CSCE 611: MP3 : DESIGN DOCUMENT
AUTHOR: SRIVIDHYA BALAJI
UIN: 827007169

PROBLEM STATEMENT:

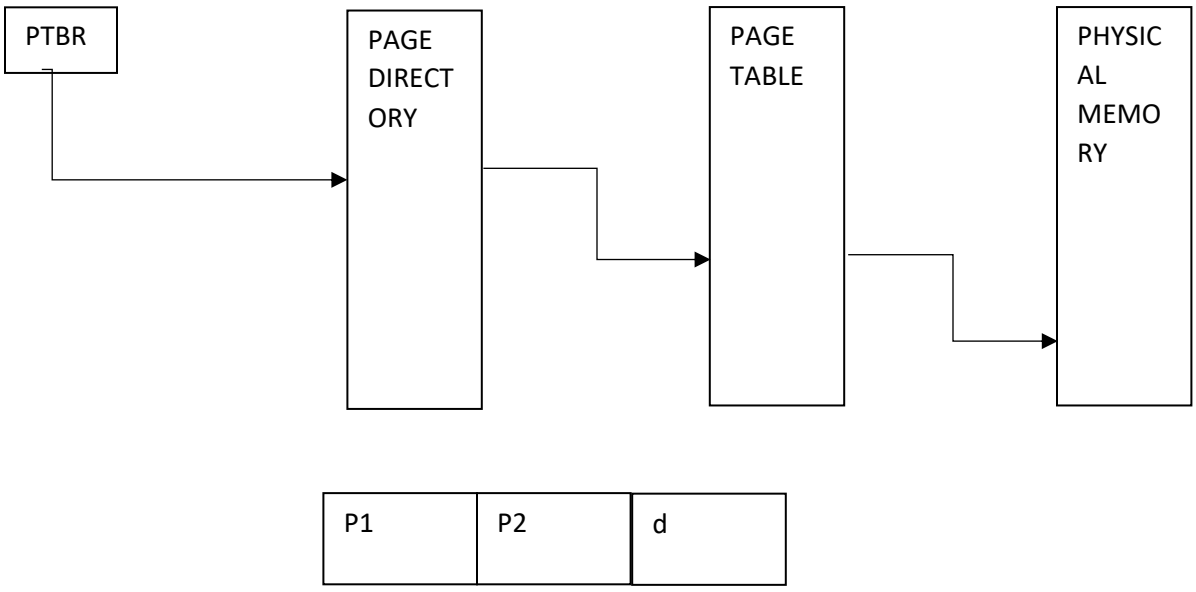
To setup and initialize the paging system and page table infrastructure

CONSTRAINTS:

1) Total Memory in the machine = 32 MB
2) First 4MB reserved for Kernel
3) Memory within the first 4MB directly mapped to physical memory
4) First 1MB contains all global data, memory mapped to devices
5) Actual Kernel code starts at 1MB

DESIGN:

The overall schematic involves a Multilevel paging in the x86 system. Here we incorporate a two level paging. Level : 1 has a page directory, Level : 2 has a Page table.



Logical address frame in PTBR. P1, represents the offset index in the page directory. 10 bits is used to represent this. P2, represents the offset index in the page table.10 bits is used to represent this. The last 12 bits represents the offset in the Physical memory. This is how using paging a logical address is mapped to a physical address.

PAGE FAULT: A page fault occurs when the physical memory cannot be mapped in either the page directory(level 1) or the page table(level 2). In this case, a frame is allocated to the corresponding address bits thus creating a corresponding logical memory for the address in the Page table system.

IMPLEMENTATION:

PART 1: DIRECT MAPPING OF SHARED MEMORY

The PageTable constructor sets up entries in the page directory and the page table. The page table entries are directly mapped to the address and its attributes are set to supervisor level, read/write and present. The remaining PageDirectory entries apart from the first one are marked as not present. To mark an entry as present the last bit is set to 1, and to mark it as not present the last bit is set to 0. Once this is done, the page_table is loaded and its address is written in the CR3 REGISTER. This stores the starting address of the Page Table directory . Its also called as the Page Table Base Register. Once the Page table is setup for the shared memory the paging is enabled.

FUNCTIONS USED:

1) **PageTable::PageTable() :** This is the constructor. The direct mapping which involves mapping the addresses and setting the bits to be present and the Page directory entries to be not present happens in the constructor.

   PSEUDO CODE:

```
Page_table = kernel_mem_pool->getframes(1)*PAGE_SIZE
address = 0
For i=0 to 1024
   Pagetable[i] = address |3
   address = address + PAGE_SIZE
PageDirectory[0] = Page_table |3
For i= 1 to 1024
   PageDirectory[i] = 2  // Marking as not present
```

2) **PageTable::load() :** Once the page_table is setup, here the current object's page directory index is extracted and stored in the PTBR ie CR3 register

   PSEUDO CODE:

```
Current_page_table = this;
Write_cr3(current_page_table->Page_Directory)
```

3) **PageTable::enable_paging() :** Next the paging is enabled by setting a specific bit in the CR0 register.

   PSEUDO CODE:

```
Write_cr0(read_cr0 | 0x80000000)
Paging_enabled = true
```

While setting up the page table, the frames are obtained using the ContifFramePool::getframes function. Here frames are obtained from the Kernel pool, thus directly mapping the complete Kernel pool.

TEST_CASES:

To test the direct mapping, the given test suite is made to run and the paging is enabled successfully and thus memory up to 4MB could be mapped and accessed.

PART 2: MEMORY BEYOND 4MB , PAGE FAULT AND HANDLING:

Memory pages above 4MB have no physical memory associated with them. So if tried to be accessed, it triggers an EXCEPTION 14 , i.e a Page Fault occurs and the Page Fault Handler needs to handle it. When a Page fault occurs the register CR2 contains the faulty address. In order to handle the fault, it is necessary to read the address for which a Page fault has occurred, get the corresponding Page_Directory and Page_Table indexes from the address bits and see if it is mapped to the Page Directory and Page Table. IIf not mapped, request for a frame and map it in the Page Directory and Page Table respectively. So once this mapping is done, next time when the address is accessed, it won't create a Page fault since its corresponding Logical address entries are mapped to the two level Paging system.

FUNCTIONS USED:

PageTable::handle_fault(REGS * _r) :

- Finds the faulty address
- Find a free frame from the Frame Pool to map the address in Page Directory and Page Table
- Page Fault handler returns
- Next time it doesn't raise a fault as the corresponding logical memory exists

PSEUDO CODE

```
address = read_cr2();
* ptr_page_dir = current_page_table->page_directory;
  obtained_page_dir_index = First 10 bits of address
  obtained_page_table_index = Next 10 bits of address

 if((ptr_page_dir[obtained_page_dir_index] & 1) == 0)
        page_table =  kernel_mem_pool->get_frames(1) *PAGE_SIZE
        ptr_page_dir[obtained_page_dir_index] = page_table |3;

  For i = 0 to 1024
        page_table[i] = 2

  if((page_table[obtained_page_table_index] &1 ) == 0)
         page_table_entry = process_mem_pool->get_frames(1) * PAGE_SIZE
        page_table[obtained_page_table_index] = page_table_entry |= 3;
```

The first if checks if the entry is not present in the Page Directory and allocates a frame . The second if checks if the entry is not present in the Page Table and allocates a frame for the Page_Table_entry.

TESTCASES:

The fault handler was evaluated for two cases.

1) Fault address = 4MB
2) Fault address = 8MB

For both these the given test suite had passed thus successfully managing the memory.