MP3: Page Table Manager
Chonglin Zhang
UIN: 833003072
CSCE611: Operating System

Assigned Tasks

The goal of Machine Problem 3 is to create an initial single address space paging mechanism utilizing the page table infrastructure. In this task, we work within the confines of a 32MB memory allocation, with the initial 4MB designated for the kernel. Additionally, it's important to note that the first 4MB of memory is directly linked to the physical memory, and the first 1MB encompasses all global data. Consequently, the effective kernel initiation point is at the 1MB boundary.

System Design

We employ both a page directory and a page table to create a two-level multi-level page table, as depicted in Figure 1. Within the page table base register, a frame comprises three distinct components. The initial 10 bits correspond to the offset within the page directory, the subsequent 10 bits signify the offset within the page table, and the final 12 bits map to the offset within physical memory.

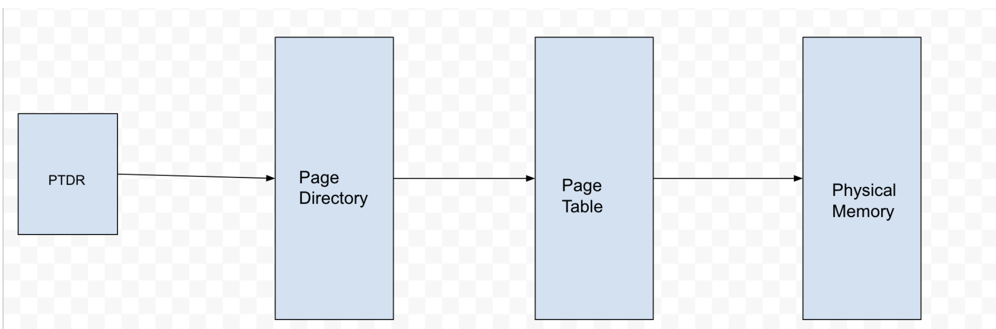

Figure 1

Code Description

page_table.c: constructor: The constructor for the page table associates addresses and adjusts bits to the present state, while marking the entries in the page directory as not present.

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

    for (int i = 0; i < shared_frames; i++) {
        unsigned long mask = i * PAGE_SIZE;
        page_table[i] = mask | 3;
        page_directory[i] = (i == 0) ? (unsigned long) page_table | 3 : mask | 2;
    }

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

page_table.c: load(): After the configuration of the page table is completed, the page directory index is deposited in the CR3 register.

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

}
```

page_table.c: enable_paging() : The method initiates paging by configuring the CR0 register.

```
void PageTable::enable_paging()
{
     paging_enabled = 1;
     write_cr0(read_cr0() | 0x80000000);
    //assert(false);
    Console::puts("Enabled paging\n");
}
```

page_table.c: handle_fault() : This method manages page faults by identifying the addresses in error and addressing them in the following manner:

1. Start by extracting the address associated with the page fault.
2. Look for an available frame from the FramePool, enabling the mapping of the address in the page table and page directory.

```
void PageTable::handle_fault(REGS * _r)
{
    if (!(_r->err_code & 1)) {
        unsigned long *dir = (unsigned long *) read_cr3();
        unsigned long dirIndex = (read_cr2() >> 22) & 0x3FF;
        unsigned long pageIndex = (read_cr2() >> 12) & 0x3FF;
        unsigned long *tbl;

        unsigned long entry = dir[dirIndex];
        if (entry & 1) {
            tbl = (unsigned long *) (entry & 0xFFFFF000);
        } else {
            dir[dirIndex] = (kernel_mem_pool->get_frames(1) * PAGE_SIZE) | 3;
            tbl = (unsigned long *) (dir[dirIndex] & 0xFFFFF000);
            unsigned int i = 0;
            while (i < shared_frames) tbl[i++] = 2;
        }

        tbl[pageIndex] = (process_mem_pool->get_frames(1) * PAGE_SIZE) | 3;
    }

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

Testing

The figure 2 and figure 3 display the kernel output.

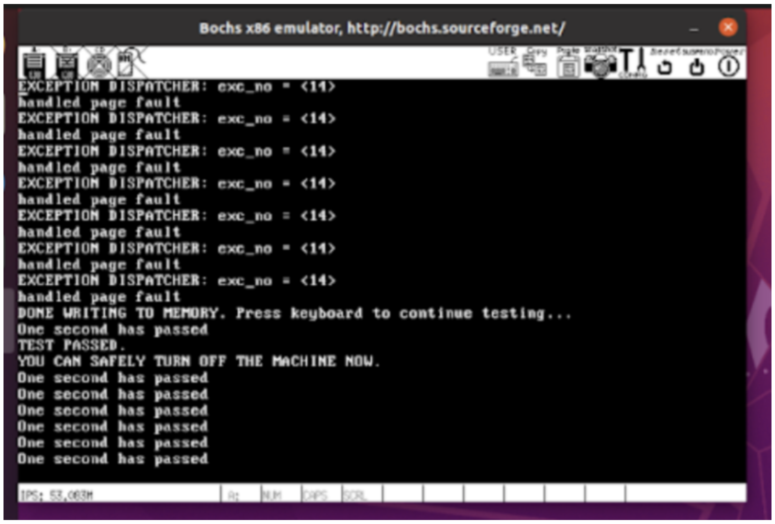

Figure 2

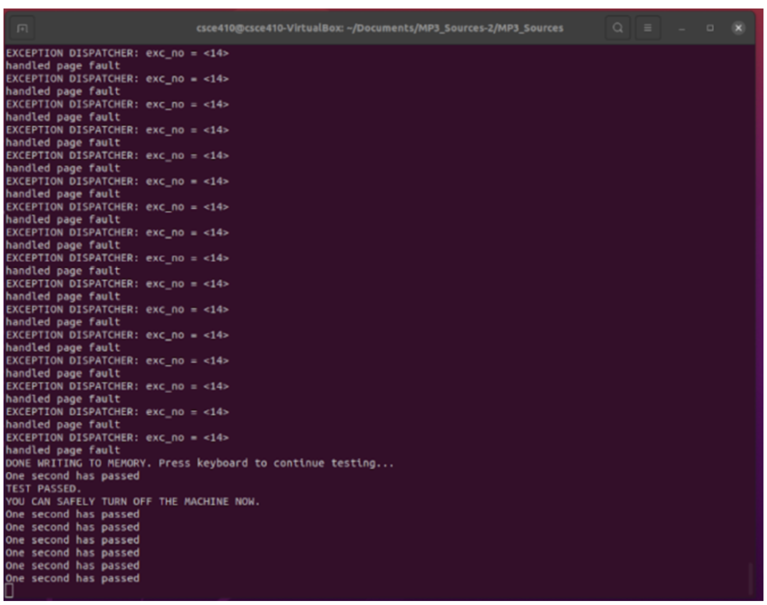

Figure 3

Test Case: Passed

Testing on the fault handler for fault addresses of both 4MB and 8MB.