CSCE - 611

MP 4: VIRTUAL MEMORY MANAGEMENT & MEMORY ALLOCATION

DESIGN DOCUMENT

Submitted by: RAJENDRA SAHU (731008796)
github-repo : CSCE-410-611-fall-2021/rpsahu_CSCE611

INTRODUCTION:

This virtual memory manager is responsible for allocating and releasing virtual memory as per the demand of the user. The core allocate & release functions are hooked to the new & delete operator in kernel.C. This MP is basically extension of previous MP in 3 directions. The directions are 1. Recursive Lookup in the Page Table 2. Adding extra support in PageTable for virtual memory management 3. Virtual Memory Allocator VMPool. The context & code layout had already been established by the *handout_mp4.pdf* & *MP4_Source.zip*. This document is intended to give the reader an idea/logic of the implementation behind the page table management without having to deduce it from the code which generally should be avoided. The source codes submitted with this guide also has self-explanatory comments aiming for a good code documentation which will turn helpful in future.

SOURCE CODES MODIFIED/ADDED:

1. *kernel.C: Commenting the _TEST_PAGE_TABLE to test the virtual memory allocator.*
2. *cont_frame_pool.o : using the instructor provided frame allocator solution, added this file*
3. *page_table.H: core implementation*
4. *page_table.c: core implementation*
5. *vm_pool.C: Core Implementation*
6. *vm_pool.H: Core Implementation*
7. *makefile: commenting the compilation of cont_frame_pool*

CORE IMPLEMENTATION (Only the incremental additions/modifications have been mentioned, the rest of the code stays same as provided):

1. ***PageTable()* constructor** :
   We prepare the Page Table for the recursive lookup & vm_pool compatible here.

- Allocating frames from the *process_frame_pool* for the the pd & pt since we want to remove the direct mapping
- Establishing the link between the last entry in pd & the pd itself; making the 1023 entry to point to the pd.
- Added a new attribute ***vmpool_list*; the list of all the registered vm pools.
- Added a new attribute *vmpool_list_count;* the count of the number of registered pools.
- We leave everything as it is as per the previous MP.

2. ***PageTable::handle_fault() :*** This is the same static function in the previous MP. There has been addition in this handler to support the features of virtual memory pool. The following is the description.

- Since the page directory & table has been moved to the *process_mem_pool,* we can't follow the logic of direct mapping translation; this is where recursive lookup comes into picture.
- We use the *PageTable::PDE_Address()* & *PageTable::PTE_Address()* to extract the logical addresses of the pde & pte entries as per the logical to physical translation mantra.
- We use the above logical addresses to access the pde & pte.

- Added an extra check of address_legitimacy to check if the logical address that triggered the page fault is legitimate or not. Since the vmpool allocator is a lazy allocator frames aren't allocated unless it page faults, hence it is important to check if such an address was actually allocated by the user or not. Not to be confused with address being valid (PRESENT bit in the pte or pde). The *VMPool::is_legitimate()* is called on each of the registered pool.

3. ***PageTable::register_pool():*** This function by every newly constructed VMPool object to register themselves in the list maintained by PageTable. We add an entry to the *vmpool_list* & increase the count.

4. ***PageTable::free_page():*** This function is called by *VMPool::release()* whenever a page is getting released. This function in turn calls the *ContFramePool::release()* which actually releases the physically allocated frame. The page is marked invalid & the TLB is flushed here too.

5. ***VMPool() Contructor:*** We add two more attributes on top of provided VMPool class. Two lists *allocated_list* & *free_list*. These attributes as their name suggests store info about the allocated & free regions of that vmpool instance. These lists are arrays of a structure *base_address* & *size* demoting the specific memory region (free or allocated). We add one more flag attribute which will be helpful in accessing the above two lists for the first time to set their entries. We also allocate an actual page for the above 2 lists and split into two equal halves each for a list. We explicitly use the *base_address* of the vmpool object to allocate the page since the *VMPool::allocate()* is not ready yet. Then we set the first entries in both lists logically implying we have allocated a page. We set the other entries to zero & zero (base address & size).

6. ***VMPool::allocate():*** The function responsible for allocating page (lazily, not allocating the frame immediately. It traverses the *free_list* searching for a free enough memory region as per the requested no of bytes. Once it finds the index of such a memory region, it deducts the requested no of bytes from that region and add an appropriate entry of that mush size in the *allocated_list.* It returns the base address of the allocated memory region. If it couldn't find a big enough free memory region it returns zero.

7. ***VMPool::release():*** This function traverses the *allocated_list* looking for the memory region starting with the given memory address. Once it finds that region, the specific region entry is invalidated (reset to 0) in the *allocated_lis*t and then an appropriate entry is added to the *free_list.* At the end we call page_table::free_page() on that memory region.

8. ***VMPool::is_legitimate():*** This function traverses through the *allocated_list* to check if the passed logical address belongs to any allocated memory region or not. It returns either true or false. The first call to this function is from the page fault handler when it tries to access the list in constructor to set the entries. The flag attribute helps us to determine that its this specific case and we need to pass true address_legitimacy so that the entries can be set.

9. ***kernel.c:*** Commenting the *_TEST_PAGE_TABLE* to test the virtual memory allocator.

10. ***VMPool.***H: Added the list (*allocated_list , free_list*) entry structure datatype i.e. *base_address & size*.

11. ***PageTable.***H: Added PT_LOOKUP & PD_LOOKUP macros to define the 1023 constant in the recursive lookup.

TESTING:

The instructor provided frame allocator have been used. Increased the size of code_pool & heap_pool to 512 MB. An then increase the GenerateVMPoolMemoryReferences() parameters to (100, 200) for both of the pools. The test passes on my implementation. But the delivered code is set on the original parameters.

CODE MAINTENANCE:

The entire source code base has been pushed to the student's github repository. This have been done progressively over the course of development. This should help the TA in grading the assignment. Apart from the code base provided in the zip file submitted on canvas, the code can also be compiled from the github repo code base.
The instructor provided *ContFramePool.o* was pushed to the mp4 repo too. Please take note the compilation of

ContFramePool is commented in the *makefile*. Keep the ContFramePool.o handy somewhere else in case the user deletes it using the make clean command. If *make clean* command is used, make sure to copy paste the object file solution to make the code complete.

GENERIC INSTRUCTIONS:

This code has been developed on the new code environment.
The TA is expected to unzip the file and the run *make.* If *make clean* command is used, make sure to copy paste the ContFramePool object file solution to make the code complete.