Ian Matson
CSCE 410

P4 Part III Design Document

Note: The makefile was unmodified during this iteration of the project, as I ultimately decided to use .img files rather than .elf.

VM Pool

Structs:

RegionInfo: The RegionInfo struct is used to keep track of allocated regions within the virtual memory pool. It contains only two data members: *start_frame*, which is the address of the first page of the region, and *size*, which is an unsigned long containing the number of bytes in the region.

Data-Members:

base_address: The base address (virtual) of the memory pool. This is where the pool begins in virtual memory.

size: An unsigned int containing the number of bytes in the virtual memory pool.

frame_pool: A pointer to an instance of the ContFramePool class. This corresponds to the frame pool that the virtual pages are stored in when needed in physical memory.

page_table: A pointer to the instance of the PageTable class that the virtual memory pool belongs to. This is the page table that the pool will register with in the constructor.

num_allocated: An unsigned int maintaining the size of the list *allocated_list.*

allocated_list: A list of RegionInfo instances representing each of the currently allocated regions within the virtual memory pool. The list itself is stored in the first page of the pool.

Methods:

VMPool: This method constructs the VMPool instance. First, it ensures that the base address is a page boundary, so that no management between pages is needed. Then, it initializes the data members *base_address, size, frame_pool,* and *page_table* with the passed parameters. Next, it constructs a RegionInfo object corresponding to the first page, and inserts it into *allocated_list* to indicate that the first page is used by the list itself. Lastly, it registers the virtual memory pool with the provided page table.

allocate: This method simply allocates a region of virtual memory with the specified number of bytes. At the start of the method, the requested size is rounded to an even number of pages to prevent inter-page memory management. Once this is done, the method simply iterates through the list of allocated segments checking to see if there is enough space before or after each region. Upon finding any gap with enough space, the address and size is inserted into the list of allocated regions and the address is returned.

release: The release method, when provided the starting address of an allocated virtual memory region, will iterate through all regions and free the corresponding region. This is done by simply iterating through *allocated_list* and finding the entry with the matching *start_frame*. Once this is done, the PageTable method *free_page* is called on each contained page, and the RegionInfo object for the region is removed from *allocated_list*.

is_legitimate: This method is used to determine if the provided address part of any of the allocated regions in the VMPool instance. This is done by iterating though each allocated region in *allocated_list* and checking to see if there exists a region such that the provided address is inside of it. If so, the address is legitimate. An interesting exception is made in this method- if there are no allocated regions, then the method returns true. This is because when the first entry of *allocated_list* is initialized in the constructor, a page fault occurs and the *is_legitimate* method is called. Therefore, it must return true to allow for this single allocation.

insert_item: This method is used to insert a RegionInfo object into *allocated_list* at the provided index. This is done in O(n) time by shifting all subsequent entries back one index and inserting the new item.

remove_item: This method is used to remove the RegionInfo object at the provided index from *allocated_list*. This is done by shifting all subsequent entries forward and decrementing *num_allocated.*

Page Table

Data-Members:

PageTable: A pointer to the currently loaded page table object. In future projects, this will change whenever the process changes.

paging_enabled: An unsigned int indicating whether or not paging is enabled on the system.

kernel_mem_pool: A pointer to the cont_frame_pool instance that maintains the frames for the kernel mem pool. These frames are all directly mapped.

process_mem_pool: A pointer to the cont_frame_pool instance that maintains the frames for the process mem pool. These frames are indirectly mapped using the page table.

shared_size: The total shared size of the address space.

page_directory: A pointer to the location of the page_directory for the page table. This is non-static because it must be individually stored for each instance of the PageTable class.

vmPools[10]: An array containing pointers to all virtual memory pools for the given page table. This array is allocated on the stack and can hold no more than 10 pools. It is non-static, because the registered virtual memory pools will likely be different for each instance of the PageTable class.

num_vmPools: This variable simply maintains the length of the array vmPools.

Methods:

init_paging: This static method initializes paging for the entire system by setting all the static members of the class for the paging system. This includes the kernel/process memory pools and the total shared size. This method must be called before any instances of the class can be constructed.

PageTable: The PageTable constructor creates and initializes the page directory and the first page table in the directory- this coincides with the shared directly mapped portion of memory. Once this has been done, logical addresses can be used.

load: This method sets the static member current_page_table to be the current instance and writes the address of the current page directory into the cr3 register for use by the system.

enable_paging: Sets the paging bit of cr0 to indicate to the system that that logical addresses should be used rather than physical ones.

handle_fault: This is the most complex method of the PageTable class because it does a series of things. The first thing that must be done to handle a page fault is to retrieve the address that caused the page fault and determine which page table and page this corresponds to. If the page table is not present, meaning it is uninitialized, the method gets a frame of memory from the kernel memory pool, initializes all pages inside, and then puts the address of the newly created page table inside the directory, marking it valid. Now, the method must initialize the

page itself. It does this by getting a frame from the process memory pool (this frame doesn't need to be directly mapped, since it is for use by the process and not by the PageTable class) and putting its address in the page table. After these two things have been done, the address is now valid and initialized making it ready for use by the process.

get_middle_10_bits: This method does exactly as it implies- when given an address, it returns an unsigned long containing the middle 10 bits of the address. This method was added to create a simpler and more understandable design.

get_first_10_bits: Much like the above method, get_first_10_bits simply returns the first 10 bits of any passed unsigned long.

contstruct_pte_address: This method is used to construct a virtual address that corresponds to the provided page table entry (the index of the page table and page table entry are passed as parameters). Using this information, the method creates an address wherein the first 10 bits are all 1, the second 10 bits correspond to the page directory entry containing the page table, and the last 10 bits (excluding bits 0 and 1) correspond to the page table entry index. Using this address, the page table entry can be accessed.

construct_pde_address: This method is used to construct a virtual address that corresponds to the provided page directory entry (the index of the desired page directory entry is passed as a parameter). The method does this by returning an address wherein the first 20 bits are set to 1, and the last 10 bits (excluding bits 0 and 1) are set to the index of the page directory entry. In this way, the CPU will index the page table twice using the first two sets of 10 bits, then index into the page table one last time using the last 10 bits to give us the page directory entry.

check_address: This method is used to check a given address to determine if it is a valid address within the page table. It does this by iterating through each registered VMPool inside the list vmPools and calling the is_legitimate() method on the passed address. If any of them report a legitimate address, then the address is valid.

register_pool: This method is used by instances of the VMPool class to register with a given PageTable instance. The method simply adds the passed pointer to a VMPool to the vmPools list and increments num_vmPools.

free_page: This method, when passed a virtual page number, will mark the page as free as well as free the physical frames associated with the page. This is done by using a recursive approach to edit the page table and page directory entries accordingly and calling release_frames() on the passed page. Once this is done, cr3 "reset" to allow the TLB to be reloaded with only valid entries.