MP3 Design Documentation

Name: Shiyang Chen   UIN: 623009273

PageTable

I did modification on following functions within PageTable class: PageTable constructor, handle_fault(REGS * _r), free_page, register_vmpool. Let's check them out one by one.

PageTable constructor:

First we allocate a frame(4KB) from process memory pool to store page directory, then store the base physical address of that frame (4KB-aligned, so the lower 12bit are all zeros) into the last entry (1024th entry) of the page directory. Still, we need to map first 4MB physical address onto first 4MB virtual address so that any variable(basic data type and objects of a class) defined before paging is on can still work well after paging is on.

PageTable::handle_fault(REGS * _r)

In this function, we actually do the same thing as what we did with MP2. But the problem is, page directory is no longer stored in first 4MB physical address and paging is on so we cannot use symbol "page_directory[i]" to denote ith entry in page directory since CPU now treats variable "page_directory" physical address as virtual address and make another undesirable(in our view) transformation.

Therefore, we need to transform page directory's physical address into virtual address and use virtual address to denote a page directory and its entries.

Since we use the recursive page table look up, virtual memory "0xFFFFF000" actually points to the first PDE in page_directory and any page table virtual address can be denoted by "0xFFC00000 + (pdindex * 4096)" where pdindex is the 10bits in the middle of a faulting virtual address. Some detail can be found in comments of the codes.

PageTable::free_page(unsigned long _page_no)

This function does two things: release the physical frame that relates to the specified page and update the page table entry accordingly.

First we compute the base address of a virtual page by multiplying _page_no by 4KB. Then we look up the corresponding physical frame in page table and get the base address of it. We divide it by 4KB to compute frame number and call release_frame() function to put this frame into free frame list. After all this, we set the value in corresponding PTE as "non-present".

PageTable::register_vmpool(VMPool *_pool)

This function will be called in VMPool constructor each time we create a virtual pool memory. Here I simply use an array with 10 elements each stores the start address of a separate virtual memory pool. Since in Kernel.C, only two VMPool object are created, the other 8 elements in the array are NULL.

VMPool

The most challenging part of this class is how to allocate a region(containing multiple of pages) and how to get this region back into the pool. I made up a new class called Regions, each Regions object stores the start address of the current region, the number of bytes it has, the number of pages it has, a pointer points to the last region and a pointer points to the next region.So, it's basically a two-way linked list.
I also use an array (I call it "indicator") to indicate each virtual page's current state: "1" means it has been allocated within a region, "0" means it's free to use.

VMPool constructor
In addition to passing all regular parameters into the constructor, I initialize Regions type pointer "pt_firstRegion" as NULL to indicate there hasn't been any regions allocated. Besides, PageTable::register_vmpool is called. Its parameter is the VMPool object being created, so we use "this".

VMPool::allocate(unsigned long _size)
I use First-Fit strategy to allocate multiple of pages to satisfy a request on bytes, there might be internal fragmentation and start addresses of all allocated regions are 4KB-aligned. Meanwhile I sort those allocated regions by their start address and differentiate several cases when allocating.

1. If there hasn't been any allocated region: simply create a new Regions object and make pt_firstRegion points to it.
2. If there has been some regions and the request can be fulfilled by space that span from start address of the current virtual memory pool to start address of the first region, then we creat a new Region object, make pt_firstRegion points to it, set pt_firstRegion's next region pointer to previous first region.
3. If one interval between two regions can fulfill the request, just insert a new Regions object into it and alter relationships between pointers accordingly.
4. If space after the last allocated region can fulfill the request, simply append a new Regions object at the end of Regions list.
5. If request cannot be satisfied anyway, return 0xFFFFFFFF assuming this couldn't be a normal start address

More detail can be found in comments of the codes.

VMPool::release(unsigned long _start_address)
After function get a start address of an allocated region, it starts searching this value over the linked list of Regions defined above until it locate a region that match. There is two cases of searching result.

1. If a Regions object is found, delete this object and alter relationships between

pointers accordingly.
2. If such region does not exist, display words “no such a region with a start_address you specified” on screen.

More detail can be found in comments of the codes.

VMPool::is_legitimate(unsigned long _address)

This function acts as a “forerunner” of page_fault function: if the virtual address we references is not in any allocated region, it makes up an error itself so there is no need to call page_fault function to give it a physical address at all.
First, we divide given virtual address by 4KB to get corresponding page number. Then we check out this page’s current state by look up “indicator” array: if its value is 1, then given address is valid. Otherwise it’s not.

Problem

Using new operator to allocate a memory for Regions object seems like a bad try and I have tried to use malloc but it didn’t work out either. I am figuring it out by using VMPool::allocate instead. Perhaps the problem won’t be sorted through before the due, but it worth trying anyway.