Design Report

This is the implementation details for Machine Problem 4 to design memory allocation with page table and virtual memory. In this assignment, the page table class is extended to support large sizes of address spaces and a simple virtual-memory allocator is also implemented. In this assignment, I received guidance from Tim Robinson's tutorial "Memory Management 1" and Memory Mapping With A Recursive Page Directory to implement part of the functionalities.

Page Table

The main change in the implementation is to use recursive page table look-up strategy to locate page directory and page table in the given logical address because page directory and page table are now in virtual address space now. So, the first thing is to change the memory location of page tables from kernel memory area to process memory area.

```
page_directory = (unsigned long*)(process_mem_pool->get_frames(1)*PAGE_SIZE);
page_table = (unsigned long*)(process_mem_pool->get_frames(1)*PAGE_SIZE);
```

The rest of code remain unchanged in the page table construction except that the last entry in the page directory needs to be directed to the start of the page directory to make it recursive.

```
page_directory[1023] = (unsigned long)page_directory|3
```

Second, I have to hook the virtual memory object with the page table class. As per the handout, each page table can be linked to many virtual memory object, so I created an array of virtual object pointer to link this page table with other virtual memory object.

```
for (unsigned int i = 0; i < VM_POOL_SIZE; i++)
    vmpool_list[i] = NULL;
```

For each page table register process, I will search the entire virtual pool array to find the non-linked one and then linked it with the virtual memory object, namely, `vmpool_list[vm_index] = _vm_pool`

When the page is requested to release, a simple call to process memory object can be made to release the page once the page number is available, and then reload the page table to refresh the TLB.

```
void PageTable::free_page(unsigned long _page_no) {
    process_mem_pool->release_frames(_page_no);
    load();
}
```

When page fault occurs, the page fault handler will be invoked to handle the page fault and the logical address will be provided to the handler to modify the page table and load pages from main memory. So the first thing is to find the page directory. Now the page directory is in the virtual address space and with the recursive look-up approach, we know that the page directory starting address is from `0xFFFFF000`, and the page table starting address in virtual space is from `0xFFC00000`. For the given logical address, the shift for page directory and page table can be easily obtained through shifting the logical address 22 bits and 12 bits. Then check if the presence of page table in the given page directory, if not exits then create one, otherwise check the entries in the page table. Since page fault occurs, we know that the page table entry is not marked with presence. Then, we need to create one page table entry with the entry from the physical address and mark it with presence.

```
page_addr = read_cr2()
page_table_index = page_addr>>22
page_frame_index = ((page_addr>>12) & 0x3FF)
page_directory = (unsigned long*)(0xFFFFF000|
page_table = (unsigned long*)(0xFFC00000|(page_table_index<<12));
page_table[page_frame_index] = (unsigned long)(process_mem_pool->get_frames(1)*PAGE_S
IZE)|3;
```

During the above page fault handling process, a logical address check is required because we need to know if the logical address is valid or not. To do that, we can simply iterate the entire virtual pool list, and check if the logical address is within the range of allocated logical address.

```
int check_legitimacy(VMPool **vm_pool, unsigned int _size, unsigned long page_addr) {
    for (unsigned int i = 0; i < _size; i++)
    {
        if (vm_pool[i] == NULL) num_null += 1;
        else if (vm_pool[i] != NULL){
               if (vm_pool[i]->is_legitimate(page_addr))
                    return i;
        }
    }
    return -1;
}
```

Virtual Memory Object

The purpose of virtual memory object is to allocate memory in the logical address space and it also needs to remember the start address and size of the allocated memory. So I use a array to save this information. In order to describe the region information more conveniently, I used a struct data structure to store the start and size information.

```
struct Region_Descriptor {
        unsigned long start;
        unsigned long size;
};
```

For each virtual memory pool object, I used the below variables to save related frame pool description, page table description, and region description.

```
unsigned long        base_address;
unsigned long        size;
ContFramePool        *frame_pool;
PageTable            *page_table;
Region_Descriptor    mem_info_list[REGION_SIZE];
unsigned int         region_count;
```

Now let's look at the individual method to allocate, release and check legitimacy of memories.

Memory Allocation

I used a variable region*count to count how many regions are allocated, and the array mem*info_list is to store all the information for all allocated regions. This makes the allocation is very simple. For each allocation process, we just need to store the start and size information of the current region, and no need to allocate any physical memory in this step because allocation is lazy. The code is below and I recalculate the `_size` variable because I want it to be multiples of `PAGE_SIZE`.

```
_size = ((_size/PAGE_SIZE) + (_size%PAGE_SIZE>0?1:0))*(PAGE_SIZE);
search_start = mem_info_list[region_count-1] + mem_info_list[region_count-1].size
mem_info_list[region_count].start = search_start;
mem_info_list[region_count].size = _size;
region_count++;
```

Memory Release

The memory release step is a little bit complicated because we need to identify which region needs to be released. Not only we need to know the start address of the region, but also we need to know the size of the region. To do so, I iterate the region array to find the region index.

```
index = -1;
for (i=0; i < region_count;i++) {
    if(mem_info_list[i].start == _start_address) {
            index = i;
            break;
    }
}
```

Then, I find the logical start address of the region and size of the region. Here, I need to change the region size to be the number of pages with `total_page_no = (mem_info_list[index].size/PAGE_SIZE)`. Then, we can release the physical memory assigned to this region page by page with a for loop.

```
for (i = 0; i < total_page_no; i++) {
    //page_table->free_page(_start_address);
    frame_address(page_table, _start_address);
    _start_address += PAGE_SIZE;
 }
```

In this for loop, the address used is the logical address, but the page table class only releases page frame number, so the page number needs to be translated into page frame number. Here, the same recursive technique is used, retrieve page directory offset and page table offset from starting address through shifting of the logical address, then locate the page directory and page table accordingly. After that, the corresponding presence bit needs to be set to not presence and frame can be freed through page table release function `page_table->free_page(page_no)`, then the array can be restructured because some entry in this array is removed.

```
table_index = address>>22;
frame_index = ((address>>12) & 0x3FF);
page_table_addr = (unsigned long*)(0xFFC00000|(table_index<<12));
    if (page_table_addr[frame_index] & 1) {
        first_frame_no = (page_table_addr[page_frame_index])/PAGE_SIZE;
        page_table_addr[page_frame_index] = 2;
        page_table->free_page(first_frame_no);
    }
```

On step further, we can check the presence of all the page table entries. If all table entries are marked as not being present, then the frame for that table can be released also. For me, I think it is good to release the frame for the page table if all of its entries are marked as not being present.

```
    for (i = 0; i < 1024; i++) if (page_table_addr[i] & 1) break;
    if (i == 1024) {
         cur_page_directory = (unsigned long*) 0xFFFFF000;
         page_no = (cur_page_directory[page_table_index] & 0xFFFFF000)/PAGE_SIZE;
         cur_page_directory[page_table_index] = 2;
         page_table->free_page(page_no);
    }
```

The page table class is responsible for release of physical frames and reload the page table to refresh TLB.

Legitimacy

Now we need to implement the method to check legitimacy of a given logical address. The checking process is simple and use the brute force search to iterate the entire array and if the address falls inside the region, then it means the address is legitimate.

```
for (i = 0; i < region_count; i++) {
    Region_Descriptor rd = mem_info_list[i];
    if(rd.start <= _address && _address < (rd.start + rd.size))
     return true;
}
```

Otherwise, the method will return False.