Design notes

Mian Qin UIN:725006574

1 Overview

In this machine problem, we mainly implement three parts. First part is to modify the previous page table implementation in MP3 to allocate frame for page table page from process pool rather than kernel pool. Second part is to implement a simple virtual memory allocator to allocate and free memory in virtual memory address space. Third part is to hook up the VM allocator with the page table to implement lazy frame allocate and release frame from page table.

For the first part we need to use reverse lookup (access physical address in virtual memory mode) to make it works. In my implementation, I keep the page directory page in kernel frame pool, but move all the page table pages into process frame pool. In page fault handler we only need to slightly modify the page table page pointer to utilize reverse page lookup to access content in the page table page. However, in the PageTable constructor, it's a little bit tricky. In the PageTable constructor, we need to setup the page table for the kernel memory first (first 4MB direct map memory). If we move the page table page to process pool, there is an issue when we construct more than one page table, since the first time we construct PageTable we are in real mode, the following constructions of PageTable are in virtual memory mode. So, we need check in the PageTable constructor whether we enabled paging and do different things. If we enabled paging, we need to setup a pte for the new page table page in the current page table and initialize the new page table ptes.

For the second part implementing a simple virtual memory allocator.

I implement a simple but rather practical virtual memory allocator. I used two fixed size arrays (512 entries) to store the metadata. One is called occupy region list, the other is called free region list (region is defined as a 8 byte data structure, 4 byte for start address, 4 byte for size). The free region list is initialized as the whole vm_pool virtual space except the first two page of metadata.

Every time we do allocation we will consult the free region list and find a region that can split part of it to serve the allocation request and adjust the free region list. If no free region can meet the request, we print "Out of virtual memory " and assert. Then we add a new entry to the occupy region list for further release and legitimate test.

Every time we do de-allocation, we first find the region in the occupy region list and remove the entry in the occupy region list (by swapping the last entry in the occupy region list). Then we call the free_page() in the corresponded PageTable object to actually release the physical frames. Finally, we will traverse the free region list to see whether we can coalesce with a entry in the free region list, if not we add a new entry in the free region list for further allocation use.

The occupy region list and free region list can work together to dynamically allocate and free virtual memory. If you release the allocated memory (delete) timely (for example, the default test case in kernel.C), you can do allocation and de-allocation infinitely. However, if you don't release memory timely, there may have a external fragmentation issue.

For the third part, we need to hook up the virtual memory allocator with the page table. We use lazy frame allocation, when the virtual address allocated by the VMPool is referenced, a page fault occurs due to pte not present, in the page fault handler, we allocate physical frames according to the frame pool associate with VMPool and set up the corresponded pte. When release the memory allocated by the virtual memory allocator, we need to call the free_page() in the PageTable to invalid the corresponded pte (also need to flush the TLB).

2 Implementation

2.1 PageTable::PageTable()

In PageTable constructor, in order to move the page table page from kernel pool to process pool, I need to do certain modification. First, get page table frame from process pool. Second, for the initialization part (kernel direct map page table), we need to differentiate whether we are in real mode or virtual memory mode. If in real mode. Is simple since cpu issues physical address, if in virtual memory mode, we need to use reverse look up to set up the kernel page table (this happens when more page table are initialized). More details are covered in the overview part.

2.2 PageTable::handle_fault(REGS * _r)

For the page fault hander part, basically I changed to parts, first one is to check legitimate of the faulted address. This is done by traversing the registered VMPool instances and call is_legitimate(). Once I find corresponded VMPool, I will then further allocate frames using the associate frame pool to the VMPool. If not, I will panic "segmentation fault".

2.3 PageTable::register_pool(VMPool * _vm_pool)

For the VMPool register, I use the similar trick like the cont_frame_pool in MP2. In VMPool class, I use a pointer to link all the VMPool object which belongs to a same PageTable, and in the PageTable object, there is a VMPool pointer that store the header of the VMPool linked list. The register_pool() is responsible for linking all the VMPool registered to the PageTable, and the PageTable page fault handler will traverse the VMPool list to check legitimate.

2.4 PageTable::free_page(unsigned long _page_no)

For the free_page() function, it will call the release_frame() function in the cont_frame_pool to remove the pte in the page table and also need to flush the TLB since the page table has made a change.

2.5 VMPool::VMPool

As the design of VMPool allocator stated in the overview section. VMPool have two pages of metadata, storing a free region list and a occupy region list. In the VMPool constructor, the two region list will be initialized and also VMPool object will be registered to the corresponded PageTable.

2.6 VMPool::allocate(unsigned long _size)

For the allocate algorithm, it will first round up the _size to a page granularity (multiple of page size), then it scan the free region list to find a region that satisfy the memory request size. Finally it will add a new entry to the occupy region list.

2.7 VMPool::release(unsigned long _start_address)

For the release algorithm, it will first scan the occupy region list to find the corresponded entry, if it's not in the occupy region list, then panic and assert. If we find the entry in the occupy region list, then it will scan the free region list and try to coalesce with a entry, if we cannot coalesce, I will add a new entry to the free region list which will cause external fragmentation in this scenario.

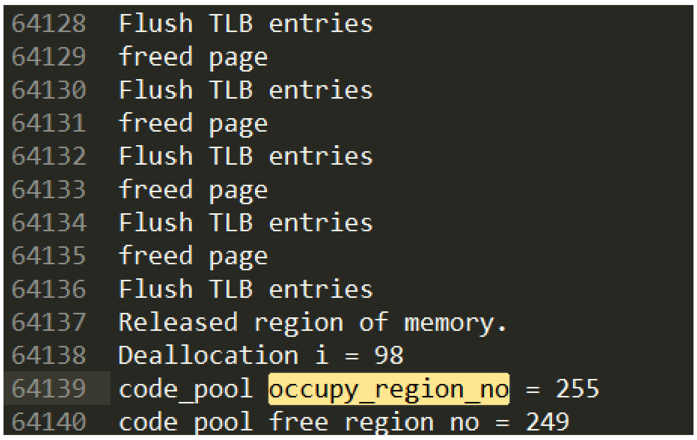
2.8 VMPool::is_legitimate(unsigned long _address)

In the is_legitimate check function, it first whether the input address belong to the metadata area which is legal address area. Then, it will traverse the occupy region list to check the input address.

3 Debug and verification consideration

For debugging, I hook up with gdb to enable debug feature. This is already done in MP1.
For verification consideration, the original kernel.C gives only one test case to test the page fault handler for page table pages allocated in the process pool and simple test for virtual memory allocator. I do more test to further verify the correctness of the virtual memory allocator and multiple Page Tables.

1) More allocate/release pattern. I write two test that have more allocate/release pattern, in the first test, I use a loop to allocate increase size of memory but not release (less than 512 regions), then use a loop to release all the regions, the test pass successfully.
   The second test use a random allocate/release pattern, I will not release all the regions in each loop, but only release half of the allocated regions, this will cause external fragmentation, I changed the free_region_no and occupy_region_no to public and print out as below. This shows the external fragmentation.



```
Flush TLB entries
freed page
Flush TLB entries
freed page
Flush TLB entries
freed page
Flush TLB entries
freed page
Flush TLB entries
Released region of memory.
Deallocation i = 98
code_pool occupy_region_no = 255
code_pool free_region_no = 249
```

2) Multiple page tables, after testing allocate/release pattern, I setup another Page Table, and initialize a new VMPool associate with the page table and do more test, it still works fine.
3) Infinite allocate/release. I write a test that allocate/release a lot of times the total memory size is much larger than the initial virtual memory space. After memory test, print the occupy_region_no and free_region_no, it doesn't change compare to the beginning of the test, which means we can infinitely allocate/release memory.