CSCE 611

Machine Problem 3: Page Table Management

Submitted By – Vaibhav Rawat
UIN: 626008171

Main task in this assignment was to implement a demand paging based virtual memory system for kernel. Design for the page table management involves two levels: a page directory and a page table. Page directory stores the Page Directory Entries (PDE) which point to addresses of page tables and page table stores Page Table Entries (PTE) which point to frames in physical memory. For allocation of these frames, getframes API implemented for contiguous frame pools in last machine problem are used. Both Page Directory and Page Table are of page size of 4 KB, and each PDE and PTE consume 4 bytes each. So, they each store 1024 PDE and PTE entries.

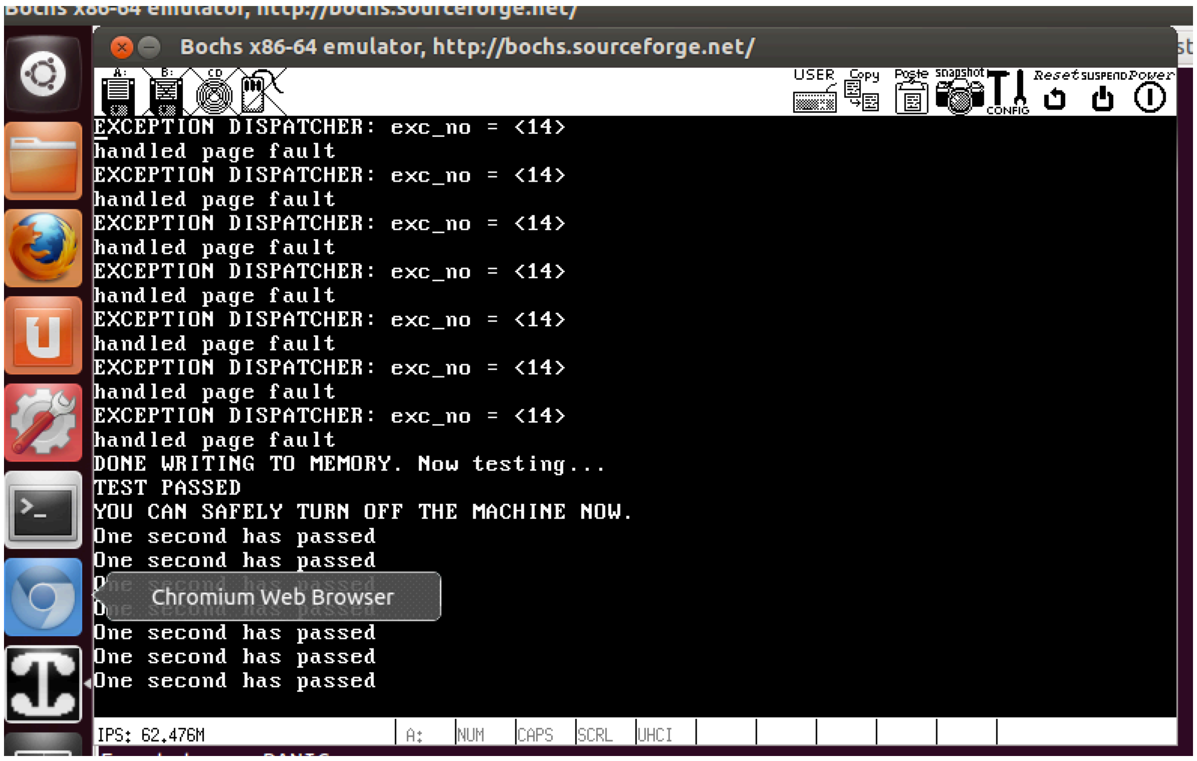
PageTable() contructor - initializes by allocating a free frame for page directory and a free frame for page table. The page table is directly-mapped to first 4 MB of physical memory that is the first 1024 frames in kernel (processes address space will share this kernel memory), so all the entries in page table are pointing to physical frames. First 20 bits of page table entries contain the address for the frames with rest of the control bits set to specify supervisor mode with write control. For the page directory, there is only one entry pointing to an actual page table with its first 20 bits and rest of bits specifying supervisor mode with write control. For all the other entries in page directory, their control bits are set to specify invalid entry that is not pointing to page table.

Load function is used to set the current page table. So, when enable paging is called it writes the page directory of current page table to CR3 register and sets the paging bit in CR0. This tells the CPU to enable paging. [Paging is enabled after CPU has already started (with paging disabled)].

So initially before any page fault has happened, page directory has a single entry to page table directly mapped to first 4 MB of kernel and rest of the entries are to be freely mapped once a page fault occurs. So once a page fault happens, handle_fault function is invoked. If exception is 14 which corresponds to page fault, I extract the fault address which is stored in CR2 register. Extract the first 10 bits of fault address which correspond to dir_entry number in page directory. Similarly I extract the next 10 bits which correspond to table_entry number in page table. Last 12 bits of fault address point to offset in frame.

Next I check to see if dir_entry has its not present bit set. If so, dir_entry does not point to any page table. Using kernel pool's getframes, a frame is allocated for page table, dir_entry is made to point to address page table with its control bits set to supervisor and write. For newly created page table we initialize all the entries to be invalid and update the entry requested in virtual address. All the invalid entries will point to a frame only on demand of that entry.

Next, we fetch the page table address using dir_entry of page directory and look for table_entry in page table. If table_entry bit is not_present then we get a new free physical frame using process pool's getframes, table_entry is made to point to the free frame. I successfully verified my logic with different memory requests. Here is a sample snapshot from one of the runs.



```
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
One second has passed
One second has passed
One second has passed
```