Design for MP3

1. Implementation of page table

1.1. Constructor

— PageTable()

In MP3, there two changes in the PageTable constructor.

First, since there is only one page directory, we still store it in the kernel memory pool; however, we move our page tables into process memory pool to save space for limited kernel memory. Thus, the change is that we get our first page table from process memory pool. Because the paging has not been enabled yet, we can use the physical address of this page table to manipulate its entries.

Second, we fill the address of the page directory into its last entry, in order to implement "recursive page table lookup" scheme.

1.2. void handle_fault(REGS* _r)

There is a little change in the page fault handler. Every time a page fault occurs, we first check if the virtual address is legitimate towards the virtual memory pool. Only when it is legitimate, we get a page for that address.

1.3. void free_page(unsigned long _page_no)

First of all, we need to guarantee that the page number is present. If it is present, we free that page. To implement that, because now we are in protected mode and we do not know the virtual address of that page table, we use the "recursive page table lookup" scheme — that is, we use a fake address <1023><page directory index><page index><00> to manipulate the page table. The address can be considered as the virtual address of that page table and we can therefore manipulate its entries like an array.

There are two things we do in the page table. First, we mark invalid the corresponding entry in the page table and free the corresponding frame. Second, we check if all the entries in that page table is invalid. If that is the case, we free that page table as well and mark invalid the corresponding entry in the page directory table and free the corresponding frame. Since the page directory is in the directly mapped memory pool, we can use its physical address as its virtual address.

1.4. void register_vmpool(VMPool* _pool)

In this function, we want the PageTable know that now it is connecting which virtual memory pool. So we just simply let vm_pool = _pool.

2. Implementation of virtual memory pool

2.1. Constructor —

VMPool(unsigned long _base_address, unsigned long _size, FramePool* _frame_pool,
PageTable* _page_table)

In the constructor, we set some essential variables and get this vm_pool registered.

We use a structure to record the allocated regions — the structure contains start address and the size of that region, both of which are unsigned long type.

We need to leave some space for storing the region information. Here, we leave the first 5 frames to store region information. To implement that, we record that the first 5 frames of this virtual memory pool have been allocated so that they will never be touched.

2.2. unsigned long allocate(unsigned long _size)

First, we register this virtual frame pool to the current page table.

Here we use the first-fitted scheme, that is, the first region we find that can fit the required size. If we find that, simply record the start address and the size and put the structure in the region information area. Note that we always keep the allocated regions sorted according to its base address, so we need to insert that newly-allocated region into the list and rearrange the following regions. The reason why we keep them sorted is that this supports searching better. Every time we want to allocate a region, we just traverse the former part of the list for searching and traverse the latter part of the list for rearrangement, which takes O(n). Otherwise, searching will be complicated, time-consuming and hard to implement.

2.3. void release(unsigned long _start_address)

First, we register this virtual frame pool to the current page table.

Then, we need to find the region that is about to release. Since we already record all the start address of allocated regions, traversing will achieve that. Similarly, to keep the list sorted, after we find the region to release and remove it, we need to rearrange the following part of that list.

Also, we need to tell the page table to free the corresponding page. Calling free_page function will do that. Every time we free a page, we need to call load function again, in order to flush TLB. The reason is that TLB may still contain the invalid page entry that we already released. When that address is referenced, TLB may lead to an address that no longer belongs to the process.

2.4. BOOLEAN is_legtimate(unsigned long _address)

We just traverse all the regions allocated to see if the address falls into the range of one of them. Note that in the constructor, we tell the system that the first 5 frames have been reserved, we wants to write this information at the base address. Since we never left any frame for that, a page fault handler will occur. In the page fault handler, it wants to check if this address is legitimate or not — however, this information has not been written to memory yet at that moment, so the is_legitimate function will return false. Consequently, the region is never written and will calls page fault handler infinitely.

To avoid that, in is_legitimate function, we need to let the function know that the first 5 frames from the base address must be legitimate. Therefore, every time we want to record allocated region information, we can write into memory because the address must be legitimate and page fault handler will take care of that if it needs more frames. Also, since we already record that the first 5 frames have been allocated, we guarantee that these frames will never be allocated for other usage.