Virtual Memory Management

INTRODUCTION:

In this machine problem we extended the page table management to support very large numbers and sizes of address spaces. Earlier MP had both page directory and page tables in kernel space which is just 4MB. Now We will play with virtual memory space. (i.e. where process_pool is created)

IMPLEMENTATION OVERVIEW:

- We implemented virtual memory allocator and extended the page table manager to support virtual memory allocator.
- In the previous MP, since both page directory and page table are in direct mapped memory, it was easy to change the contents of these. Now that these are moved to mapped memory, virtual address is no more same as physical address.
- So, we are following this technique of recursive page table look up.
- CPU always issues virtual address, MMU always reads first 10 bits to be the index in page directory and next 10 bits to be index in page table that is being pointed by the PDE.
- In recursive paging, last entry of PDE will point to the PDE itself. This will enable us to edit the entries of PD and PT. | 1023 : 10 | 1023 : 10 | offset : 12 | This addressing help us edit the entries of Page directory. | 1023 : 10 | X : 10 | Y : 10 | 0 : 2 | This kind of addressing helps edit the page table.
-

CODE EXPLAINED:

1) In *page_table constructor*, both PD and PT are created in process pool, 1023 entry of PD is pointed to PD, list_vm_pools are initialized to NULL.
2) In *fault_handler,* we read the address from cr2 that is creating the page fault. We check if this address is a part of any of the allocated regions(This is done by calling is_legitimate function from vm_pool class). If it's is, we allot a page/frame. If not, program is aborted here. We also change the way page directory and page table are accessed. We follow the above recursive way to do it.
3) In order to extend support to virtual memory allocation, page table class is modified. 2 new functions are included: register_pool, free_page.
4) *Register_pool* makes the entry of the pointer to virtual memory pool whenever a new one is created. So for this a static array, list_vm_pools is created. This array is used by other functions of page table object as well , hence static . Register_pool function is called in the constructor of vm_pool and 'this' pointer is passed on to this function.
5) *Free_page*: we get the address from which we have to access the page table index to get the frame number that is to be released.
6) In *vm_pool class*, we have constructor, allocate, release and is_legitimate functions. Apart from these, I defined a struct to store the base address and size of each region that is allocated. I also created an array to store the references to these struct's.
7) In *vm_pool constructor,* we are initializing the variables and also allocating a frame from the process_pool to store the array that has the information regarding the registered pools.
8) In *allocate* function, we are checking if the number of registered pools is 0 , which means this is the first pool and its start address will be the base address of the pool. If

it’s not 0, we find the end address of previously allocated region from the start address and size which are stored in the array. After we update variables in the struc of allocated regions, we increment the index of this array. (which is the number of allocated regions).

9) In *release* function, region start address is given that is supposed to be released. So, we find the region’s size by accessing the array of the regions. We now calculate how many frames have been allocated in this size and release start of every frame by calling free_frame function in pagetable.