P4 DESIGN

Operating Systems – CSCE 613

ABSTRACT

Virtual Memory Pool

Tharun Battula

82400197

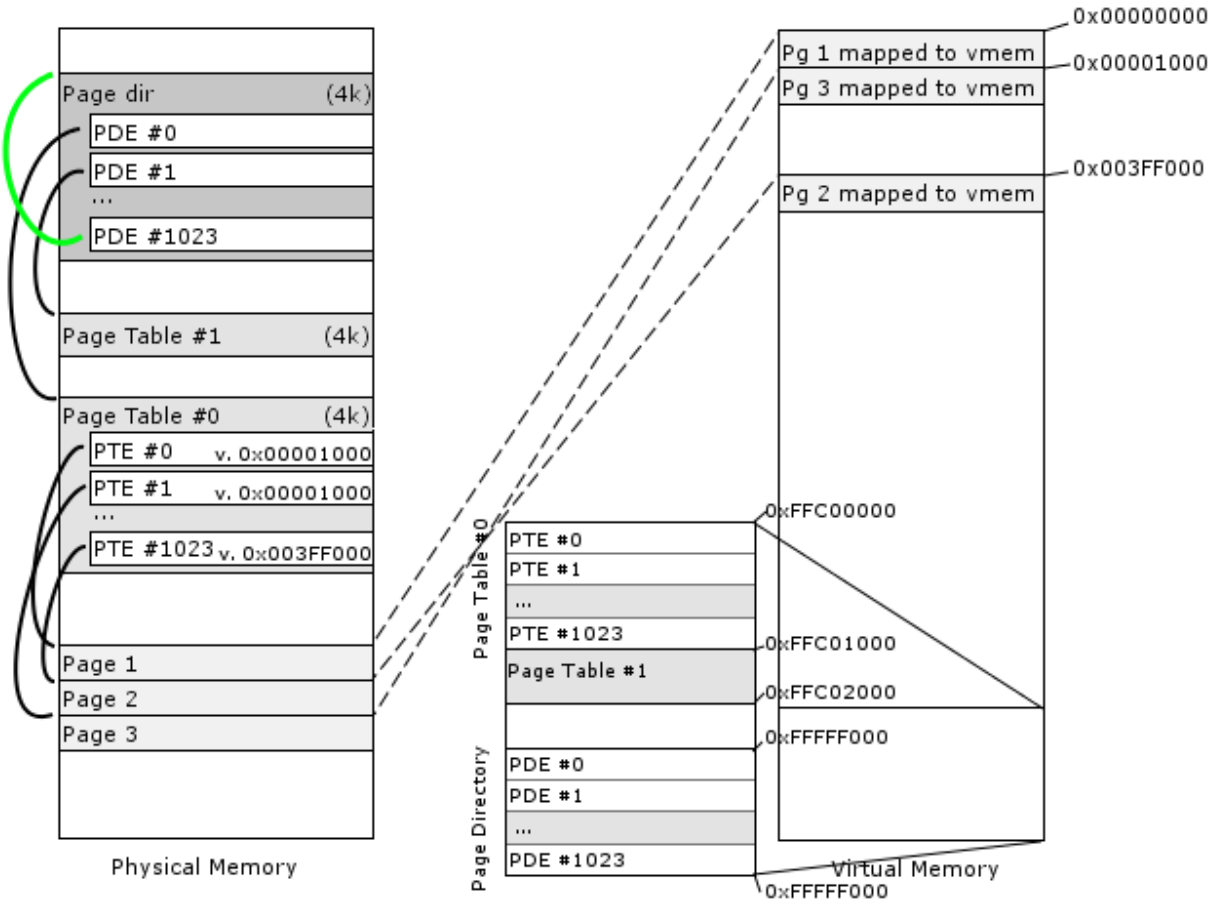
This is an extension of P3 to P4 with Main Targets additions as

1. Recursive Page Lookup for the previous implementation
2. Virtual memory management is implemented to allocate and deallocate regions in virtual memory pool for “new” and “delete” functions respectively.

DESIGN

- The specifications mentioned in the assignment are followed
    - 32 MB of memory layout, Kernel Pool 2MB to 4MB
    - Process Pool beyond 4MB, Inaccessible region 15MB to 16 MB
- The P2 assignment structure with frame pool class is retained
    - Kernel Memory Pool
    - Process Memory Pool
- Recursive Page table lookup
    - After the paging is enabled, it becomes difficult to access a page table with physical address. The get_frame() function gives page table location in terms of physical address space. For process pool obtained frames, Its corresponding virtual mapping might not be have been mapped in the page directory lookup.
    - The Idea to mitigate the above challenge is using recursive page table, where when we create Page directory we point the page directory last entry to page directory itself.
    - This way for the virtual addresses above 0xFFC0 0000 looked up last entry of PDE .i.e., Page directory itself recursively. Hardware will map PDE last entry to PDE as page table. Now 0th entry of PDE(i.e.) page table maps to physical page table. Similarly if we know directory index of any page table we can access the contents of page table by virtual address by looking up (0xFFC0 0000 +(dir_index<<12))

- For more understanding on details refer to this below diagram and reference



Reference: http://www.rohitab.com/discuss/topic/31139-tutorial-paging-memory-mapping-with-a-recursive-page-directory/

Modifications

- Modifications in page_table.H to add some additional variables
  VMPool *pt_pools_list[VM_POOL_LIST_SIZE];
  - Data Structure to keep track of all the various VM pools in single array.
  - The Maximium VM_POOL_LIST_SIZE is currently set to 100, based on the data structure of vmpool size as calculated quotient with page size.
  - For this code it only uses 2 in the given example, it can surely accommodate more.

- Modifications in page fault handler
  Implementation for recursive page directory, the above mentioned design logic is added over P3 implementation and it is tested
- Extra API is added to allocate memory exclusively using kernel memory pool for vmpool data structure purpose details explained in the next section.

There are two cases for faulting address:

- Case 1. Page Directory "valid" index is NOT set – this means page table is not available in memory. Thus, firstly, get a frame and set a page table. Then make an entry of address of that page table physical memory location in page directory at appropriate index.
  Now, get a new frame for mapping physical memory for data to page table and make an entry in page table at appropriate index. Clear all other entries of newly created page table.
- Case 2. Page Directory "valid" index is set – this means page table is already available in memory. Thus, just get a new frame for mapping physical memory for data to page table and make an entry in page table at appropriate index.
- Checking the faulting address using is_legitimate function

When page fault occurs, we need to check if the faulting address is within the bounds of the region that is currently allocated through list of VM Pools residing in the same page table. If it is not, then it is a system error. Thus, in my implementation, if this error occurs, program will terminate executing further and will go into infinite loop.

The above mentioned design with assignment mentioned API guidelines is coded.

- To debug the code → page_table.c file make debug enable flag to 1.
  `#define DEBUG_ENABLE 1`
- To Print memory footprints
  `#define MEMORY_PRINT 1`

Free page function

This is new function added in the PageTable implementation to set an entire page to invalid at once with the entry and freeing the memory for further allocation. This function can be called by vm_pool.C. It takes page_address as

input and according to the store information in the vm_pools it correspondly deleted. It is handy function for VMPool implemantations.

Virtual memory Design

For the implementation I maintained two separate pools

`Linked List Element Structure:`

```
typedef struct l_list_elem{
     unsigned long address; /* Starting Address for Storing
     the current memory info*/
      int size; /* Signifying length of the storeage */
      struct l_list_elem *next;/* Next element*/
   }l_list_elem;
```

- `l_list_elem *alloc_list;` → `List maintaining Allocated Memory`
- `l_list_elem *avail_list;` → `List holding Available free memory`

Initialization:

At first there holding the data structures for each VMpool is tricky as the paging is enabled. The new frames obtained from process pools are obtained from process memory pool which give physical memory as the output. It is difficult to align the virtual memory accordingly. It becomes challenging with the page fault handler trying to verify memory range check before initialization for the first page fault itself giving rise to chicken and egg problem. This is solved by getting frames from kernel memory pool as they have direct memory mapping and paging would not cause any issues.

For this purpose an extra API is created in the PageTable just as a proxy to get hold of kernel memory frame

```
unsigned long allocate_kernel_frame();
  /* This function is exposes allocation exclusivelly a
kernel frame,
   * this can be used by vm_pool class for allocating for
holding
   * core data structure intact */
```

1. Register VMPool function

Input of this function is the pool that is created in page table class. There could be multiple pools in each page table class. An array data structure holds list of all list of vmpools per page table.

This pool is registered by storing the VM pool data structure (start address, size and valid) in a separate variable “Registered pool”.

Note that a frame is allocated to store this information of registered pools.

2. Implementation of VM Pool

VM Pool is used to allocate virtual address space to the variables defined by using “new” and to deallocate space by using “delete”. In this way, VM Pool takes care of allocation and deallocation of pages. When pages are freed or allocated, frames are also allocated or released respectively. (For example, when region in VM Pool is released, it will free the virtual address space and call page_table.C to free up corresponding frames in physical address space).

There could be multiple virtual memory pools in the program and each pool has multiple regions allocated to the arrays or variables.

First we need to define a data structure for managing virtual memory pool. This data structure is a structure which consists of (1) Start Address of the region, (2) Size of the region (3) Valid entry telling if the region is valid or now.

2.1 Constructor

1. Store the base address, size, frame_pool and page_table in local variables
2. Initialize region number (to track number of regions in the pool) to 0
3. Define maximum number of regions that the VM pool can hold. One frame is allocated to manage the VM pool data structure. Thus maximum number of regions is equal to size of pool divided by size of data structure
4. Allocate a frame to the data structure for managing VM Pool
    - It uses kernel frames for holding the core data structure to avoid paging issues of chicken and egg.

5. Register the VM pool in page table

2.2 Allocate function

This function is called when there is “new” array/variable created in the program. In this project, I used first fit algorithm to allocate the region.

1. If it is the first region, then start address of that region is base address. Else look if there is empty region with valid bit not set where new variable can fit into (first fit algorithm implemented). If nothing works out, then allocate a new region within VM Pool.
- The data structure is lists for maintaining the two types of memory is used. In the available frames it is allocated continuously based on the pool. The starting of address pointer is moved to next address it can point, and size is updated accordingly. Once a region is allocated it is added to list of allocated pools.

2. Keep check on the maximum regions that can be allocated and throw an error if it exceeds the limit.
3. Set the valid and size appropriate for the region in the data structure created.
4. Correspondingly update the allocated linked list – which keeps track of allocated regions in the pool.

2.3 Release function

Start address is passed to the release function. This function is called by the program when we need to “delete” certain variable and free up the space in address space.

1. First, search for the region that needs to be deleted. Find the corresponding region in allocated data structure which keeps tracks of the region allocated in the VM pool.
2. Once the region number is located, call free_page function of page_table.C. Note that this function needs to be called for the number of pages allocated to it (using for loop).
3. Since it is freed memory it can be added in the list of free lists, it is added accordingly. So when a new allocation requests comes. This is directly searched and if it matches the requirements. It is accordingly allocated.

Thus, virtual address space (pages) and physical address space (frames) are released in the page tables and frame pool data structures.

2.4 Is legitimate function

This function checks if the address is part of the regions that are currently allocated in the VM Pool. This function is used to choose appropriate VM Pool to which the variable is allocated.

To find if address is legitimate, iterate through all the linked lists of allocated regions in the VM_pool using linked list traversal and check if the address is within the bound of the region. Base address to base address + size.

Assumptions

- Infernal Fragmentation is not taken care of. If size less than Page size whole page size allocated
- With first fit algorithm, the external fragmentation is also not completely avoided in this scheme, ideally other sophisticated algorithms can be added on top of the current code.
- The sequence of initial steps provided in Kernel Code is assumed. I,.e., PageTable::init_paging will be called any page table functions. Before, Page Table constructor will be called, then load, and enable_paging will be called in the same order
- Paging is setup before it is called. VM pools and PageTables are initialized before using them.
- Page Replacement algorithm is not available. Since in our case 32 MB is physical memory, we will quickly run out of physical memory for higher data requests. The data needs to be freed for other process usage.

Testing

- Tested basic features , Results Passing – Verified
    - Base kernel given as part of given code.
    - Checked different combinations.
    - Repetitive memory references.
- Tested Recursive page table with prints
- PRINTS in the corner areas for every new activity

- MEMORY footprints printed

References:

1. Wikipedia
2. osdev.net – tutorials and wiki pages
3. http://www.rohitab.com/discuss/topic/31139-tutorial-paging-memory-mapping-with-a-recursive-page-directory/