CSCE 613: Project 4 Design

Student: Caio Duarte Diniz Monteiro

Date: 03/14/16

FramePool class

FramPool class implementation is exactly the same from Project 3. No changes were made on it.

PageTable class

The PageTable class was modified to store page directory and page tables on the process memory pool. This change is made to avoid being limited by the 4MB of directly mapped memory of the Kernel. In order to achieve this goal, a technique called Recursive Page Table Lookup was used. Also the PageTable class now keeps track of registered virtual memory pools.

```
static const unsigned int _MAX_NO_POOLS = FRAME_SIZE/sizeof(VMPool*); //define how many entries of VM Pools can be stored on one frame
VMPool** vm_pools; /*keeps track of all the registered vm pools*/
unsigned int pool_index; /*index of the next vm pool to be registered*/
```

Register VMPool

```
void register_vmpool(VMPool * _pool);
/* The page table needs to know about where it gets its pages from.
   For this, we have VMPools register with the page table. */
```

The register_vmpool function is called on the VMPool class contructor to register the VMPool being created into the PageTable class. This function simply store the pool passed as a parameter into the vm_pools array and increment the pool_index used to identify the position of the next pool to be registered.

Free Page

```
void free_page(unsigned long _page_no);
/* Release the frame associated with the page _page_no */
```

The free_page function will receive a page number as a parameter, compute the frame number of it, and call the release frame on the frame pool.

Recursive Page Table Lookup

```
page_directory[ENTRIES_PER_PAGE - 1] = (unsigned long)page_directory;
page_directory[ENTRIES_PER_PAGE - 1] |= 3;
```

The trick to store the page directory and page tables on the process memory pool and be able to later access them is to make the last entry of the page directory points to itself. That way you can access the page directory referencing the last page of the virtual memory.

PageTable handle_fault function

```
static void handle_fault(REGS * _r);
```

The fault handler remains with the same basic functionality. The only differences is in how to access the page directory and page tables. Since we have them now on the process memory pools and we are using the recursive page table lookup technique, we will access the page directory as following:

```
unsigned long* _page_directory = (unsigned long*) 0xFFFFF000;
```

And similarly the page tables are access as an offset of the page directory virtual address:

```
unsigned long* page_table = (unsigned long *) (0xFFC00000 + (directory_entry * PAGE_SIZE));
```

VMPool class

The VMPool class is responsible to manage a virtual address space. Keeping track of available regions and checking if memory accesses are valid.

```
unsigned long base_address;
unsigned long size;
FramePool* framePool;
PageTable* pageTable;

static const unsigned int _MAX_NO_REGIONS = FRAME_SIZE/sizeof(Allocated_Region);//define how many entries of allocated region can be stored on one frame
Allocated_Region regions[_MAX_NO_REGIONS]; //currently allocated regions on the pool
unsigned int region_index; /*index of the next region to be registered*/
```

VMPool constructor

```
VMPool::VMPool(unsigned long _base_address, unsigned long _size, FramePool *_frame_pool, PageTable *_page_table)
```

The VMPool constructor sets the class variables like base_address and size and also initialize the Allocated_Region array with the appropriate value. It starts as a single element representing the whole virtual address space as available.

VMPool Allocate

```
unsigned long allocate(unsigned long _size);
/* Allocates a region of _size bytes of memory from the virtual
 * memory pool. If successful, returns the virtual address of the
 * start of the allocated region of memory. If fails, returns 0. */
```

The allocate function receives a size as a parameter and needs to allocate a region with that size on the virtual memory. To avoid problems the _size is rounded to the next multiple of the page size (4KB). And then the array of allocated regions is iterated looking for a region available and with enough space to hold the requested size. When a region with these criteria is found, the following steps occur:

1. Set the available region size to be just what the request need, storing the size difference as a remaining size;
2. Set this region as unavailable;
3. Create a new available region with the remaining size computed on step 1;
4. Put this newly created region right after the region marked as unavailable, shifting all other regions to the right.

When the array of regions is already full, then we simply execute just the step 2, since there is no room to split this memory region in two.

VMPool Release

```
void release(unsigned long _start_address);
/* Releases a region of previously allocated memory. The region
 * is identified by its start address, which was returned when the
 * region was allocated. */
```

The release function receives the start address of a region as a parameter. It then searches on the allocated regions array for the region with that starting address and once it is found call the free_page function of the PageTable for every page within the region. It will also mark the region as available, so it can be used later by the allocate function, and flush the TLB calling the load function of the PageTable.

VMPool Is Legitimate

```
BOOLEAN is_legitimate(unsigned long _address);
/* Returns FALSE if the address is not valid. An address is not valid
 * if it is not part of a region that is currently allocated. */
```

The is_legitimate function receives an address as parameter and check if this address is currently allocated on the virtual memory pool. In order to do so, it iterates over the allocated regions array searching for an unavailable region which contains that specific address. If a region is found it returns true, otherwise false.