CSCE611: Operating System
MP4: Virtual Memory Management and Memory Allocation

Name: Rahul Ravi Kadam
UIN: 834005554

System Design

The goal of this machine problem is to extend the page table management to support virtual memory and implement a virtual memory allocator. In this machine problem, we allocate page table pages in mapped memory, i.e., memory above 4 MB. These frames are handled by the process frame pool.

Handling Large Addresses Spaces:

As the size of the directly mapped portion of the memory is small, we cannot use it for larger address spaces. And so, we make use of the process memory pool to allocate memory for the page table pages. Moreover, when paging is turned on, the CPU issues logical addresses. Hence, to map logical addresses to frames and modify entries in the page directory and page table pages, we make use of a technique called 'Recursive Page Table Lookup'.

In the method 'Recursive Page Table Lookup', the last entry of the page directory points to the start of the page directory itself. The page directory and page table pages both contain physical addresses.

In order to access a page directory entry, we make use of the following logical address:

| 1023 : 10 | 1023 : 10 | offset : 12 |

The MMU will use the first 10 bits (value 1023) to index into the page directory to look up the PDE. PDE number 1023 (the last one) points to the page directory itself. The MMU treats the page directory like any other page table page.

The MMU then uses the second 10 bits to index into the (supposed) page table page to look up the PTE. Since the second 10 bits of the address also have value 1023, the resulting PTE points again to the page directory itself. Again, the MMU treats the page directory like any frame : It uses the offset to index into the physical frame.

In order to access a page table page entry, we make use of the following logical address:

| 1023 : 10 | X : 10 | Y : 10 | 0 : 2 |

The MMU will use the first 10 bits (value 1023) to index into the page directory to look up the PDE. PDE number 1023 points to the page directory itself. The MMU treats the page directory like any other page table page.

The MMU then uses the second 10 bits (value X) to index into the (supposed) page table page to look up the PTE (which in reality is the Xth PDE). The offset is now used to index into the (supposed) physical frame, which is in reality the page table page associated with the Xth directory entry. Therefore, the remaining 12 bits can be used to index into the Yth entry in the page table page.

In this manner, we can manipulate a page directory entry or page table page entry stored in virtual memory.

Virtual Memory support for Page Tables:

The page table must know all created virtual memory pools to distinguish between a legitimate memory access and an invalid access. To support this, the page table must have a list of all virtual memory pools. Also, when a page fault occurs, we check if the address is legitimate. And finally, we release previously allocated pages.

We support the above-mentioned features through the implementation of register_pool, is_legitimate and free_page functions.

We make use of a linkedlist of virtual memory pool regions available for allocation. Each node of the linkedlist contains information about the base address, size of memory pool region, number of regions and available virtual memory pool size.

Simple Virtual Memory Allocator:

We implement a virtual memory allocator which can allocate and deallocate virtual memory in multiples of pages. Using the allocate function, we can allocate a region of virtual memory within the virtual memory pool. Similarly, using the deallocate function, we can deallocate a region of virtual memory within the virtual memory pool.

An array is used for tracking the allocation and de-allocation of virtual memory regions; where each array element is structure 'alloc_region_info' object type. This structure object holds information such as the base address and length of virtual memory allocated.

List of files modified:

The following files were modified since Machine Problem 3:

1. page_table.H
2. page_table.C
3. vm_pool.H
4. vm_pool.C
5. cont_frame_pool.C

Code Description

1. page_table.H : class PageTable:

In class PageTable, data structures for the frame pools were declared:

- static PageTable * current_page_table - Pointer to the currently loaded page table object
- static unsigned int paging_enabled - Integer flag to indicate whether paging is turned on
- static ContFramePool * kernel_mem_pool - Pointer to kernel memory frame pool
- static ContFramePool * process_mem_pool - Pointer to process memory frame pool
- static unsigned long shared_size - Size of shared address space
- static VMPool * vm_pool_head - Pointer to virtual memory pool linked list
- unsigned long * page_directory - Pointer to page directory

2. page_table.C : init_paging():

This method is used to initialize the private variables for class PageTable. The global parameters for the paging subsystem are setup.

3. page_table.C : PageTable():

This method is the constructor for class PageTable. It sets up entries in the page directory and the page table. The page directory is initialized by allocating a single frame from the kernel frame pool. The last entry of the page directory points to the start of the page directory itself (to allow for recursive page table lookup). The first and last page directory entries are marked 'valid', while the remaining entries are marked 'invalid' using bitwise operations. Next, the page table is initialized by allocating a single frame from the process frame pool. The page table entries for the shared portion of the memory (first 4MB which is directly-mapped) are marked 'valid' using bitwise operations.

4. page_table.C : load():

This method is used to load in a page table by storing the address of the page directory in the page table base register by writing the address into register CR3. Overall, a new page table is loaded and made the current table.

5. page_table.C : enable_paging():

This method is used to enable paging on the CPU. This is done by setting the 32nd bit to 1 and writing the value into register CR0. The 'paging_enabled' variable is also updated to 1.

6. page_table.C : handle_fault():

This method is used to handle page-fault exceptions of the CPU. This method will look up the appropriate entry in the page table and decide how to handle the page fault. If there is no physical memory frame associated with the page, an available frame is brought in and the page table entry is updated. If a new page table has to be initialized, a new frame is allocated for that, and the new page table page and the directory are updated accordingly. This method follows the below algorithm:

- Read the error code of Exception 14 and check if bit 0 is not set to verify whether 'page not present' fault occurred.
- Obtain the page fault address by reading register CR2.
- Obtain the page directory address by reading register CR3.
- Calculate the page directory index (first 10 bits) and page table index (next 10 bits) from the page fault address.
- Iterate through the list of virtual memory pool regions and check if the page fault address exists within any of the virtual memory regions.
- Check whether the page fault is in the page directory or the page table. This is done by checking the 'Present field' of the page directory address.

$31 \cdots 12$	$11 \cdots 9$	$8 \cdots 7$	6	5	$4 \cdots 3$	2	1	0
Page frame	Avail	Reserved	D	A	Reserved	U/S	R/W	Present

- If page fault occurred in the page directory, then we allocate a free frame from the process pool for the page directory entry and mark the page directory entry as 'valid' using bitwise operations.

- To avoid raising another page fault subsequently (since the corresponding page is not present in the page table), an available free frame is allocated from the process pool and the page is marked as 'valid' using bitwise operations.
- Next, we initialize the corresponding page table by allocating a free frame from the process pool and mark all the entries in the page table as 'invalid' using bitwise operations.
- If page fault occurred in the page table, then we allocate a free frame from the process pool for the page table entry and mark the page table entry as 'valid' using bitwise operations.
- Return from the page fault handler.

7. page_table.C : register_pool():

This method is used to register a virtual memory pool with the page table. We use a Linked List to maintain a list of virtual memory pools. New virtual memory pools are appended to the end of the list.

8. page_table.C : free_page():

This method is used to release a frame and mark the page table entry 'invalid'. We extract the page directory index and page table index from the page number. Next, we construct the logical address and access the page table entry to obtain the frame number. We release the frame from the process memory pool and mark the page table entry 'invalid'. Finally, to erase all stale TLB entries, we flush the TLB by reloading the page table.

9. vm_pool.H : class VMPool :

In class VMPool, data structures for the virtual memory pool were declared:

- unsigned long base_address - Logical start address of virtual memory pool
- unsigned long size - Size of virtual memory pool in bytes
- unsigned long num_regions - Number of virtual memory pools
- unsigned long available_mem - Size of virtual memory available within the pool
- struct alloc_region_info * vm_regions - Pointer to virtual memory regions allocation list
- ContFramePool * frame_pool - Pointer to frame pool that provides the virtual memory pool with frames
- VMPool * vm_pool_next - Pointer to virtual memory pool linkedlist

In structure alloc_region_info, we store the information of each region in virtual memory:

- unsigned long base_address - Base address of virtual memory region
- unsigned long length - Length of virtual memory region

10. vm_pool.C : VMPool() :

This method is the constructor for class VMPool. It initializes all the data structures needed for the management of the virtual memory pool. The virtual memory pool is registered with the page table. The base address and length of the virtual memory region is stored in the structure 'alloc_region_info'. The number of virtual memory regions allocated and available virtual memory in the pool are kept track using variables 'num_regions' and 'available_mem'.

11. vm_pool.C : allocate() :

This method allocates a region of memory from the virtual memory pool. First, the allocation size requested is compared with the available virtual memory pool size. Next, we calculate the number of pages required to be allocated. We calculate the base address and length; and store details of the new virtual memory region in the data structure 'vm_regions', adding it to the allocated regions array. We recalculate the available virtual memory pool size and increment the variable 'num_regions'. On success, the method returns the virtual address of the start of the allocated region of memory. On failure, the method returns zero.

12. vm_pool.C : release() :

This method releases a region of previously allocated virtual memory. Using the start address argument, and iterating through the virtual memory regions allocated, we identify the region the start address belongs to. Next, we calculate the number of pages to be freed; and free each page iteratively. We update the virtual memory allocation information by deleting the array element which holds information of the region to be released. We recalculate the available virtual memory pool size and decrement the variable 'num_regions'.

13. vm_pool.C : is_legitimate() :

This method is used to check if the address is part of a region that is currently allocated. We check if the address argument exists between the bounds base_address and (base_address + size). On success, it returns a boolean true. On failure, it returns boolean false, if the address is not valid.

14. cont_frame_pool.C : release_frames_in_pool() :

This method is used to release all the frames of a particular frame pool. A correction was made in the logic for this method. In the previous implementation, after checking if the first frame state is HoS, we did not traverse the frames until we reached a frame that is Free or HoS. This led to 'Frame state is not HoS' errors. In the new implementation, after verifying that the first frame state is HoS, we iterate through the frames until we reach a frame that is Free or Hos. And until we reach such a frame, we mark all the frames we traverse as Free.

Outputs:

1. Testing the page table (Default test code given in kernel.C) for Fault Address = 4 MB and NACCESS = 2 KB

```
#define FAULT_ADDR (4 MB)
/* used in the code later as address referenced to cause page faults. */
//#define NACCESS ((1 MB) / 4)
#define NACCESS (2 KB)
/* NACCESS integer access (i.e. 4 bytes in each access) are made starting at address FAULT_ADDR */
```

```
#define _TEST_PAGE_TABLE_

#ifdef _TEST_PAGE_TABLE_

          /* WE TEST JUST THE PAGE TABLE */
       GeneratePageTableMemoryReferences(FAULT_ADDR, NACCESS);
```

```
rahul-611@rahul-611-VirtualBox:~/Desktop/MP4_Sources/MP4_Sources$ make run
Linux
qemu-system-x86_64 -kernel kernel.bin -serial stdio
Installed exception handler at ISR <0>
Installed interrupt handler at IRQ <0>
Frame Pool Initialized
Frame Pool Initialized
POOLS INITIALIZED!
Installed exception handler at ISR <14>
Initialized Paging System
Constructed Page Table object
Loaded page table
Enabled paging
Hello World!
EXCEPTION DISPATCHER: exc_no = <14>
EXCEPTION DISPATCHER: exc_no = <14>
handled page fault
handled page fault
DONE WRITING TO MEMORY. Now testing...
Test Passed! Congratulations!
YOU CAN SAFELY TURN OFF THE MACHINE NOW.
One second has passed
One second has passed
rahul-611@rahul-611-VirtualBox:~/Desktop/MP4_Sources/MP4_Sources$
```

Test Passed!

2. Testing the virtual memory pools (Default test code given in kernel.C)

```
//#define _TEST_PAGE_TABLE_

#ifdef _TEST_PAGE_TABLE_

           /* WE TEST JUST THE PAGE TABLE */
        GeneratePageTableMemoryReferences(FAULT_ADDR, NACCESS);

#else

           /* WE TEST JUST THE VM POOLS */

           /* -- CREATE THE VM POOLS. */

           /* ---- We define the code pool to be a 256MB segment starting at virtual address 512MB -- */
        VMPool code_pool(512 MB, 256 MB, &process_mem_pool, &pt1);

        /* ---- We define a 256MB heap that starts at 1GB in virtual memory. -- */
        VMPool heap_pool(1 GB, 256 MB, &process_mem_pool, &pt1);

        /* -- NOW THE POOLS HAVE BEEN CREATED. */

        Console::puts("VM Pools successfully created!\n");

        /* -- GENERATE MEMORY REFERENCES TO THE VM POOLS */

        Console::puts("I am starting with an extensive test\n");
        Console::puts("of the VM Pool memory allocator.\n");
        Console::puts("Please be patient...\n");
        Console::puts("Testing the memory allocation on code_pool...\n");
        GenerateVMPoolMemoryReferences(&code_pool, 50, 100);
        Console::puts("Testing the memory allocation on heap_pool...\n");
        GenerateVMPoolMemoryReferences(&heap_pool, 50, 100);

#endif

        TestPassed();
}
```

```
rahul-611@rahul-611-VirtualBox:~/Desktop/MP4_Sources/MP4_Sources$ make run
Linux
qemu-system-x86_64 -kernel kernel.bin -serial stdio
Installed exception handler at ISR <0>
Installed interrupt handler at IRQ <0>
Frame Pool Initialized
Frame Pool Initialized
POOLS INITIALIZED!
Installed exception handler at ISR <14>
Initialized Paging System
Constructed Page Table object
Loaded page table
Enabled paging
Hello World!
registered VM pool
EXCEPTION DISPATCHER: exc_no = <14>
Checked whether address is part of an allocated region.
EXCEPTION DISPATCHER: exc_no = <14>
Checked whether address is part of an allocated region.
EXCEPTION DISPATCHER: exc_no = <14>
Checked whether address is part of an allocated region.
handled page fault
handled page fault
handled page fault
Constructed VMPool object.
registered VM pool
EXCEPTION DISPATCHER: exc_no = <14>
Checked whether address is part of an allocated region.
Checked whether address is part of an allocated region.
EXCEPTION DISPATCHER: exc_no = <14>
Checked whether address is part of an allocated region.
Checked whether address is part of an allocated region.
handled page fault
handled page fault
Constructed VMPool object.
VM Pools successfully created!
```

```
I am starting with an extensive test
of the VM Pool memory allocator.
Please be patient...
Testing the memory allocation on code_pool...
Allocated region of memory.
Checked whether address is part of an allocated region.
EXCEPTION DISPATCHER: exc_no = <14>
Checked whether address is part of an allocated region.
handled page fault
Loaded page table
freed page
Released region of memory.
Allocated region of memory.
Checked whether address is part of an allocated region.
Loaded page table
freed page
Released region of memory.
Allocated region of memory.
Checked whether address is part of an allocated region.
Loaded page table
freed page
Released region of memory.
Allocated region of memory.
Checked whether address is part of an allocated region.
Loaded page table
freed page
Released region of memory.
Allocated region of memory.
Checked whether address is part of an allocated region.
Loaded page table
freed page
Released region of memory.
Allocated region of memory.
Checked whether address is part of an allocated region.
Loaded page table
freed page
```

```
Loaded page table
freed page
Loaded page table
freed page
Released region of memory.
Allocated region of memory.
Checked whether address is part of an allocated region.
Loaded page table
freed page
Loaded page table
freed page
Loaded page table
freed page
Loaded page table
freed page
Loaded page table
freed page
Released region of memory.
Allocated region of memory.
Checked whether address is part of an allocated region.
Loaded page table
freed page
Loaded page table
freed page
Loaded page table
freed page
Loaded page table
freed page
Loaded page table
freed page
Released region of memory.
Test Passed! Congratulations!
YOU CAN SAFELY TURN OFF THE MACHINE NOW.
One second has passed
One second has passed
rahul-611@rahul-611-VirtualBox:~/Desktop/MP4_Sources/MP4_Sources$
```

Test Passed!