate it. Once allocated, we calculate the address of the occupied region, create a new one after the block ends and change the block_base address. If we do not find a block of size > _size, we return 0 and exit, else we return the base address of the block allocated. Here, we also note that if we allocate _size bytes in a block which is way larger than _size bytes, we have block_size - _size bytes remaining which make a new block, so we define a new_block, make it available and add it to the existing block count.

- Void release(unsigned long _start_address) – This is very similar to allocate as the block to be released is identified by it's start_address. We loop till we find block_address = _start_address, once identified, we mark the block as available, release the block, decrease the block count and merge it with the next one to make one big block.

- Bool is_legitimate (unsigned long _ address) – we check if _address lies within a valid address region (base_address and max_address) , if yes, we return true for page faults, if not, we return false and the function terminates.