

# MP4: Virtual Memory Management and Memory Allocation

Geetesh Challur  
UIN: 834006606  
CSCE611: Operating System

## Assigned Tasks

**Part I:** Completed.

**Part II:** Completed.

**Part III:** Completed.

## System Design

- The first 4MB of the physical memory is directly mapped, and the remaining 28MB is freely mapped.
- Page table is shifted from kernel pool to process pool
- Recursive page table lookup is implemented
- Implemented a simple virtual-memory allocator (similar to the frame pool allocator in a previous MP) and hooked it up to the new and delete operators of C++.

## Code Description

Describe your code setup here, any instruction about how to compile the code. For example, "I changed map.h, map.c, code.h, code.c for this machine problem. To compile the code use this and that in the kernel.c file."

**makefile:** : Updated the make file to use the cont.frame\_pool.o given by the instructor instead of creating it from cont.frame\_pool.C. Also updated the make clean not to delete .o files.

```
# ==== MEMORY =====

paging_low.o: paging_low.asm paging_low.H
    nasm -f elf -o paging_low.o paging_low.asm

page_table.o: page_table.C page_table.H paging_low.H
    $(GCC) $(GCC_OPTIONS) -c -o page_table.o page_table.C

# cont_frame_pool.o: cont_frame_pool.C cont_frame_pool.H
#    $(GCC) $(GCC_OPTIONS) -c -o cont_frame_pool.o cont_frame_pool.C

# ==== KERNEL MAIN FILE =====

kernel.o: kernel.C console.H simple_timer.H page_table.H
    $(GCC) $(GCC_OPTIONS) -c -o kernel.o kernel.C
```

**page\_table.H:** : INITIALIZED SYSTEM CONSTANTS ACCORDING TO CURRENT MEMORY SIZE AND PAGE TABLE LAYOUT.

- USE\_BIT To verify if the PDE or PTE is valid or not
- WRITE\_BIT To verify if the page is being written or not
- ADDR\_SHIFT\_PAGE\_DIRECTORY Used to extract the index in the page directory
- ADDR\_SHIFT\_PAGE\_TABLE\_PAGE Used to extract the index in the page table page
- PAGE\_TABLE\_PAGE\_MASK =  $j$  Also used to extract the index in the page table page
- PAGE\_DIRECTORY\_FRAME\_SIZE Size of the page directory in number of frames
- PAGE\_TABLE\_PAGE\_FRAME\_SIZE Size of the page table page in number of frames
- PAGE\_DIRECTORY\_ENTRY\_MASK Used to extract page table address from the page directory entry

```
#define USE_BIT 0x1
#define WRITE_BIT 0x2
#define ADDR_SHIFT_PAGE_DIRECTORY 22
#define ADDR_SHIFT_PAGE_TABLE_PAGE 12
#define PAGE_TABLE_PAGE_MASK 0x003ff
#define PAGE_DIRECTORY_FRAME_SIZE 0x1
#define PAGE_TABLE_PAGE_FRAME_SIZE 0x1
#define PAGE_DIRECTORY_ENTRY_MASK 0xfffff000
```

**page\_table\_.C: init\_paging** : Used to set up the global parameters for the paging subsystem.

```
void PageTable::init_paging(ContFramePool * _kernel_mem_pool,
                           ContFramePool * _process_mem_pool,
                           const unsigned long _shared_size)
{
    // Setting up the global parameters for the paging subsystem.
    kernel_mem_pool = _kernel_mem_pool;
    process_mem_pool = _process_mem_pool;
    shared_size = _shared_size;

    Console::puts("Initialized Paging System\n");
}
```

**page\_table\_.C: constructor** : In the constructor, we first get a page from the process pool and set it as the page directory. We get another page similarly and use it as the page table (1 frame would be enough to map the first 4MB).

**INITIALIZATION OF PAGE DIRECTORY:** We update the first PDE with the address of the page table page. Then, we set the rest of the PDEs to 0. Setting it to 0 also sets the use bit of the PDE to 0, indicating that the PDE is invalid.

**DIRECT MAPPING OF FIRST 4MB:** We map the first 4MB of the page table page to the first 4MB of the physical memory

```
PageTable::PageTable()
{
    // Getting a page from kernel member pool and setting it as the page directory
    page_directory = (unsigned long *) (kernel_mem_pool->get_frames(PAGE_DIRECTORY_FRAME_SIZE)*PAGE_SIZE);

    // Getting a page from kernel member pool and setting it as the page table page
    unsigned long * page_table_page = (unsigned long *) (kernel_mem_pool->get_frames(PAGE_TABLE_PAGE_FRAME_SIZE)*PAGE_SIZE);

    // INITIALIZATION OF PAGE DIRECTORY
    // Updating the first PDE with the address of the page table page
    page_directory[0] = (unsigned long) page_table_page | WRITE_BIT | USE_BIT ;

    // Setting the rest of the PDEs to 0. Setting it to 0 also sets the use bit of the PDE to 0 indicating that the PDE is invalid
    for(int i = 1; i < (ENTRIES_PER_PAGE); i++)
    {
        page_directory[i] = 0;
    }

    // DIRECT MAPPING OF FIRST 4MB
    // Mapping the first 4MB of the page table page to the first 4MB of the physical memory
    for(int i = 0; i < (ENTRIES_PER_PAGE); i++)
    {
        page_table_page[i] = (i*PAGE_SIZE) | WRITE_BIT | USE_BIT ;
    }

    Console::puts("Constructed Page Table object\n");
}
```

**page\_table\_.C: load** : Used to set the current page table to this page table and load the page directory into the CR3 register.

```
void PageTable::load()
{
    // Setting the current page table to this page table
    current_page_table = this;

    // Loading the page directory into the CR3 register
    write_cr3((unsigned long) page_directory);
    Console::puts("Loaded page table\n");
}
```

**page\_table\_.C: enable\_paging** : Used to set the paging enabled flag to 1 and enable paging by setting the 31st bit of the CR0 register to 1.

```
void PageTable::enable_paging()
{
    // Setting the paging enabled flag to 1 and enabling paging by setting the 31st bit of the CR0 register to 1
    write_cr0(read_cr0() | 0x80000000);
    paging_enabled = 1;
    Console::puts("Enabled Paging\n");
}
```

**page\_table\_.C: handle\_fault** : In this method, we first get a frame from the process pool. Then, we get the virtual address that caused the page fault, the page directory index, and the page directory of the current page table.

**FAULT HANDLING: CHECKING PAGE DIRECTORY ENTRY:** If the PDE is not valid, then we get a frame from the kernel pool to use it as a new page table page.

**FAULT HANDLING: UPDATING PAGE TABLE CORRECTLY:** If the PTE is not valid, we update the PTE by mapping the page table page to the frame address. If the PTE is valid, then the page fault is not supposed to occur (hence, assert false)

```
void PageTable::handle_fault(REGS * _r)
{
    // Getting a frame from the process pool
    unsigned long frame_addr = process_mem_pool->get_frames(PAGE_DIRECTORY_FRAME_SIZE)*PAGE_SIZE;

    // Getting the virtual address which caused the page fault
    unsigned long fault_addr = read_cr2();
    // Getting the page directory index
    unsigned long page_dir_index = fault_addr >> ADDR_SHIFT_PAGE_DIRECTORY;

    // Getting the page directory of the current page table
    unsigned long * page_directory = current_page_table->page_directory;

    // FAULT HANDLING: CHECKING PAGE DIRECTORY ENTRY
    // If the PDE is not valid, then get a frame from the kernel pool to use it as a new page table page
    if ((page_directory[page_dir_index] & USE_BIT) == 0){
        page_directory[page_dir_index] = (kernel_mem_pool->get_frames(PAGE_TABLE_PAGE_FRAME_SIZE))*PAGE_SIZE;
        page_directory[page_dir_index] |= USE_BIT | WRITE_BIT;
    }
}
```

**VM.Pool\_.H: handle\_fault** : Every VM pool has a base address, size, and is linked to a frame pool and page table. Need to keep track of the allocated regions. A data structure for these is implemented.

**VM.Pool\_.C: handle\_fault** : The functions allocate, release and is *legitimate* are rewritten and documented clearly

## Testing

In the provided test functions, page table parameters are initialized, page table instance is created, the page table is loaded as the current page table, and the paging is enabled. The `_TEST_PAGE_TABLE` variable is used to test the implementation of the page table where the process pool contains the page table.

```
EXCEPTION DISPATCHER: exc_no = <14>  
EXCEPTION DISPATCHER: exc_no = <14>  
EXCEPTION DISPATCHER: exc_no = <14>  
EXCEPTION DISPATCHER: exc_no = <14>  
EXCEPTION DISPATCHER: exc_no = <14>  
EXCEPTION DISPATCHER: exc_no = <14>  
EXCEPTION DISPATCHER: exc_no = <14>  
EXCEPTION DISPATCHER: exc_no = <14>  
EXCEPTION DISPATCHER: exc_no = <14>  
EXCEPTION DISPATCHER: exc_no = <14>  
EXCEPTION DISPATCHER: exc_no = <14>  
EXCEPTION DISPATCHER: exc_no = <14>  
EXCEPTION DISPATCHER: exc_no = <14>  
EXCEPTION DISPATCHER: exc_no = <14>  
EXCEPTION DISPATCHER: exc_no = <14>  
EXCEPTION DISPATCHER: exc_no = <14>  
EXCEPTION DISPATCHER: exc_no = <14>  
EXCEPTION DISPATCHER: exc_no = <14>  
EXCEPTION DISPATCHER: exc_no = <14>  
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
```

We can see that all the functions are being tested one after the other, and the system doesn't crash after we enable paging, indicating that the page table is initialized correctly.

I then commented the line where `_TEST_PAGE_TABLE` is defined, to test the virtual pool allocation. We can see that all the virtual pool manager tests are passed without any invalid references.

```
EXCEPTION DISPATCHER: exc_no = <14>
EXCEPTION DISPATCHER: exc_no = <14>
Loaded page table
EXCEPTION DISPATCHER: exc_no = <14>
EXCEPTION DISPATCHER: exc_no = <14>
EXCEPTION DISPATCHER: exc_no = <14>
EXCEPTION DISPATCHER: exc_no = <14>
EXCEPTION DISPATCHER: exc_no = <14>
Loaded page table
EXCEPTION DISPATCHER: exc_no = <14>
EXCEPTION DISPATCHER: exc_no = <14>
EXCEPTION DISPATCHER: exc_no = <14>
EXCEPTION DISPATCHER: exc_no = <14>
EXCEPTION DISPATCHER: exc_no = <14>
Loaded page table
EXCEPTION DISPATCHER: exc_no = <14>
EXCEPTION DISPATCHER: exc_no = <14>
EXCEPTION DISPATCHER: exc_no = <14>
EXCEPTION DISPATCHER: exc_no = <14>
EXCEPTION DISPATCHER: exc_no = <14>
Loaded page table
Test Passed! Congratulations!
YOU CAN SAFELY TURN OFF THE MACHINE NOW.
One second has passed
```