MP3: Page Table Management

Nachiket Umesh Naganure
UIN: 532008698
CSCE611: Operating System

Assigned Tasks

Main: Completed.

System Design

The goal of the machine problem was to design a page table manager, which will cater to demand based paging and help us map virtual memory pages to contiguous frames in physical memory. We implement a two level paging to map virtual memory to physical memory. A page fault handler is also added to deal with pages which are not valid.

1. For this task, we've developed a page table manager designed to manage page frames within the memory of an x86 machine. This page table manager utilizes a two-level structure, consisting of a primary table known as the "page directory" and a secondary table referred to as the "page table." Each entry within these tables is 32 bits in size. The first 10 bits are employed for addressing within the page directory, the subsequent 10 bits are used for addressing within the page table, and the remaining 12 bits indicate the offset within the physical frame. This paging system is equipped to allocate pages and effectively handle any potential page faults that may arise.

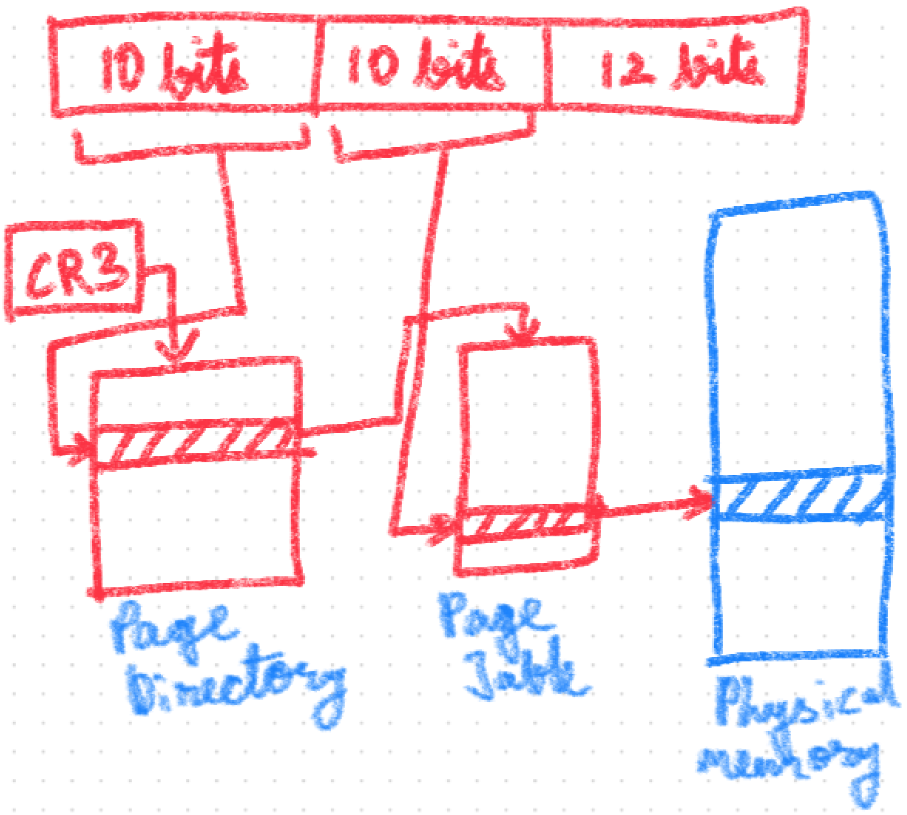
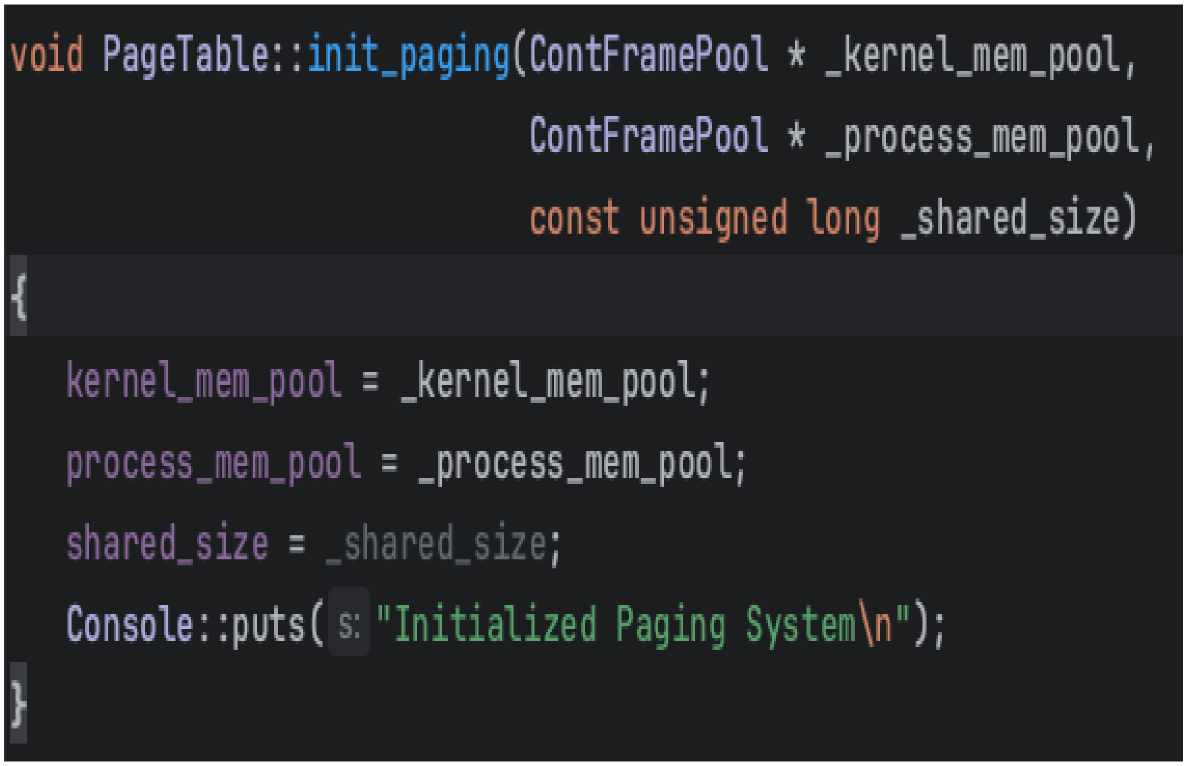


Figure 1: Memory Design Two Level Paging

Code Description

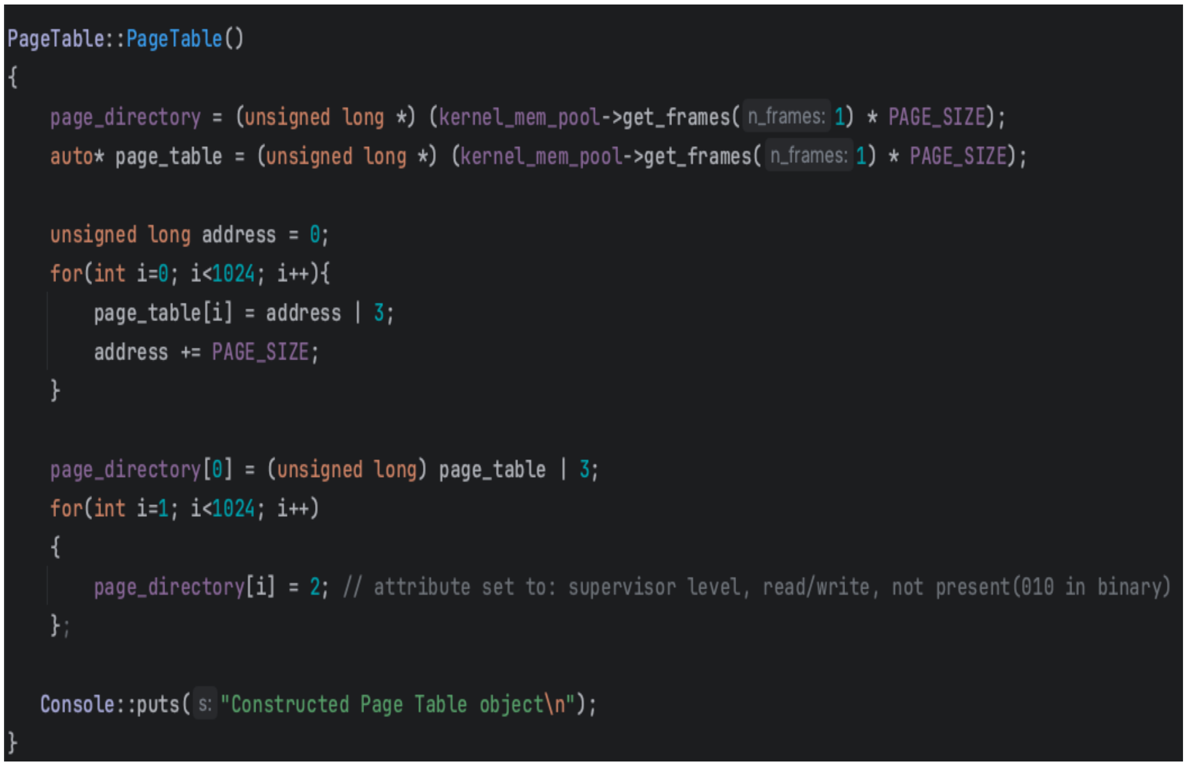
I mainly made changes in "page_table.c" and imported the "cont_frame_pool.c" , "cont_frame_pool.h" from previous machine problem. To compile the code, you need to run the make file. To run the simulator, use copykernel.sh script and run the command "bochs -f bochsrc.bxrc" to start the bochs simulator.

init_paging : Initializes the static variables of the PageTable



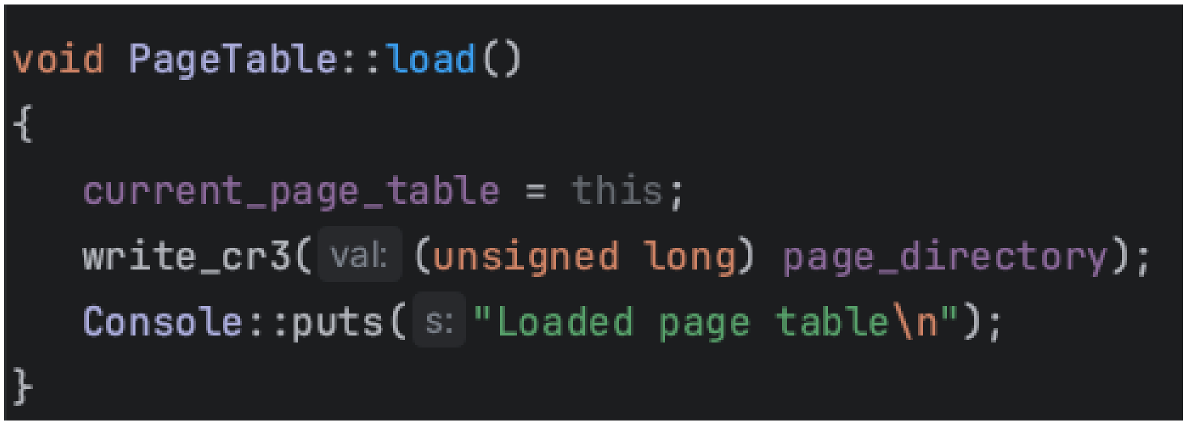
```
void PageTable::init_paging(ContFramePool * _kernel_mem_pool,
                            ContFramePool * _process_mem_pool,
                            const unsigned long _shared_size)
{
    kernel_mem_pool = _kernel_mem_pool;
    process_mem_pool = _process_mem_pool;
    shared_size = _shared_size;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

PageTable Constructor : Initially, it generates the page table specifically for directly mapped memory, which corresponds to the kernel pool. All the pages within this directly mapped table are set to the "VALID" state and configured with "READ/WRITE" permissions. In contrast, the remaining frames are designated as "INVALID" and are intended for use at the user level. And for page directory, the first entry is valid while rest of the enteries are invalid.



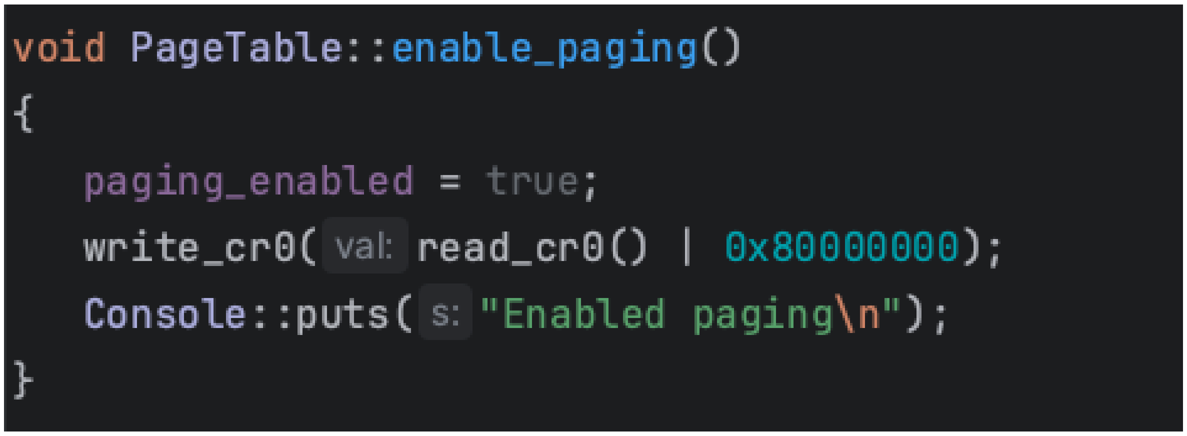
```
PageTable::PageTable()
{
    page_directory = (unsigned long *) (kernel_mem_pool->get_frames( n_frames: 1) * PAGE_SIZE);
    auto* page_table = (unsigned long *) (kernel_mem_pool->get_frames( n_frames: 1) * PAGE_SIZE);

    unsigned long address = 0;
    for(int i=0; i<1024; i++){
        page_table[i] = address | 3;
        address += PAGE_SIZE;
    }

    page_directory[0] = (unsigned long) page_table | 3;
    for(int i=1; i<1024; i++)
    {
        page_directory[i] = 2; // attribute set to: supervisor level, read/write, not present(010 in binary)
    };

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

load : This method sets the current page directory in the CR3 register and also initialises current_page_table to current page table object.



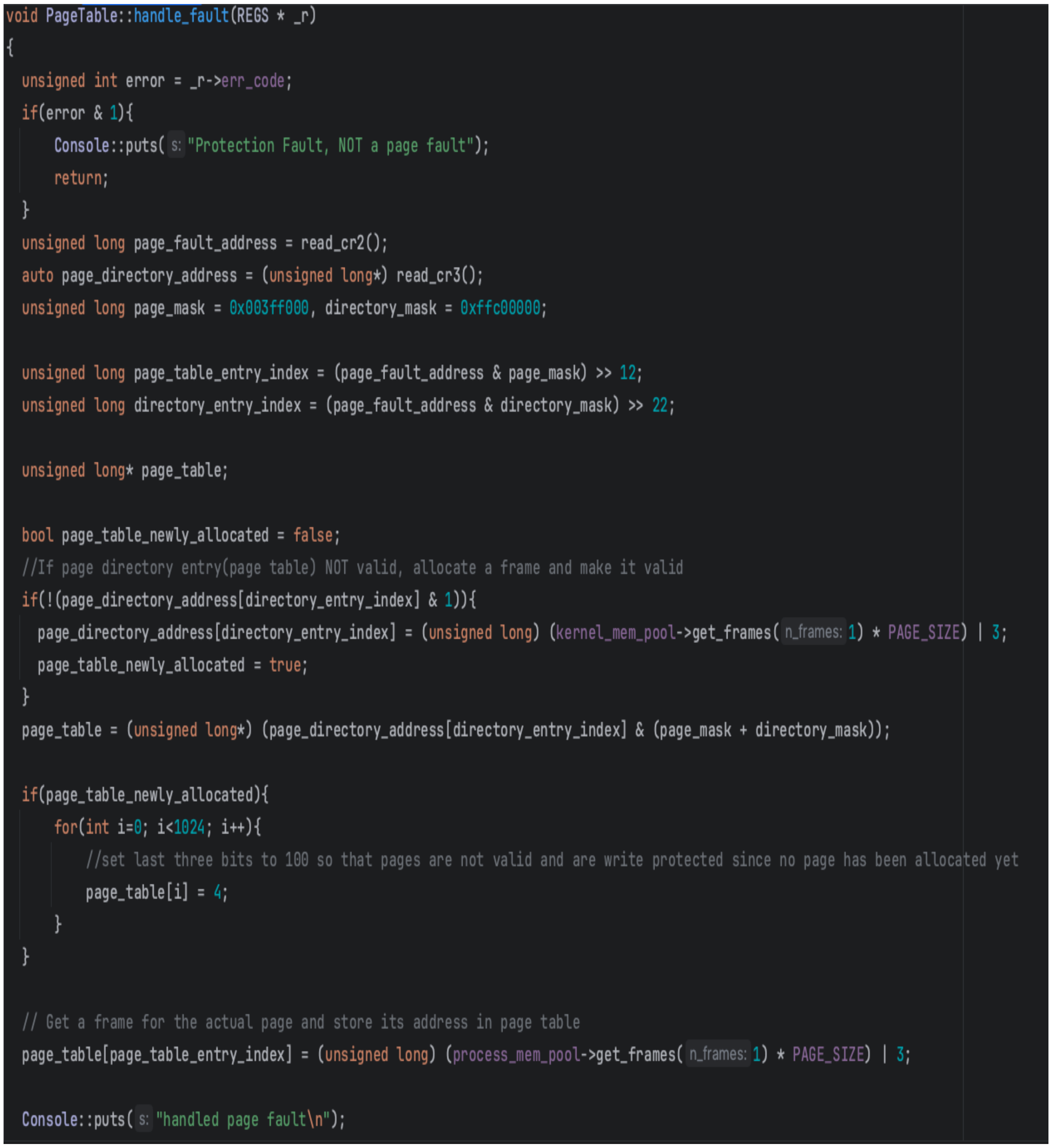
```
void PageTable::load()
{
    current_page_table = this;
    write_cr3( val: (unsigned long) page_directory);
    Console::puts( s: "Loaded page table\n");
}
```

enable_paging : This method is used set the state of CR0 register to enable paging.



```
void PageTable::enable_paging()
{
    paging_enabled = true;
    write_cr0( val: read_cr0() | 0x80000000);
    Console::puts( s: "Enabled paging\n");
}
```

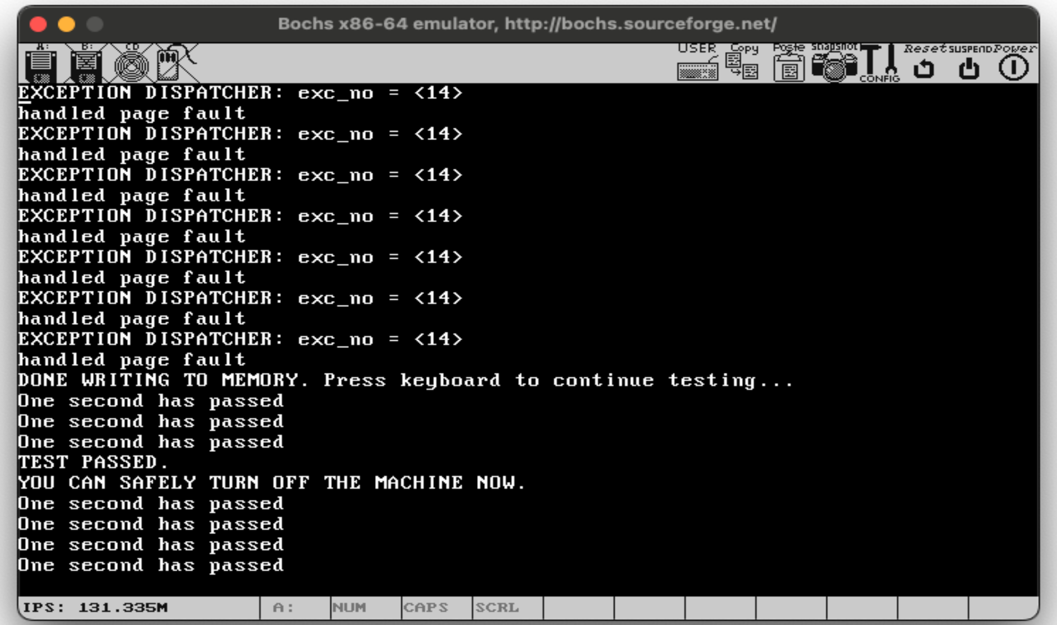
handle_fault : This method handles page fault. We check the error message from REGS struct to ensure it is indeed a page fault. We get the page fault address from CR2 Register. Based on fault address we get the index of page directory and index of page table. We check if directory entry is valid. If it is not, we allocate a frame for that entry(in other words we initialise a new page table for that directory entry to point to). Then we ensure that the page table entry is valid. If it is not, we allocate a frame for this entry(in other words get a page to which page table entry will point to)from process pool.



```
void PageTable::handle_fault(REGS * _r)
{
  unsigned int error = _r->err_code;
  if(error & 1){
      Console::puts( s: "Protection Fault, NOT a page fault");
      return;
  }
  unsigned long page_fault_address = read_cr2();
  auto page_directory_address = (unsigned long*) read_cr3();
  unsigned long page_mask = 0x003ff000, directory_mask = 0xffc00000;

  unsigned long page_table_entry_index = (page_fault_address & page_mask) >> 12;
  unsigned long directory_entry_index = (page_fault_address & directory_mask) >> 22;

  unsigned long* page_table;

  bool page_table_newly_allocated = false;
  //If page directory entry(page table) NOT valid, allocate a frame and make it valid
  if(!(page_directory_address[directory_entry_index] & 1)){
    page_directory_address[directory_entry_index] = (unsigned long) (kernel_mem_pool->get_frames( n_frames: 1) * PAGE_SIZE) | 3;
    page_table_newly_allocated = true;
  }
  page_table = (unsigned long*) (page_directory_address[directory_entry_index] & (page_mask + directory_mask));

  if(page_table_newly_allocated){
      for(int i=0; i<1024; i++){
          //set last three bits to 100 so that pages are not valid and are write protected since no page has been allocated yet
          page_table[i] = 4;
      }
  }

  // Get a frame for the actual page and store its address in page table
  page_table[page_table_entry_index] = (unsigned long) (process_mem_pool->get_frames( n_frames: 1) * PAGE_SIZE) | 3;

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

Testing

For testing, I used the given test function from kernel.C. I changed the fault address and naccess in kernel.c that it able to allocate address beyond 32 MB in virtual address. All the tests are passing, for given and additional scenarios.



```
EXCEPTION DISPATCHER: exc_no = <14>
handled page fault
EXCEPTION DISPATCHER: exc_no = <14>
handled page fault
EXCEPTION DISPATCHER: exc_no = <14>
handled page fault
EXCEPTION DISPATCHER: exc_no = <14>
handled page fault
EXCEPTION DISPATCHER: exc_no = <14>
handled page fault
EXCEPTION DISPATCHER: exc_no = <14>
handled page fault
EXCEPTION DISPATCHER: exc_no = <14>
handled page fault
DONE WRITING TO MEMORY. Press keyboard to continue testing...
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
```