MP3: Page Manager I

Nikhil Nunna
UIN: 525008698
CSCE611: Operating System

Assigned Tasks

Main: Completed.

System Design

The goal of this program was to create a Page Manager. A page manager makes it possible for the OS to give each process its own isolated functionally contiguous memory space using something called Virtual Memory. It does this by mapping a bunch of virtual pages to the frames we created in the last programming assignment. These frames are dynamically allocated as needed. So whenever a process needs to access a frame it goes to the page table which translates the page into a physical address which is somewhere in memory and hands it back.

In our implementation, we focused on 2 things, building the structure of the page table and handling a simple page fault. Our implementation of the page table has 2 levels meaning we have a page directory which is like a page table of page tables. Then each of these sub-page tables contains pages that translate to frames and they can be dynamically allocated as needed. For our implementation, we are only handling a simple fault where a page doesn't exist. This means that our job is to allocate a frame for that page, which is then stored in the page table. Then when whatever operation causes the page fault resumes it will see that there is now a page at that address so it won't trigger a page fault.

Code Description

The code compiles and runs normally. All the grader needs to do is run 'make', './copykernel.sh' and then 'bochs -f bochsrc.bxrc'. As an alternative to this, I've created test.sh which can be run using './test.sh'. The test.sh shell script just runs make, copykernel and the bochs command for you. There are 2 changes made to the makefile to use the frame pool object file given to us, one is to exclude it from the make clean, and the other is to comment out the line that makes the object file so the one given to us doesn't get overwritten before linking.

test.sh: code block : This shell file compiles and runs the project

```
#!/bin/bash

make
./copykernel.sh
bochs -f bochsrc.bxrc
```

page_table_.c: init_paging : In this function, we just set the private variables of the page table object, this includes the kernel and process frame pool, and the size of the direct mapped memory.

```
void PageTable::init_paging(ContFramePool * _kernel_mem_pool,
                            ContFramePool * _process_mem_pool,
                            const unsigned long _shared_size)
{
   // Set private variables
   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 this function, we set up the page table structure we get a frame for the page directory and the first page table. we direct map the first 4MB of memory by setting all the addresses in the first page table to present. We then add the page table to the page directory and set the remaining page directory entries to not present.

```
   // Get page directory frame
   unsigned long page_directory_address = 4096 * kernel_mem_pool->get_frames(1);
   page_directory = (unsigned long *) page_directory_address;

   // Get page table frame
   unsigned long page_table_address = 4096 * kernel_mem_pool->get_frames(1);
   unsigned long *page_table = (unsigned long *) page_table_address;

   // Direct map first 4 MB of memory
   unsigned long address=0;
   unsigned int i;

   // Loop thru page table and set all address to present for first 4 mb
   for(i=0; i<1024; i++)
   {
      page_table[i] = address | 3;
      address = address + 4096;
   }

   // Add page table to page_directory
   page_directory[0] = (unsigned long) page_table;
```

```
    page_directory[0] = page_directory[0] | 3;

    // Loop thru page directory and set all other PDEs to empty
    for(i=1; i<1024; i++)
    {
        page_directory[i] = 0UL | 2;
    }
    Console::puts("Constructed Page Table object\n");
}
```

page_table_.c: load : In this function, we load the page directory into the cr3 register and current_page_table pointer to this which sets this page table object to the one currently being used.

```
void PageTable::load()
{
    // Load base register with address of page directory
    write_cr3((unsigned long) page_directory);
    // set current page table to this object
    current_page_table = this;
    Console::puts("Loaded page table\n");
}
```

page_table_.c: enable_paging : In this function, we enable paging by flipping a bit in the cr0 register and setting our object internal data bool "paging_enabled" to true.

```
void PageTable::enable_paging()
{
    // Flip bit in cr0 to enable paging
    write_cr0(read_cr0() | 0x80000000);
    // Enable paging in the object
    paging_enabled = true;
    Console::puts("Enabled paging\n");
}
```

page_table_.c: handle_fault : In this function, we handle any page faults that occur. I first check the byte word that is given to us along with the exception. I specifically check the last 3 leftmost bits which tell me the permissions of user level and read-write access. We are also able to see whether the page fault was caused by a protection or permission issue or the page not being present in the page table. In this MP we only need to consider the case where the page is not present in memory. From here the logic is pretty simple if the page table is not in the page directory allocate a frame for a new page and page table, put the page in the page table, and then put the page table in the page directory. The other case is simpler, the page is not in the page table, allocate a frame for the page, and put it in the page table.

```
void PageTable::handle_fault(REGS * _r)
{
    // create bools for bits to check
    bool bit_2 = false;
    bool bit_1 = false;
    bool bit_0 = false;

    // create masks for bits to shift
    int mask_2 = 1 << 2;
    int mask_1 = 1 << 1;
    int mask_0 = 1 << 0;
```

```
// get error code
int code = _r->err_code;

// check bit_1
if ((code & mask_2) == 0)
{
   bit_2 = false;
}
else
{
   bit_2 = true;
}

   // check bit_1
if ((code & mask_1) == 0)
{
   bit_1 = false;
}
else
{
   bit_1 = true;
}
   // check bit_0
if ((code & mask_0) == 0)
{
   bit_0 = false;
}
else
{
   bit_0 = true;
}


// check whether or not the entry is present in the page directory
int mask_pde_present = 1;
if (!bit_0)
{
   // Create a new page table entry, check bit 2 to know which frame pool
   unsigned long *page_table_entry;
   if (!bit_2)
   {
      unsigned long page_table_entry_address =
      4096 * kernel_mem_pool->get_frames(1);
      page_table_entry = (unsigned long *) page_table_entry_address;
   }
   else
   {
      unsigned long page_table_entry_address =
      4096 * process_mem_pool->get_frames(1);
      page_table_entry = (unsigned long *) page_table_entry_address;
   }



   // Check the address in the cr2 register
   unsigned long attempted_address = read_cr2();
```

```
// get page directory index
unsigned long binary_pde_index = (attempted_address >> 22) & 1023;

// Check whether or not the address exists in the page directory
if ((current_page_table->page_directory[binary_pde_index] &
mask_pde_present) == 0)
{

   // put the new page table page in kernel mem
   unsigned long page_table_address = 4096 * kernel_mem_pool->get_frames(1);
   unsigned long *page_table = (unsigned long *) page_table_address;

   unsigned long binary_pte_index = (attempted_address >> 12) & 1023;

   // put the pte in page table
   page_table[binary_pte_index] = (unsigned long) page_table_entry;

   // set perms
   if (!bit_2)
   {
      // Kernel only page, read and write, present
      page_table[binary_pte_index] = page_table[binary_pte_index] | 3;
   }
   else
   {
      // Kernel and user page, read and write, present
      page_table[binary_pte_index] = page_table[binary_pte_index] | 7;
   }
   // Add new page to page directory
   current_page_table->page_directory[binary_pde_index] =
   (unsigned long) page_table;
   current_page_table->page_directory[binary_pde_index] =
   current_page_table->page_directory[binary_pde_index] | 3;
}


// case where PDE is good but PTE is missing
else
{
   // get page directory index
   unsigned long binary_pde_index = (attempted_address >> 22) & 1023;

   // get page table address
   unsigned long page_table_address =
   current_page_table->page_directory[binary_pde_index];

   // set last 12 bits to 0
   page_table_address = page_table_address & 0xFFFFF000;

   // access page table
   unsigned long *page_table = (unsigned long *) page_table_address;

   // get page table entry address
```

```
            unsigned long binary_pte_index = (attempted_address >> 12) & 1023;

            // put the pte in page table
            page_table[binary_pte_index] = (unsigned long) page_table_entry;

            // set perms
            if (!bit_2)
            {
               // Kernel only page, read and write, present
               page_table[binary_pte_index] = page_table[binary_pte_index] | 3;
            }
            else
            {
               // Kernel and user page, read and write, present
               page_table[binary_pte_index] = page_table[binary_pte_index] | 7;
            }

         }

      }
     Console::puts("handled page fault\n");
}
```

Testing

I didn't add any testing since I believe that the given testing covers all of my code. If my constructor was flawed then the page table wouldn't be loaded into memory and would instead error out. I had this happen a couple of times when I was debugging my direct mapping of the first 4MB for the kernel in the constructor. The load, enable_paging, and init_paging functions are all straightforward such that they either work or they don't. There is no testing to be done as they are 2-3 line functions where if the code doesn't work the given test case would fail or it would not be possible to write the page fault handler. The page fault handler doesn't need any further testing as well as the test case given essentially writes an int array into logical memory that is 1MB long and then reads it and verifies that it is correct. This tests the full functionality of the fault handler for this MP so there is no need to test further.