CSCE611: Operating System
MP3: Page Table Management

Name: Rahul Ravi Kadam
UIN: 834005554

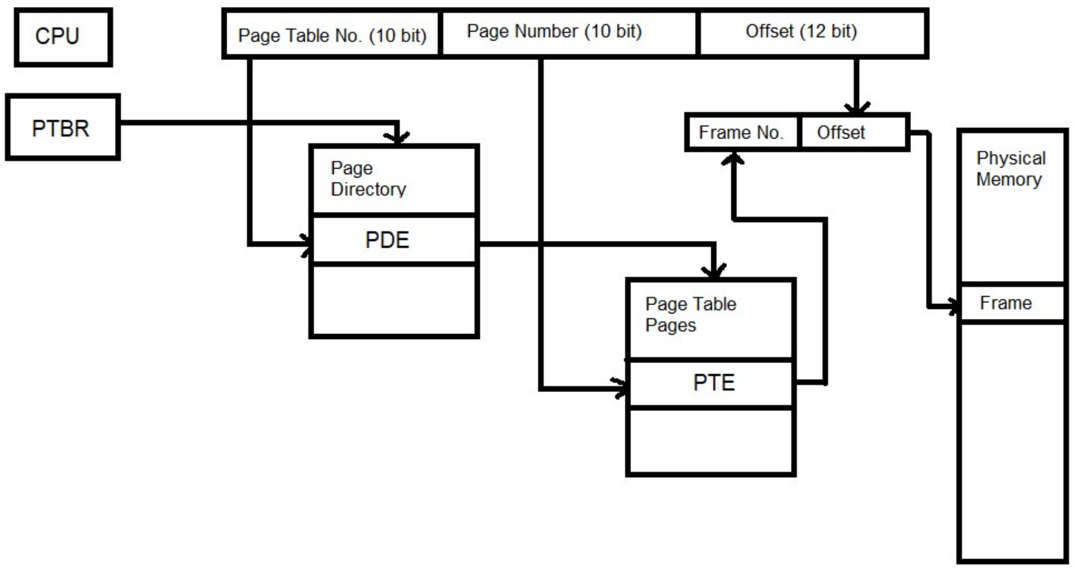
System Design:

The goal of the machine problem is to setup and initialize the paging system and the page table infrastructure on the x86 architecture for a single address space. The x86 uses a two-level hierarchical paging scheme. The 32-bit logical address is split into a 10-bit page table number, a 10-bit page number and 12-bit offset. The page table base register points to the beginning of the page directory. The page table number is used to index into the page directory to find the page directory entry. The page directory entry contains the pointer to the beginning of the page table page associated with the page table number. The page number of the address is used to index into the page table page to identify the page table entry. Finally, the offset is concatenated with the frame number that is stored in the page table entry, and the result is the physical address. In this manner, the logical address is mapped to a physical address.

The memory layout of the machine is given below:

- Total memory in the machine = 32 MB
- Memory reserved for Kernel = 4 MB (Direct-mapped to physical memory)
- Process memory pool = 28 MB (Freely-mapped to physical memory)
- Frame size = 4 KB
- Inaccessible memory region = 15 MB – 16 MB

Two-level Hierarchical Paging Diagram:



Page faults occur when physical memory cannot be mapped to either the page directory or the page table page. In such cases, a new frame must be allocated using the frame manager, and the corresponding entries in the page directory or page table pages must be updated.

The paging system includes the following functionalities:

1. Page Table Initialization
2. Loading page table
3. Handling page faults (Algorithm discussed below)

List of files modified:

1. page_table.C
2. cont_frame_pool.H (Same as in MP2)
3. cont_frame_pool.C (Same as in MP2)

Code Description:

1. page_table.C : init_paging()

This method is used to initialize the private variables for class PageTable. The global parameters for the paging subsystem are setup.

2. page_table.C : PageTable()

This method is the constructor for class PageTable. It sets up entries in the page directory and the page table. The page directory is initialized by allocating a single frame from the kernel frame pool. The first page directory entry is marked as 'valid', while the remaining entries are marked 'invalid' using bitwise operations. Next, the page table is initialized by allocating a single frame from the kernel frame pool. The page table entries for the shared portion of the memory (first 4MB which is directly-mapped) are marked 'valid' using bitwise operations.

3. page_table.C : load()

This method is used to load in a page table by storing the address of the page directory in the page table base register by writing the address into register CR3. Overall, a new page table is loaded and made the current table.

4. page_table.C : enable_paging()

This method is used to enable paging on the CPU. This is done by setting the 32nd bit to 1 and writing the value into register CR0. The 'paging_enabled' variable is also updated to 1.

5. page_table.C : handle_fault()

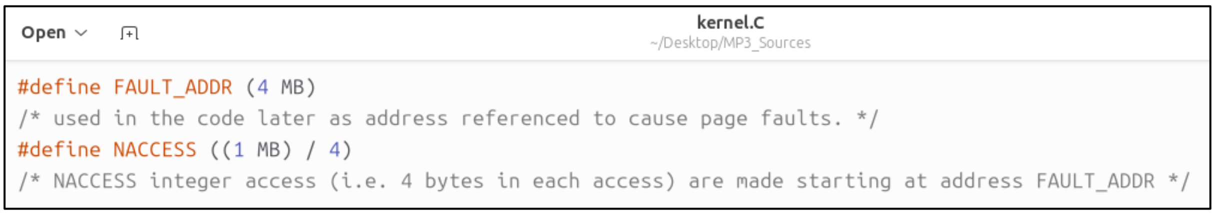
This method is used to handle page-fault exceptions of the CPU. This method will look up the appropriate entry in the page table and decide how to handle the page fault. If there is no physical memory frame associated with the page, an available frame is brought in and the page table entry is updated. If a new page table has to be initialized, a new frame is allocated for that, and the new page table page and the directory are updated accordingly. This method follows the below algorithm:

a) Obtain the page fault address by reading register CR2.
b) Obtain the page directory address by reading register CR3.
c) Calculate the page directory index (first 10 bits) and page table index (next 10 bits) from the page fault address.

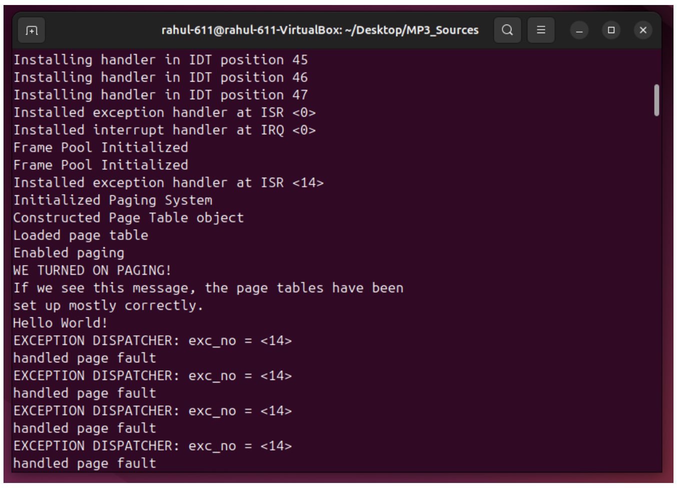
d) Read the error code of Exception 14 and check if bit 0 is not set to verify whether 'page not present' fault occurred.
e) If 'page not present' fault occurred, next we must check whether the page fault is in the page directory or the page table. This is done by checking the 'Present field' of the page directory address.
f) If page fault occurred in the page directory, then we allocate a free frame from the kernel pool for the page directory entry and mark the page directory entry as 'valid' using bitwise operations.
g) Next, we initialize the corresponding page table by allocating a free frame from the process pool and mark all the entries in the page table as 'invalid' using bitwise operations.
h) If page fault occurred in the page table, then we allocate a free frame from the process pool for the page table entry and mark the page table entry as 'valid' using bitwise operations.
i) Return from the page fault handler.
j) If a page fault occurred in the page directory previously, then another page fault is raised since the corresponding page is not present in the page table. This time, an available free frame is allocated from the process pool and the page is marked as 'valid' using bitwise operations.
k) A page fault exception is not raised the next time since the logical address exists in the paging system.

Outputs:

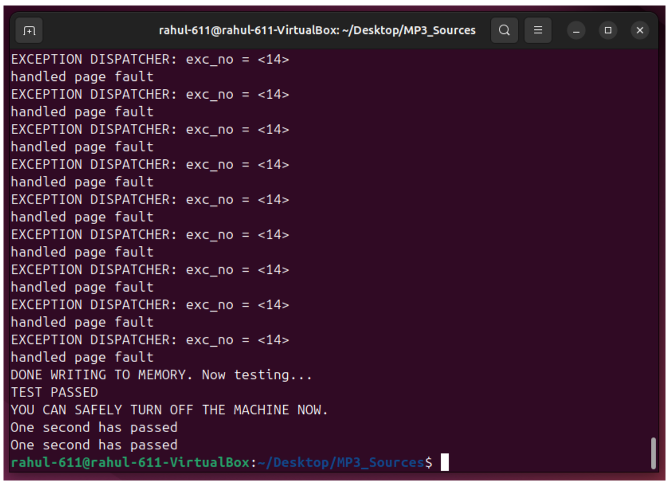
Testcase 1: Fault Address: 4 MB, Requested Memory: 1 MB



```
#define FAULT_ADDR (4 MB)
/* used in the code later as address referenced to cause page faults. */
#define NACCESS ((1 MB) / 4)
/* NACCESS integer access (i.e. 4 bytes in each access) are made starting at address FAULT_ADDR */
```



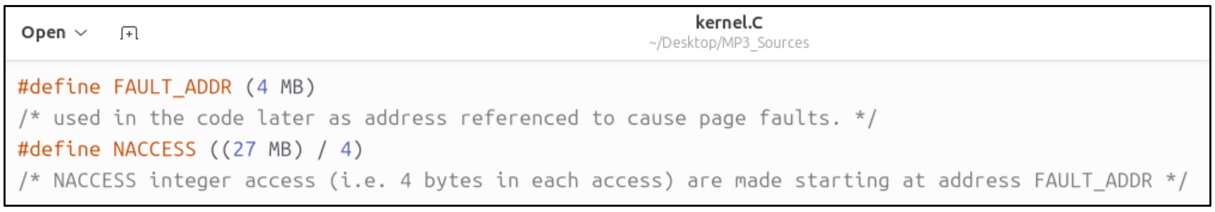
```
Installing handler in IDT position 45
Installing handler in IDT position 46
Installing handler in IDT position 47
Installed exception handler at ISR <0>
Installed interrupt handler at IRQ <0>
Frame Pool Initialized
Frame Pool Initialized
Installed exception handler at ISR <14>
Initialized Paging System
Constructed Page Table object
Loaded page table
Enabled paging
WE TURNED ON PAGING!
If we see this message, the page tables have been
set up mostly correctly.
Hello World!
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
```



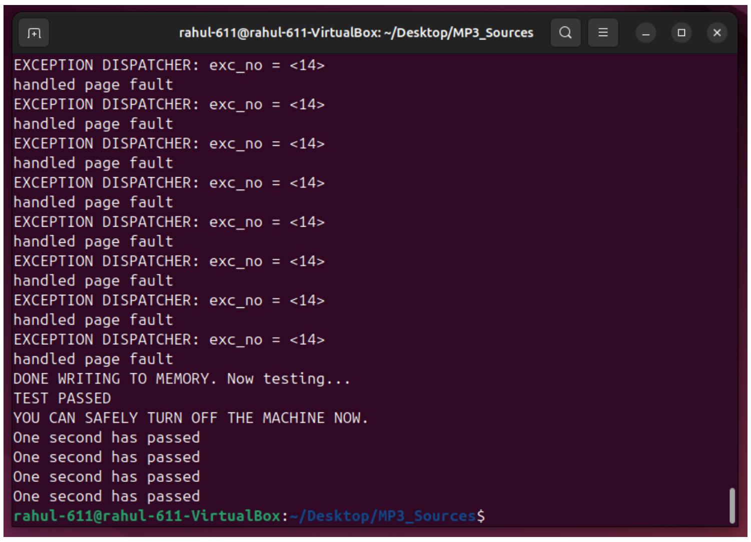
```
EXCEPTION DISPATCHER: exc_no = <14>
handled page fault
EXCEPTION DISPATCHER: exc_no = <14>
handled page fault
EXCEPTION DISPATCHER: exc_no = <14>
handled page fault
EXCEPTION DISPATCHER: exc_no = <14>
handled page fault
EXCEPTION DISPATCHER: exc_no = <14>
handled page fault
EXCEPTION DISPATCHER: 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. Now testing...
TEST PASSED
YOU CAN SAFELY TURN OFF THE MACHINE NOW.
One second has passed
One second has passed
rahul-611@rahul-611-VirtualBox:~/Desktop/MP3_Sources$
```

All page faults are handled; data written and read are read are equivalent.
Testcase passed!

Testcase 2: Fault Address: 4 MB, Requested Memory: 27 MB



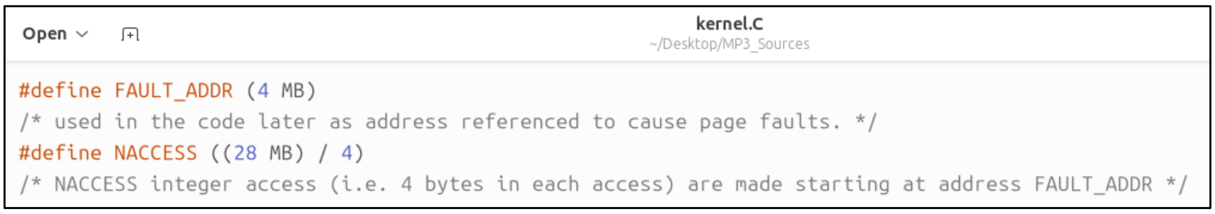
```
#define FAULT_ADDR (4 MB)
/* used in the code later as address referenced to cause page faults. */
#define NACCESS ((27 MB) / 4)
/* NACCESS integer access (i.e. 4 bytes in each access) are made starting at address FAULT_ADDR */
```



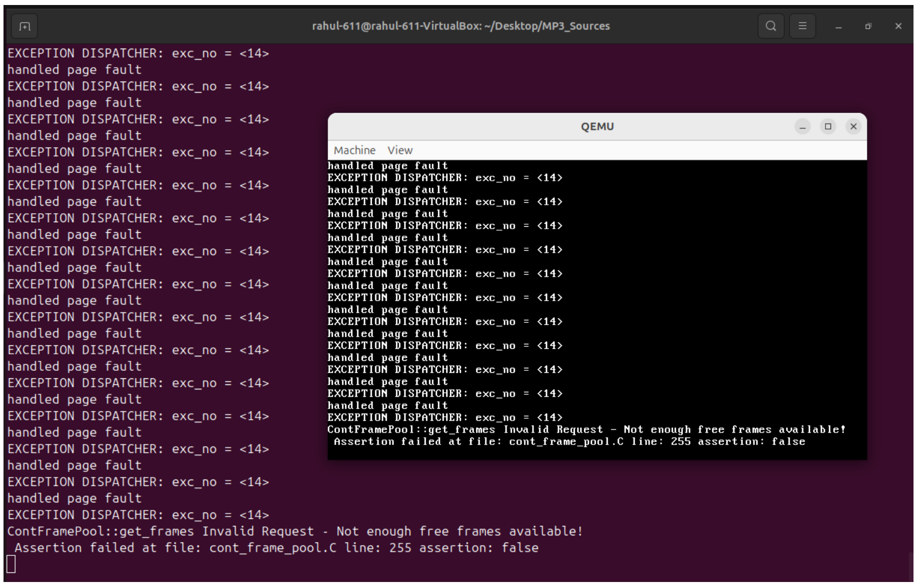
```
EXCEPTION DISPATCHER: exc_no = <14>
handled page fault
EXCEPTION DISPATCHER: exc_no = <14>
handled page fault
EXCEPTION DISPATCHER: exc_no = <14>
handled page fault
EXCEPTION DISPATCHER: exc_no = <14>
handled page fault
EXCEPTION DISPATCHER: exc_no = <14>
handled page fault
EXCEPTION DISPATCHER: exc_no = <14>
handled page fault
EXCEPTION DISPATCHER: exc_no = <14>
handled page fault
EXCEPTION DISPATCHER: exc_no = <14>
handled page fault
DONE WRITING TO MEMORY. Now testing...
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
rahul-611@rahul-611-VirtualBox:~/Desktop/MP3_Sources$
```

All page faults are handled; data written and read are equivalent.
Testcase passed!

Testcase 3: Fault Address: 4 MB, Requested Memory: 28 MB



```
#define FAULT_ADDR (4 MB)
/* used in the code later as address referenced to cause page faults. */
#define NACCESS ((28 MB) / 4)
/* NACCESS integer access (i.e. 4 bytes in each access) are made starting at address FAULT_ADDR */
```

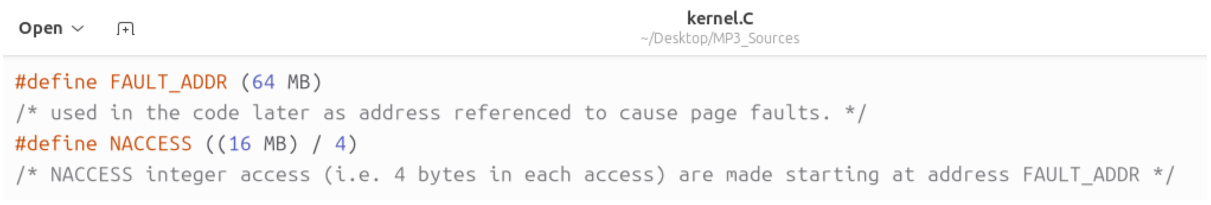


```
EXCEPTION DISPATCHER: exc_no = <14>
handled page fault
EXCEPTION DISPATCHER: exc_no = <14>
handled page fault
EXCEPTION DISPATCHER: exc_no = <14>
handled page fault
EXCEPTION DISPATCHER: exc_no = <14>
handled page fault
EXCEPTION DISPATCHER: exc_no = <14>
handled page fault
EXCEPTION DISPATCHER: exc_no = <14>
handled page fault
EXCEPTION DISPATCHER: exc_no = <14>
handled page fault
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
EXCEPTION DISPATCHER: exc_no = <14>
ContFramePool::get_frames Invalid Request - Not enough free frames available!
 Assertion failed at file: cont_frame_pool.C line: 255 assertion: false
```

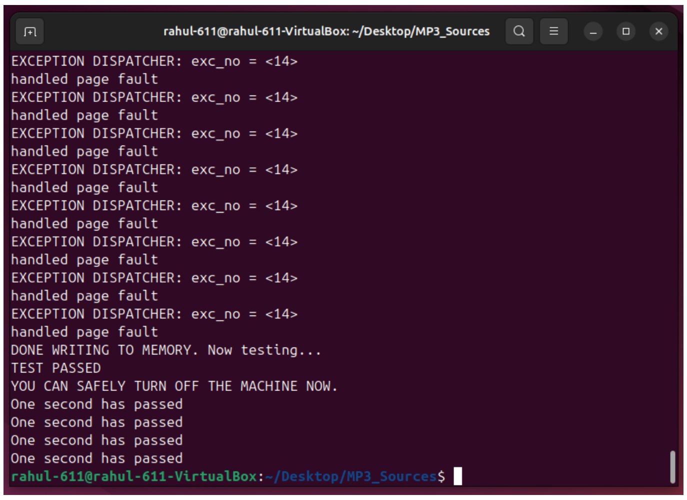
```
handled page fault
EXCEPTION DISPATCHER: exc_no = <14>
handled page fault
EXCEPTION DISPATCHER: exc_no = <14>
handled page fault
EXCEPTION DISPATCHER: exc_no = <14>
handled page fault
EXCEPTION DISPATCHER: exc_no = <14>
handled page fault
EXCEPTION DISPATCHER: exc_no = <14>
handled page fault
EXCEPTION DISPATCHER: exc_no = <14>
handled page fault
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>
ContFramePool::get_frames Invalid Request - Not enough free frames available!
 Assertion failed at file: cont_frame_pool.C line: 255 assertion: false
```

Not enough free frames are available to allocate.
Testcase passed!

Testcase 4: Fault Address: 64 MB, Requested Memory: 16 MB



```
#define FAULT_ADDR (64 MB)
/* used in the code later as address referenced to cause page faults. */
#define NACCESS ((16 MB) / 4)
/* NACCESS integer access (i.e. 4 bytes in each access) are made starting at address FAULT_ADDR */
```



```
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
EXCEPTION DISPATCHER: exc_no = <14>
handled page fault
DONE WRITING TO MEMORY. Now testing...
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
rahul-611@rahul-611-VirtualBox:~/Desktop/MP3_Sources$
```

All page faults are handled; data written and read are equivalent.
Testcase passed!