

# Design Document - Virtual Memory Management and Memory Allocation

**Operating Systems - MP4**

- Rajendra Thottempudi  
UIN : 933004901

## **Design :**

The idea of this machine problem is to complete the memory manager. In order to do this, we rely on the base memory manager that we built in our previous Machine Problems. We do this by adding the support for large address spaces and a simple Virtual Memory Allocator.

### **Support for Large Address spaces :**

To extend the usage for large address spaces, we need to use the Process memory pool for memory allocation to the Page table and Page Directory. As the CPU takes the task of generating the logical addresses randomly, we need to map them to the frames. We use the Recursive page table approach to do this. We set the last entry in the page directory (1023) to point to the page directory itself. An address of the form

**| 1023 : 10 | 1023 : 10 | offset : 12 |**

Is referred by the Memory Management Unit in the following manner : the first 10 bits refer to the page directory index, the next 10 bits are thought as the page table index and the 12 bits are considered as the offset. Also, we can access the page table entry through an address of the form :

**| 1023 : 10 | X : 10 | Y : 10 | 0 : 2 |**

Here, the 10 bits (X) is indexed to the page directory entry and the ten bits (Y) is indexed into the page table entry.

**PageTable :: Page\_table ()** : This is the constructor that we implemented in the earlier MPs. We set the recursive setting of the Page Table here by making the last entry (1023) of the page directory to point to itself.

**PageTable :: handle\_fault()** : This gets triggered when the cpu couldn't find a page or the error code is 14. In this function we first check if the address that we are trying to find is legit or not - this is achieved by passing the address to the is\_legitimate function of the virtual memory allocator. If the address is legitimate then we continue with other steps of this function and if not, we just throw an error here and move out of this function. The next step is to check if the error code is 14 or not and proceed only if the error code is 14. We then check if the directory entry is present or not and allocate a frame from the process pool if there is no such page directory and

also update the page directory entry. In case if we have the page directory already, we then check for the particular page table entry in the page directory and see if we have the page entry or not. If we don't have a page table entry, we allocate a new frame from the process pool and update the page table with the new entry. The page fault handler now returns and the next time when we get the same logical address we don't raise a fault and instead return the newly created entry.

### **Registration of Virtual memory Pools and Legitimacy check of Logical Addresses :**

To implement this new functionality, we created two new methods in the Page table class the details of which are as follows :

**PageTable :: register\_pool()** : In this function, we keep a track of the virtual memory pools in the form of a list in which we store their addresses. Whenever a new pool is created, we add its address to the list of virtual memory pools in this function.

**VMPool :: is\_legitimate()** : Whenever a page fault occurs, this method of virtual memory pool is triggered to validate if the address is a legitimate one or not. This is done by checking if the address that we received is within the range - [ base\_address : base\_address + size of the corresponding pool ]. If it exists within this range, we return true - if not we return false indicating that the address that we have is not valid.

### **Allocation and deallocation of VM Pools :**

In this part, we implement a simple virtual memory manager. This has the ability to allocate and deallocate memory in terms of pages in the virtual memory whenever the user requests for it. Virtual Memory pool has a new data structure to keep a track of the start and end addresses of the memory region of the pool and to keep the size of the memory pool. As with the other designs that we have used, we store the management information of all the regions in the first memory region. The following are the details of the implementation of the methods used :

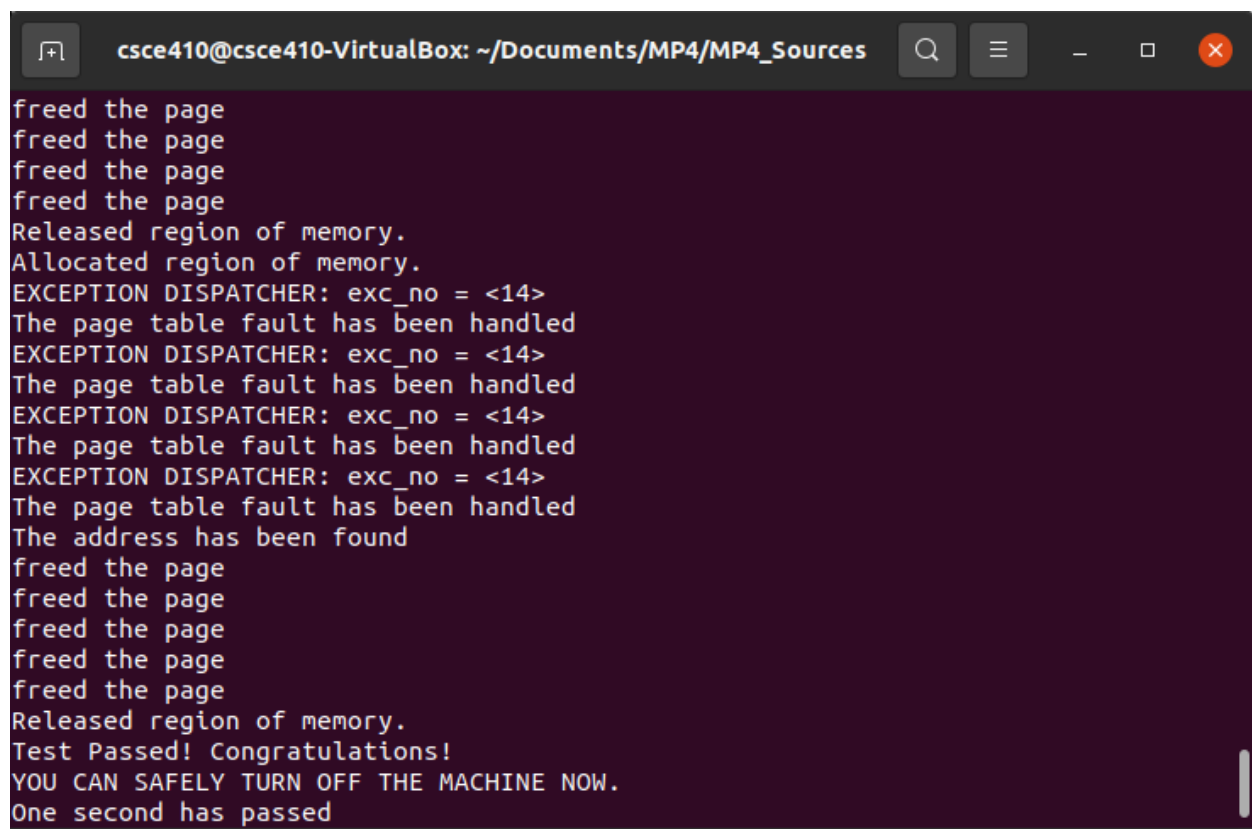
**VMPool :: VMPool()** : This is the constructor method for the virtual memory pool class. It is used to initialize the data structures for the new vm pool object. We use the first region to store the management information and use the base address and the size to determine the base address and the range of the memory block. We mark the base address as the start address of the region and mark all the other regions as invalid/available.

**VMPool :: allocate()** : In this function, we allocate pages to the user based on the requirement. We take the size as the input for this method from the user. Each time a new region is requested, a new element is added to the array which is of class object type containing the base address and the size of the pool region that is being allocated. We then return the base address of the new region that is added in the array of regions for the user.

**VMPool :: release()** : In this function, we deallocate a region whenever a user asks for it. For this, we take the base address and size(i.e. number of pages) of the region to be discarded and call the free\_page function in the pageTable which deallocates the page and frees the frames associated with the particular page. This ensures that the physical memory tied with the virtual memory that is being released is also being released. Later, this function is used to modify the region to make it contiguous.

**PageTable :: free\_page()** : In this function, we first validate the page table entry and once it is validated, we release the frames that are associated with the page and update the page table entry.

Following are the snapshots after the successful implementation of the above functionality :



```
csce410@csce410-VirtualBox: ~/Documents/MP4/MP4_Sources
freed the page
freed the page
freed the page
freed the page
Released region of memory.
Allocated region of memory.
EXCEPTION DISPATCHER: exc_no = <14>
The page table fault has been handled
EXCEPTION DISPATCHER: exc_no = <14>
The page table fault has been handled
EXCEPTION DISPATCHER: exc_no = <14>
The page table fault has been handled
EXCEPTION DISPATCHER: exc_no = <14>
The page table fault has been handled
The address has been found
freed the page
freed the page
freed the page
freed the page
freed the page
Released region of memory.
Test Passed! Congratulations!
YOU CAN SAFELY TURN OFF THE MACHINE NOW.
One second has passed
```

