# Design Document
# Virtual Memory Management and Memory Allocation

CSCE 611
Operating Systems
MP4 - Fall 2018
by
Himanshu Gupta

In this machine problem, there were several small tasks that finally gives us a complete virtual memory manager. The first part involves the implementation of the recursive page table lookup functionality for the already implemented page table. This was earlier implemented in the directly mapped memory where the Page table and page directories were stored. We first move the allocation of page tables from kernel memory to process memory and see that the test for virtual memory references will fail.

The recursive page table lookup was implemented to map the page tables in the kernel memory. For this we update the handle fault api to access the memory in a different way. The logic is to recursively loop the address in page table and directory so that it accesses the fault address in correct manner. After implementing this, I tested the memory functionality by the api :

GeneratePageTableMemoryReferences

I also increase the NACCESS parameter to (26 MB / 4KB) which worked perfectly as the page table lookup was implemented in recursive manner.

For the part II of the assignment, we modify the page table class to add some extra api's needed for virtual memory manager.

The register pool api needs to maintain a list of memory pools which have been created till now. This uses a list of VMpool of certain size, defined in the Pagetable class itself. We also maintain the number if VM pools register in another variable.

Next, we add the check for address in the page fault handler. The fault in any address should be within the region of VM pool that we are maintaining. The same is implemented in the page fault api. If this fails, we assert the same and abort.

We also add the free page api in the page table file. This will simply call the release frame api for that particular frame number passed to us in the arguments. It also marks the page table entry as invalid for this page. The flushing of TLB is handled in the next part.

For Part III, we first implement the VM pool class. There are several api's which needed careful implementation for the VM pool class to work correctly. The first one is to write the constructor of the VM pool class. The header file declared the same variables and a structure for maintaining the information about the allocated regions of the vm pool. We create a list for same and maintain a count of allocated regions. The constructor of this class assigns the base address of the pool to this list i.e. we use the first page of the VM pool to store the list of allocate regions.

The next allocate api is simply marking that the region asked for allocation in the allocated list. For the first region, we give it to the page maintaining the list of VM pool.
I also tried giving the memory from kernel frame pool to this list which worked better as we don't have to give this first frame for information management. However, since that involves changing the API signature, I used a frame from the VM pool. Another thing is we allocate always in

multiple of page size to keep the implementation simple enough. There will be internal fragmentation due to same but it can be ignored for this assignment as mentioned in the handout.

For the release api, we determine from which region it was taken by finding the base address. We find the number of pages allocated to the particular region and free all the pages for this region. After releasing the memory region, we update the allocated region list by modifying the list of entries and reducing the allocated count.

The final api is just to find that the address given is within the limits of VM pool, if not we return false.

## TESTING

First, I tested the implementation if the recursive page table lookup in part I which ran successfully. I increased the NACCESS parameter to check the whole 26 MB of memory which ran without any errors.
Next, I tested the VM pool implementation by the test given in kernel.C with code_pool and heap pool.
After this ran successfully, I ran a stress test on the memory manager by increasing the test size of the GenerateVMPoolMemoryReferences api to 500 and 1000. The change is illustrated below.

```
    /* -- CREATE THE VM POOLS. */

    VMPool code_pool(512 MB, 256 MB, &process_mem_pool, &pt1);
    VMPool heap_pool(1 GB, 256 MB, &process_mem_pool, &pt1);

    /* -- NOW THE POOLS HAVE BEEN CREATED. */

    Console::puts("VM Pools successfully created!\n");

    /* -- GENERATE MEMORY REFERENCES TO THE VM POOLS */

    Console::puts("I am starting with an extensive test\n");
    Console::puts("of the VM Pool memory allocator.\n");
    Console::puts("Please be patient...\n");
    Console::puts("Testing the memory allocation on code pool...\n");
    GenerateVMPoolMemoryReferences(&code_pool, 200, 500);
    Console::puts("Testing the memory allocation on heap_pool...\n");
    GenerateVMPoolMemoryReferences(&heap_pool, 500, 1000);

#endif
```

The whole tested ran successfully for about 20 mins. This helped me find some small bugs in my implementation which I fixed later and can be verified from the github repository. This verifies that the implementation is correct and robust.