Cameron Bourque      UIN: 826004886

**Design:**

**Page Table:**

For the page table class, I added a VMPool list which is a static, private variable containing a list of vm pools.

For the page table I implemented the init paging function to first, make sure the kernel mem pool and process mem pool passed in aren't null. Then I set the kernel mem pool, process mem pool and shared size to what was passed into the function. This is the same as in

For the page table constructor I start by allocating the page directory in the process mem pool and then creating a page table in the process mem pool for the first page directory entry. Then I mapped the first 4MB into the page table that was created and set the attributes to supervisor, read and write, and present. Then I filled the rest of the page directory with zeroed addresses and set the attributes for them to supervisor level, read and write, and non present.

For the page table load function I set the current page table to this, the page table which called load, and then wrote the page directory address into CR3.

For the page table enable paging function, I just read CR0, or it with the enable paging bit and write the result back into CR0. Then I set paging_enabled to 1, true.

For handling page faults in the page table, I used the MP3 helper code to clean up my code. I first check if the page is present or not. If it is then I print that a protection fault occurred and kill the kernel. Otherwise I read CR2 to get the fault address and then check if it is a legit address in the list of registered VM pools. If so, then I hold a pointer to the vm pool containing it. Otherwise I kill the kernel. Then I get the page directory at address 0xFFFFF000 and find the directory offset based on masking off the fault address by 0xFFC00000 and right shifting 22 bits. Then I hold the page directory entry in a variable. I then get the page table at address 0xFFC00000 ored with the directory offset left shifted 12 bits. Then I store the page table index by masking off the fault address by 0x3FF000 and right shifting that 12 bits. Next I check if the page directory entry is present. If not then I create a frame for the page table in the process mem pool and store it in the page directory at the given offset. Then I create a new frame. If the directory offset is 0 then I create it in the kernel mem pool. Otherwise I use the vm pool I saved to get its frame pool and get frames from that. Then I store that in the page table at the given offset. Then I mark the attributes based on if it is read only or read/write and also if it was in supervisor mode.

For the page table register pool function, I check if the vm list is null. If so then I save the given vm pool there. Otherwise I iterate over the vm lists until I find one whose next null. Then I set its next to null.

For the page table free page function I get the page directory at address 0xFFFFF000 and find the directory offset based on masking off the fault address by 0xFFC00000 and right shifting 22 bits. I then get the page table at address 0xFFC00000 ored with the directory offset left shifted 12 bits. Then I store the page table index by masking off the fault address by 0x3FF000 and right shifting that 12 bits. Then I calculate the frame number based on the page table entry and release it. I then set the page table entry to invalid with address 0 but retain all its other attributes. Then I flush the TLB by writing CR3 with CR3.

**VM Pool:**

For the vm pool class, I created a struct for regions containing an address and a size. Then I created variables for the base address, size, page table which were private. Then I created public variables for the frame pool, region list and next vm pool. I left all the functions the same.

For the constructor, I make sure the page table and frame pool aren't null and that the base address is outside the first 4 MB. Then I register the vm pool with the page table and set the next variable to null. Then I set all the other variables based on the what was passed in. I create the region list at the base address and set its address to the base address and the size to fill a page. Then I zero out the rest of the list.

For the vm pool allocate function, I loop over the indexes of the region list until I find one with size 0. Then I check to see if its the last available address. If so then I see if I can fit it and save the address if so. Otherwise I make another index variable and find the next address that exists in the list. I then check if the allocation can fit in the bounds of the 2 surrounding addresses. If so then I save the address and break out of the loop. Otherwise I continue looping to find another region to fit it in. If no address is saved then 0 is returned as nothing is allocated. Otherwise, I fill the region list at the index and return the address saved.

For the vm pool release function, I make sure the start address is in the bounds of the base address and the base address plus the size. Then I loop over the region list until I find the region with the matching address. I zero it out and then call the free page function on the page table.

For the vm pool is legitimate function, I check if the address is the base address of the vm pool. If so then I return true. Otherwise, if the address is in the first 4 MB, below the base address, or above the address of base address plus the size then I return false. Otherwise I loop over the region list and if the address is saved in it then I return true, otherwise false.

**Online Participation:**

I posted about the page fault handling problems (@144). I also posted the follow ups to it regarding how it got fixed (@144_f1). I also posted about the page fault happening at the page table entry that was supposed to point to the beginning of the vm pool (@149).