CSCE6111
MP4
Siru Li
222007338

Design Document

In this machine problem, we implemented the recursive table lookup of our page table manager. This allows us to place the page table and page directory in the shared space, such that they do not necessarily have to be directly mapped. In addition, we implemented the virtual memory pool object and overloaded the C++ "new" and "delete" operator.

1. Page Table

There are two main modifications to the page table manager. Firstly, page tables and page directories are now allocated from process_mem_pool, where logical addresses are not directly mapped to physical addresses. In order to support this, we used the trick called "recursive page table lookup". Before we turn on paging, we need to mark the last entry of the page directory to point to itself. Now, we can locate the page directory through carefully constructed virtual addresses like the following:

```
unsigned long* page_directory = (unsigned long*) 0xFFFFF000; //recursive lookup
```

Fig. 1 Recursive lookup on page directory

Since the MMU always follows the mantra: PDE->PTE, the above address is decomposed as follows: 1023 | 1023 | 0. This way, the MMU will fetch the beginning of the page directory instead. Similarly, we can do the same to access page table entries (PTE):

```
unsigned int page_table_nr = (fault_addr & 0xFFC00000) >> 22; //first 10 bits

…

unsigned long* page_table = (unsigned long*) (0xFFC00000 | (page_table_nr << 12));
```

Fig. 2 Recursive lookup on page directory

This is decomposed as: 1023 | page_table_nr | 0, which gives us the location of the page table that manages the physical frame mapped to this virtual address. The remaining steps are identical as that of the previous assignment.

The second important modification we made was to allow the freeing of pages. To free, we first mark the corresponding page table entry invalid, which is straightforward. However, there is one caveat. Since the TLB is not aware of our changes, we need to "update" the TLB. In order to do that, we simply reload the cr3 register. It is not the most efficient approach, but it is good for now.

Lastly, I created a VMPool* head and VMPool* tail pointer, to make a linked list of VMPool objects. In register_pool() function, I simply added the VMPool object to the linked list. By traversing the list, we can locate the virtual frame pool.

2. Virtual Frame Pool

The implementation of virtual frame pool is quite similar to that of the cont_frame_pool. However, here, we cannot use a bitmap to keep track of allocated regions due to the arbitrarily large virtual memory space. To simplify the assignment, I just created an array of "Regions" objects, and dedicate the first page of the pool to the array. The Region object is as follows:

```
class Regions{
    public:
        unsigned long region_base;
        unsigned long region_len;
};
```

Fig. 3 Region Object

The size of Regions object is 8 bytes. Having a 4KB page, we limit our regions to 512. This is enough according to the assignment instructions. This array keeps track of allocated regions, and is used to check address legitimacy by traversing each allocated regions and see if the address lies in between any of the regions.

Thus, we complete the assignment, and all tests pass.