MP3: Page Manager 1

Geetesh Challur
UIN: 834006606
CSCE611: Operating System

Assigned Tasks

Main: Completed.

System Design

The goal of the machine problem is to create a Page table class that stores all the necessary information required to enable and use paging.

- The first 4MB of the physical memory is directly mapped, and the remaining 28MB is freely mapped.
- In this machine problem, a two-level paging is implemented.
- As soon as a Page table instance is created, a page directory and a page table are created, which is used to map the initial 4MB of the physical memory. A single frame is used for the page directory, and a single frame is used for each page table page. Since one frame can be used to address 1000 frame entries, a single frame will be enough for a page directory. The frames from the kernel pool are used for this purpose.
- Whenever a process requests a page, we first check in the page table whether the page is allocated the physical memory. If it is allocated the physical memory, the page table is used to map the virtual address to the physical address.
- If it is not allocated to the physical memory, a page fault exception is called when it gets a new frame from the process pool and allocates it to the page, and the page table is updated accordingly. So initially, all the pages pointing above the 4MB will not have a frame allocated to them.

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
- ADD_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 =¿ 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 kernel 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 inavlid
   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 registee
   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.

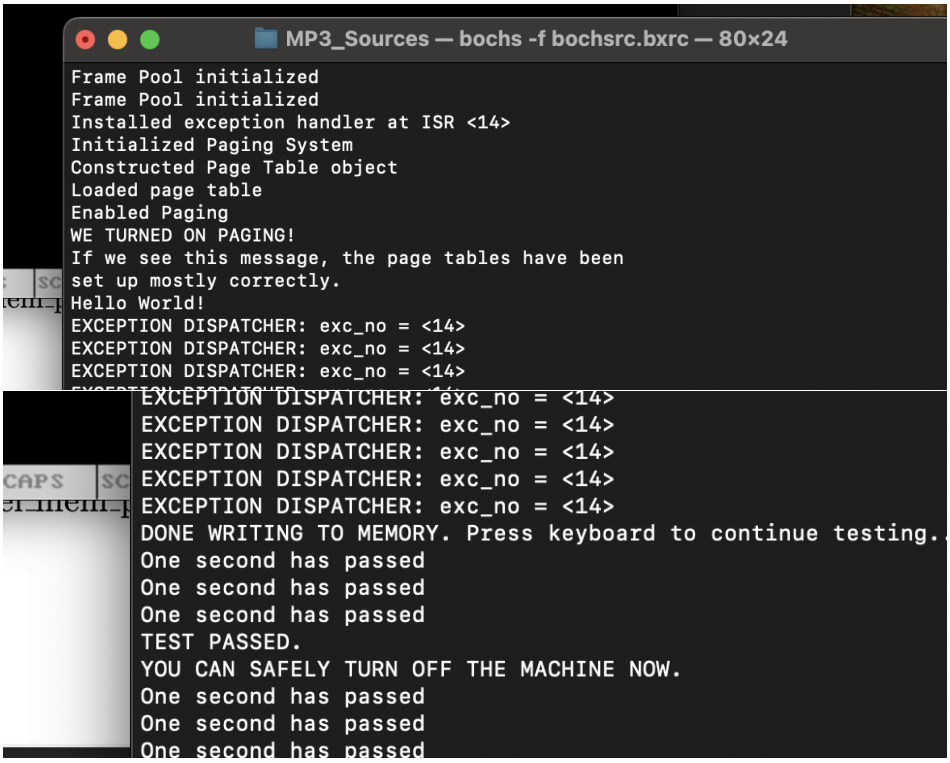
FAULT HANDING: CHECKING PAGE DIRECTORY ENTRY: If the PDE is not valid, then we get a frame from the kernel pool to use it as a new page table page.

FAULT HANDLING: UPDATING PAGE TABEL 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 HANDING: 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;
   }
```

Testing

In the provided test functions, page table parameters are initialized, page table instance is created, the page table is loaded ad the current page table, and the paging is enabled. Then the CPU issued some logical addresses which are not yet allocated to the physical frames, which causes a page fault exception, which in turn allocates a physical frame.



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. The page fault exception is being called soon after we enable paging. The tests are passed, indicating that the physical frame is being allocated correctly as soon as the page fault exception is caused.