CSCE 611 MP3

Jicheng Lu, 525004048

1 Introduction

In the previous Machine Problem, we have built a frame manager, which is used to manage the physical memory in the manager. The total amount of memory in the machine is 32MB with a kernel space from 0 to 4MB and a user space from 4MB to 32MB.

In this Machine Problem, we focus on the paging mechanism on the x86 architecture and set up and initialize the paging system and the page table infrastructure for a single address space. Based on previous frame manager, we start to manage the page table in our kernel, which maps the virtual memory to the physical memory. Memory within the first 4MB is direct-mapped to the physical memory, while the memory beyond the first 4MB will be mapped to non-overlapping sets of memory frames.

2 Page Table

Our paging system has two level: a page directory and multiple page table pages, which are stored in the kernel pool frames with management information for the kernel and process frame pool. Each page directory entry points to a page table, and each page table entry points to a frame in the physical memory.

For page table initialization, the first entry in the page directory is a page table that points to the first 1024 pages in memory which translates to the first 4 MBs of our kernel. This is how we implement the direct mapping and sharing of the kernel space. The rest of the pages are demand paged. In other words, virtual addresses are mapped to physical addresses when needed. This is achieved by the 'get frame' operation of frame pool .

The Constructor 'Page Table' initializes the page directory with a page table that points to the first 4 MBs as mentioned above. It marks the first page table as 'present' (i.e, the last three bits of PDE are set to 011) and all the other page tables as 'not present' (i.e, the last three bits of PDE are set to 010) in memory.

```
28
29  //
30  PageTable::PageTable()
31  {
32      page_directory = (unsigned long *)(kernel_mem_pool->get_frames(1)*FRAME_SIZE);
33      page_table = (unsigned long *)(kernel_mem_pool->get_frames(1)*FRAME_SIZE);
34
35      unsigned long address = 0;
36      unsigned int i;
37
38      //map the first 4MB memory
39      for (i = 0; i < shared_size/PAGE_SIZE; ++i){
40
41          // attribute set to: supervisor level, read/write, present(011 in binary)
42          page_table[i] = address | 3;
43          address += PAGE_SIZE;
44
45      }
46
47      // fill the first entry of the page directory
48      // attribute set to: supervisor level, read/write, present(011 in binary)
49      page_directory[0] = (unsigned long)page_table;
50      page_directory[0] = page_directory[0] | 3;
51
52
53      for(i = 1; i < shared_size/PAGE_SIZE; ++i)
54      {
55          // attribute set to: supervisor level, read/write, not present(010 in binary)
56          page_directory[i] = 0 | 2;
57      }
```

Figure 1: Consturctor Page Table

Before we construct the page table, we need to do some initialization. That is, we need to get the kernel frame pool and the process frame pool. Moreover, we need to determine the shared size of memory, which is 4MB in this case.

```
14
15 void PageTable::init_paging(ContFramePool * _kernel_mem_pool,
16                            ContFramePool * _process_mem_pool,
17                            const unsigned long _shared_size)
18 {
19     kernel_mem_pool = _kernel_mem_pool;
20     process_mem_pool = _process_mem_pool;
21     shared_size = _shared_size;
22     paging_enabled = 0;
23
24    Console::puts("Initialized Paging System\n");
25 }
```

Figure 2: Initialize paging.

Then we can load the page table as follows: we take the given page table as the current page table and then write the page directory pointer into the control register 3 (i.e., CR3).

```
66 //
67 void PageTable::load()
68 {
69     current_page_table = this;
70     write_cr3((unsigned long)page_directory);
71
72     Console::puts("Loaded page table\n");
73 }
```

Figure 3: Load page table.

Next, we enable paging by setting the 32nd bit of the control register 0 (i.e., CR0) to 1

```
77 //
78 void PageTable::enable_paging()
79 {
80     unsigned long cr_0 = read_cr0();
81     write_cr0(cr_0 | 0x80000000);  //
82
83     paging_enabled = 1;
84
85     Console::puts("Enabled paging\n");
86 }
```

Figure 4: Enable paging.

The page table $'handle - fault'$ is implemented to handle the page-fault exception of the CPU. First, the fault handler reads the desired access address from the control register 2 (i.e., CR2). The address read from CR2 is interpreted as follows: the first 10 bits is the index of the page table in the page directory, while the next 10 bits is the index of the page in the page table to be accessed. The last 12 bits is the offset of the pages.

Then the fault handler checks the error code from the register to determine whether we have a page fault. After getting these information, we can determine by checking the last bit in PDE if the page table has already been created in memory. If the page table is not in memory, a new page-table page has to be initialized and a frame is brought in from kernel pool to store the new page table. We update the page directory entry by putting the frame number of the new page table in its first 20 bits and modify the last three bits as 011. Then we get a frame from the process pool, and update the page table entry by putting this frame number in its first 20 bits and modify the last three bits as 011. On the other hand, if the page table is already present in memory, we don't need to update the page directory entry. In this case, all we need to do is finding the corresponding page table, getting a frame from the process pool, and updating the page table entry accordingly.

```
if ((page_Dir[address>>22] & 1) == 0){ // check the last bit in PDE

    Console::puts("Page is not present.\n");
    // get a free frame first
    // move the frame number towards left 12 bits, set last 3 bits 011
    // and put it into PDE
    // PDE = page_Dir[address>>22]
    page_Dir[address>>22] = (unsigned long)((kernel_mem_pool->get_frames(1)<<12) | 3);

    // get the page table address and set the last 12 bits zero
    page_Table = (unsigned long *)((page_Dir[address>>22] >> 12) << 12);

    // initialize the PTE
    for(int i=0; i<ENTRIES_PER_PAGE; ++i){
        page_Table[i] = 0;
    }

    // put the physical frame number into PTE
    // get the frame address, move towards left 12 bits, and set last 3 bits 011
    // page number = (address >> 12) & 0x3FF
    // PTE = page_Table[(address >> 12) & 0x3FF]
    page_Table[(address>>12) & 0x3FF] = (process_mem_pool->get_frames(1)<<12) | 3;
```

Figure 5: Page fault handler when page table is not present.