MP3: Page Manager I

Sushant Vijay Shelar
UIN: 733001479
CSCE611: Operating System

Assigned Tasks

Main: Completed.

System Design

In this machine problem, we created a tool to manage how a computer's memory is organized, specifically for an x86 system. Think of it like managing pages in a book.

- The memory management is organized into two levels:
    1. The top level is called the "page directory," which is like the table of contents in a book.
    2. The second level is called the "page table," similar to the chapters or sections in a book.
- Each piece of information in this memory tool is like a small piece of data, and it's made up of 32 parts (bits).
- The first 10 parts help us find where to look in the page directory, like using page numbers in the table of contents.
- The next 10 parts guide us to the specific spot in the page table, similar to finding the right chapter or section.
- The last 12 parts tell us exactly where to find the information within the physical memory, much like the page number within a chapter.
- This memory management system can do two important things: it can give out new sections of memory when needed, and it can efficiently deal with any problems that might come up while using the memory.

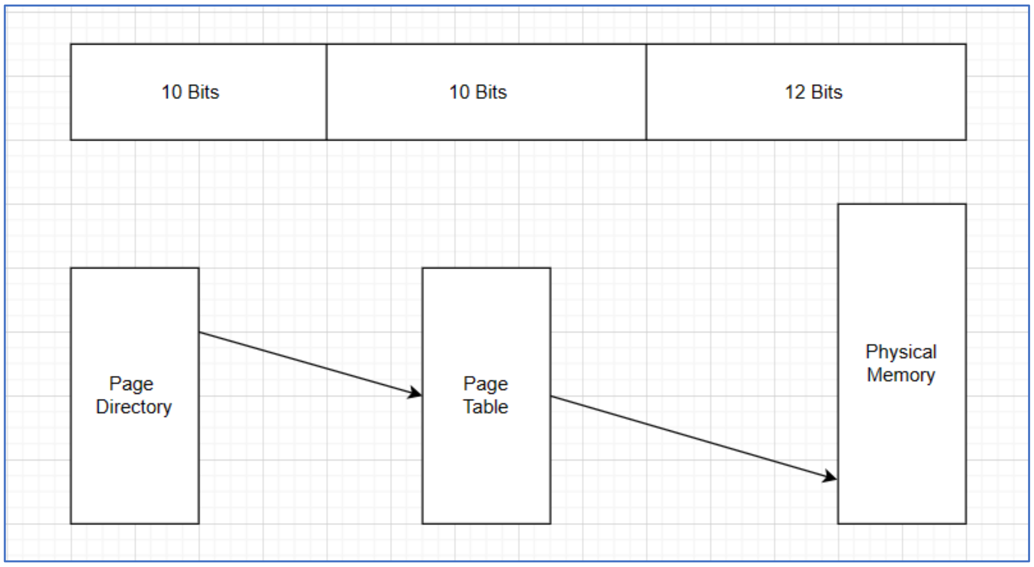


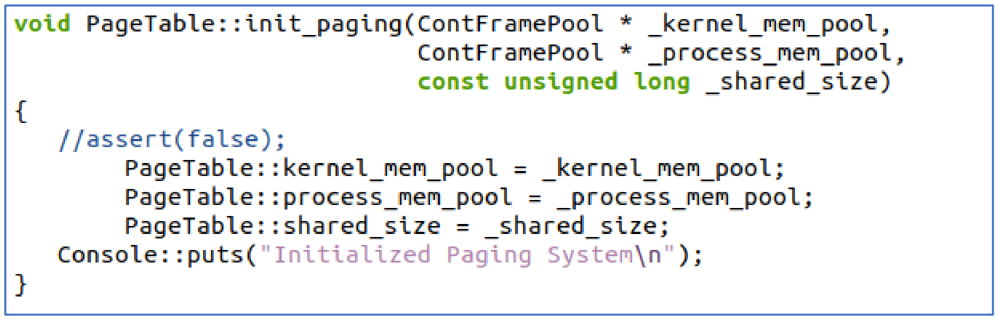
Figure 1: Page Manager

Code Description

- I made changes only to the "page table.C" file for this assignment.
- I've reused the "cont frame pool.C" and "cont frame pool.H" files from a previous submission.

init_paging:

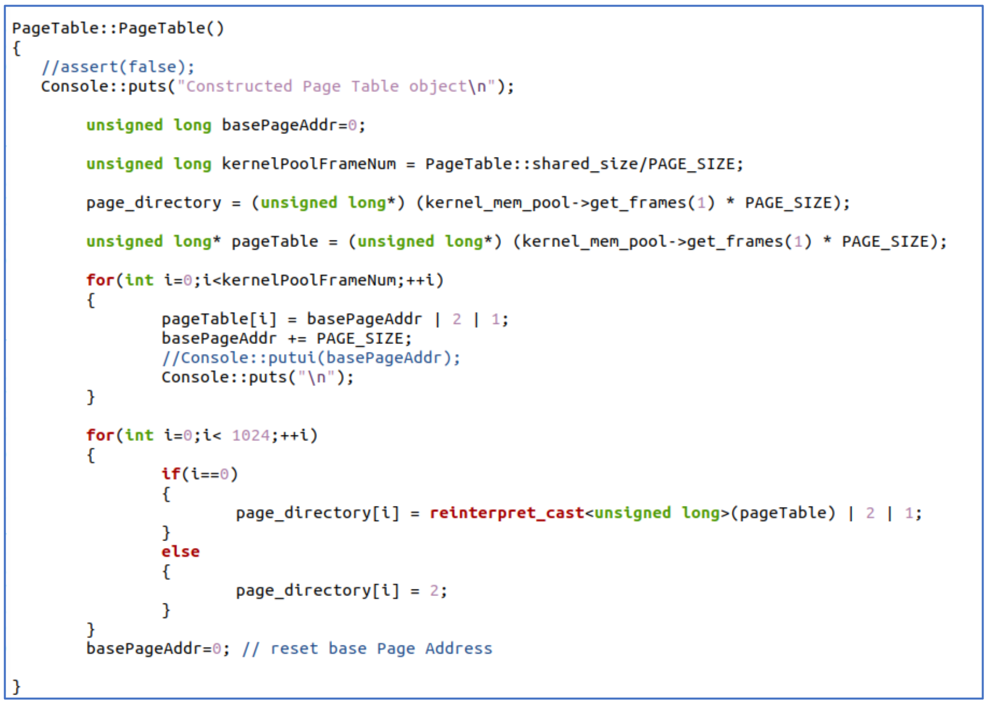
- This action sets the initial values for the static private variables within the PageTable class.



```
void PageTable::init_paging(ContFramePool * _kernel_mem_pool,
                            ContFramePool * _process_mem_pool,
                            const unsigned long _shared_size)
{
   //assert(false);
        PageTable::kernel_mem_pool = _kernel_mem_pool;
        PageTable::process_mem_pool = _process_mem_pool;
        PageTable::shared_size = _shared_size;
   Console::puts("Initialized Paging System\n");
}
```

PageTable constructor:

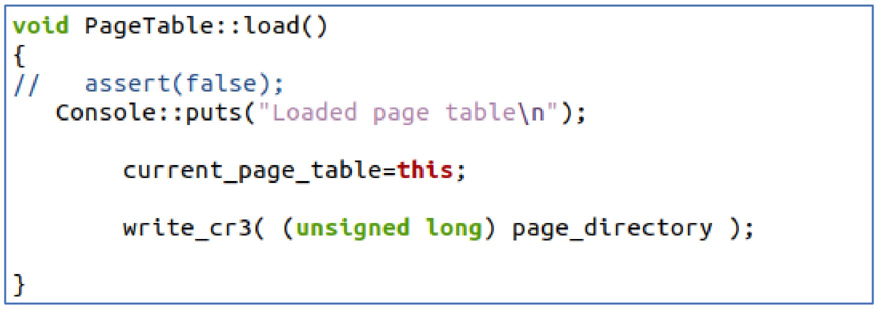
- Initially, it generates a page table specifically for directly mapped memory, which in this context is the kernel pool.
- The size of this table matches the shared memory size, which is 4MB. In this table, all pages are designated as valid and have read/write permissions.
- The remaining frames are marked as invalid and set with user-level permissions.



```
PageTable::PageTable()
{
   //assert(false);
   Console::puts("Constructed Page Table object\n");

        unsigned long basePageAddr=0;

        unsigned long kernelPoolFrameNum = PageTable::shared_size/PAGE_SIZE;

        page_directory = (unsigned long*) (kernel_mem_pool->get_frames(1) * PAGE_SIZE);

        unsigned long* pageTable = (unsigned long*) (kernel_mem_pool->get_frames(1) * PAGE_SIZE);

        for(int i=0;i<kernelPoolFrameNum;++i)
        {
                pageTable[i] = basePageAddr | 2 | 1;
                basePageAddr += PAGE_SIZE;
                //Console::putui(basePageAddr);
                Console::puts("\n");
        }

        for(int i=0;i< 1024;++i)
        {
                if(i==0)
                {
                        page_directory[i] = reinterpret_cast<unsigned long>(pageTable) | 2 | 1;
                }
                else
                {
                        page_directory[i] = 2;
                }
        }
        basePageAddr=0; // reset base Page Address

}
```

PageTable:: Load

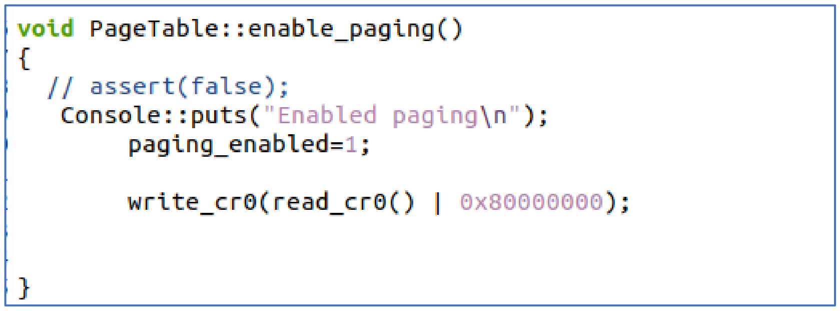
- This process involves setting the CR3 register with the address of the page directory.
- Additionally, it loads the pointer to the current Page Table object into the current page table variable.



```
void PageTable::load()
{
//   assert(false);
   Console::puts("Loaded page table\n");

        current_page_table=this;

        write_cr3( (unsigned long) page_directory );

}
```

enable_paging:

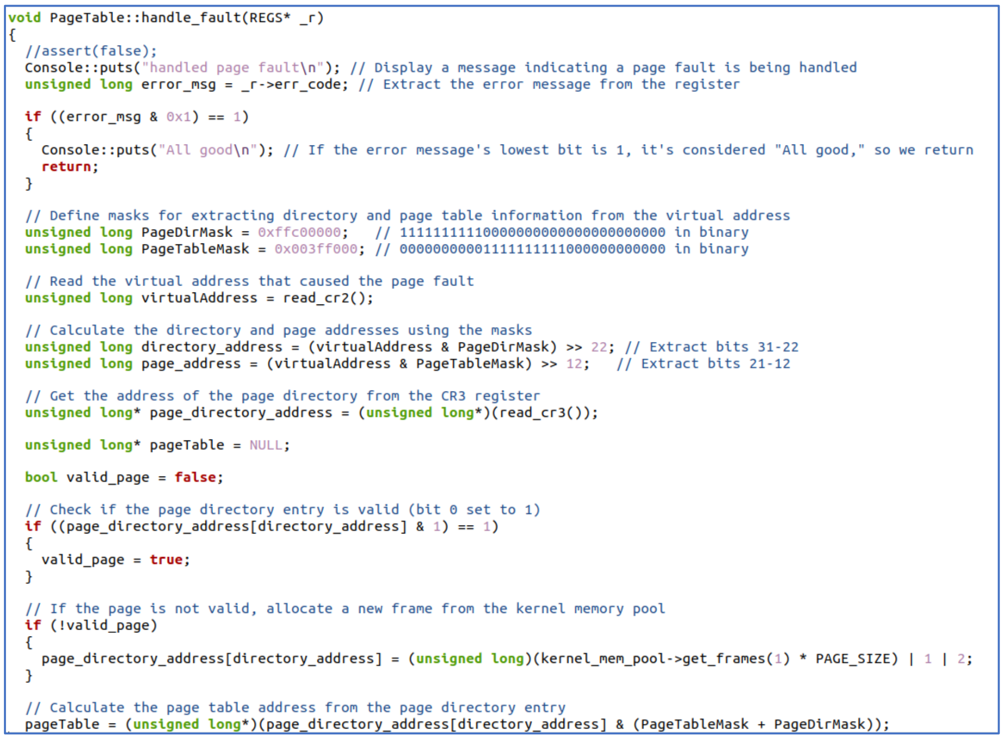
- This operation involves toggling a specific bit in the CR0 register to enable paging. The bit is set according to the provided value, and the private variable "enable paging" is subsequently set to 1.



```
void PageTable::enable_paging()
{
  // assert(false);
   Console::puts("Enabled paging\n");
        paging_enabled=1;

        write_cr0(read_cr0() | 0x80000000);


}
```

handle_fault:

- A page fault can potentially happen in either the page directory or the page table, so the initial step is to determine where the fault occurred. The address where the error happened is stored in the "cr2" register, which is then used to look up the corresponding entry in the page directory.
- If the fault is identified in the page directory, the system proceeds to allocate a new frame from the kernel pool. Subsequently, another page is allocated from the process memory pool, and this page's address is saved as an entry in the newly created page directory.
- On the other hand, if the error is located in the page table, a new page table is allocated from the process memory, and its address is updated within the relevant entry in the page directory.

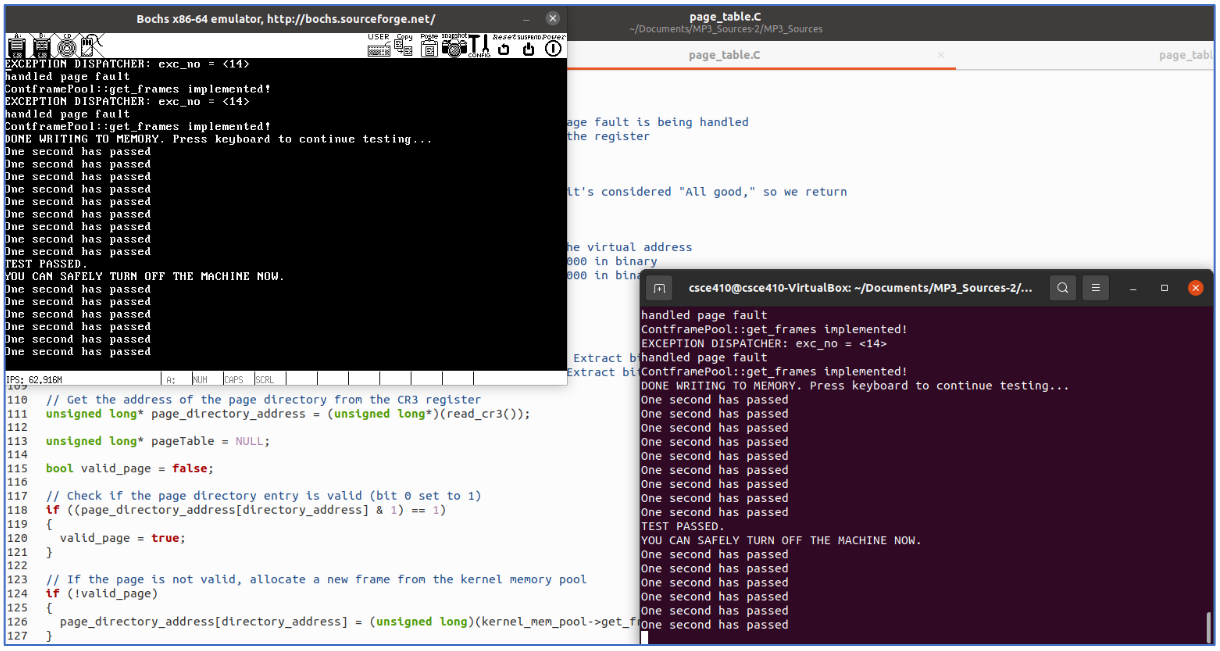


```
void PageTable::handle_fault(REGS* _r)
{
  //assert(false);
  Console::puts("handled page fault\n"); // Display a message indicating a page fault is being handled
  unsigned long error_msg = _r->err_code; // Extract the error message from the register

  if ((error_msg & 0x1) == 1)
  {
    Console::puts("All good\n"); // If the error message's lowest bit is 1, it's considered "All good," so we return
    return;
  }

  // Define masks for extracting directory and page table information from the virtual address
  unsigned long PageDirMask = 0xffc00000;   // 11111111110000000000000000000000 in binary
  unsigned long PageTableMask = 0x003ff000; // 00000000001111111111000000000000 in binary

  // Read the virtual address that caused the page fault
  unsigned long virtualAddress = read_cr2();

  // Calculate the directory and page addresses using the masks
  unsigned long directory_address = (virtualAddress & PageDirMask) >> 22; // Extract bits 31-22
  unsigned long page_address = (virtualAddress & PageTableMask) >> 12;   // Extract bits 21-12

  // Get the address of the page directory from the CR3 register
  unsigned long* page_directory_address = (unsigned long*)(read_cr3());

  unsigned long* pageTable = NULL;

  bool valid_page = false;

  // Check if the page directory entry is valid (bit 0 set to 1)
  if ((page_directory_address[directory_address] & 1) == 1)
  {
    valid_page = true;
  }

  // If the page is not valid, allocate a new frame from the kernel memory pool
  if (!valid_page)
  {
    page_directory_address[directory_address] = (unsigned long)(kernel_mem_pool->get_frames(1) * PAGE_SIZE) | 1 | 2;
  }

  // Calculate the page table address from the page directory entry
  pageTable = (unsigned long*)(page_directory_address[directory_address] & (PageTableMask + PageDirMask));
```

Testing

I used the 'make' command to compile all the files in the "mp3" folder. Then, I ran a script called 'copy kernel' to copy the bin file to the disk. After that, I executed the code using the "bochs -f bochsrc.bxrc" command.

All the test cases worked as expected, and I've attached a screenshot for reference:


Bochs x86-64 emulator, http://bochs.sourceforge.net/
USER
Copy
Paste
snapshot
CONFIG
Reset
SUSPEND
Power
IPS: 62.916M
A:
NUM
CAPS
SCRL
page_table.C
~/Documents/MP3_Sources-2/MP3_Sources
csce410@csce410-VirtualBox: ~/Documents/MP3_Sources-2/...

```
EXCEPTION DISPATCHER: exc_no = <14>
handled page fault
ContframePool::get_frames implemented!
EXCEPTION DISPATCHER: exc_no = <14>
handled page fault
ContframePool::get_frames implemented!
DONE WRITING TO MEMORY. Press keyboard to continue testing...
One second has passed
One second has passed
One second has passed
One second has passed
One second has passed
One second has passed
One second has passed
One second has passed
One second has passed
TEST PASSED.
YOU CAN SAFELY TURN OFF THE MACHINE NOW.
One second has passed
One second has passed
One second has passed
One second has passed
One second has passed
One second has passed
```

```
age fault is being handled
the register

it's considered "All good," so we return

he virtual address
000 in binary
000 in bina

 Extract bi
Extract bi
```

```
// Get the address of the page directory from the CR3 register
unsigned long* page_directory_address = (unsigned long*)(read_cr3());

unsigned long* pageTable = NULL;

bool valid_page = false;

// Check if the page directory entry is valid (bit 0 set to 1)
if ((page_directory_address[directory_address] & 1) == 1)
{
  valid_page = true;
}

// If the page is not valid, allocate a new frame from the kernel memory pool
if (!valid_page)
{
  page_directory_address[directory_address] = (unsigned long)(kernel_mem_pool->get_f
}
```

```
handled page fault
ContframePool::get_frames implemented!
EXCEPTION DISPATCHER: exc_no = <14>
handled page fault
ContframePool::get_frames implemented!
DONE WRITING TO MEMORY. Press keyboard to continue testing...
One second has passed
One second has passed
One second has passed
One second has passed
One second has passed
One second has passed
One second has passed
One second has passed
One second has passed
TEST PASSED.
YOU CAN SAFELY TURN OFF THE MACHINE NOW.
One second has passed
One second has passed
One second has passed
One second has passed
One second has passed
One second has passed
```