Lab Assignment 3: Printing Page Table

Name :- Syed Abrar Roll Number:- CS22BTECH11058

Solution Methodology:-

To implement the vmprint() function as per the given requirements, I followed these steps:

Definition: I added the definition of the vmprint() function in defs.h so that it can be called from other files, especially from exec.c.

Implementing _pteprint() Function: I implemented a function _pteprint() to recursively traverse the page table and print its contents. This function iterates over each PTE, checks if it's valid, prints its information, and recursively calls itself if the PTE points to a lower-level page table.

Implementing vmprint() Function: I implemented the main vmprint() function. This function prints the address of the top-level page table and then calls the _pteprint() function to print its contents recursively.

Integrating with exec.c: Then, I added the vmprint() function into the exec.c file just before the return statement, as instructed.

Testing: I tested the implementation thoroughly to make sure I will get the desired output,

This way I ensured that the vmprint() function was correctly implemented into the xv6 operating system, to print the contents of a process's page table as required.

Question 1:-

```
hart 1 starting
hart 2 starting
page table 0x0000000087f6c000
.. 0: pte 0x0000000021fda001 pa 0x0000000087f68000
.. .. 0: pte 0x0000000021fd9c01 pa 0x0000000087f67000
.. .. .. 0: pte 0x0000000021fda41b pa 0x0000000087f69000
.. .. .. 1: pte 0x0000000021fd9817 pa 0x0000000087f66000
.. .. .. 2: pte 0x0000000021fd9407 pa 0x0000000087f65000
.. .. .. 3: pte 0x0000000021fd9017 pa 0x0000000087f64000
.. 255: pte 0x0000000021fdac01 pa 0x0000000087f6b000
.. .. 511: pte 0x0000000021fda801 pa 0x0000000087f6a000
.. .. .. 510: pte 0x0000000021fdd007 pa 0x0000000087f74000
.. .. .. 511: pte 0x0000000020001c0b pa 0x0000000080007000
init: starting sh
```

Question 2:-

```
init: starting sh
page table 0x0000000087f5f000
.. 0: pte 0x0000000021fd6c01 pa 0x0000000087f5b000
.. .. 0: pte 0x0000000021fd6801 pa 0x0000000087f5a000
.. .. .. 0: pte 0x0000000021fd701b pa 0x0000000087f5c000
.. .. .. 1: pte 0x0000000021fd641b pa 0x0000000087f59000
.. .. .. 2: pte 0x0000000021fd6017 pa 0x0000000087f58000
.. .. .. 3: pte 0x0000000021fd5c07 pa 0x0000000087f57000
.. .. .. 4: pte 0x0000000021fd5817 pa 0x0000000087f56000
.. 255: pte 0x0000000021fd7801 pa 0x0000000087f5e000
.. .. 511: pte 0x0000000021fd7401 pa 0x0000000087f5d000
.. .. .. 510: pte 0x0000000021fdb407 pa 0x0000000087f6d000
.. .. .. 511: pte 0x0000000020001c0b pa 0x0000000080007000
```

Question 3:-

```
$ sleep 1
page table 0x0000000087f42000
.. 0: pte 0x0000000021fcf801 pa 0x0000000087f3e000
.. .. 0: pte 0x0000000021fcf401 pa 0x0000000087f3d000
.. .. .. 0: pte 0x0000000021fcfc1b pa 0x0000000087f3f000
.. .. .. 1: pte 0x0000000021fcf017 pa 0x0000000087f3c000
.. .. .. 2: pte 0x0000000021fcec07 pa 0x0000000087f3b000
.. .. .. 3: pte 0x0000000021fce817 pa 0x0000000087f3a000
.. 255: pte 0x0000000021fd0401 pa 0x0000000087f41000
.. .. 511: pte 0x0000000021fd0001 pa 0x0000000087f40000
.. .. .. 510: pte 0x0000000021fd8007 pa 0x0000000087f60000
.. .. .. 511: pte 0x0000000020001c0b pa 0x0000000080007000
$
```

Question 4:-

In xv6, every page frame measures 4KB, which is like having 512 small sections in each page. The virtual address space, where data lives in the computer's memory, can only be as big as 39 tiny parts. However, when it comes to figuring out where things are physically located in the computer's memory, we use a much larger space, made up of 56 parts. This big space includes 44 parts coming from the Page Table Entry (PTE) and another 12 parts from the offset, which is like the position within a section.

Question 5:-

In xv6, when figuring out where things are stored in memory, we use 39 tiny pieces for virtual addresses. Among these, the last 12 bits are used to find the exact spot within a section. The remaining 27 bits are divided into three groups of 9 bits each. The first 9 bits help us find the outer page table, the next 9 bits locate the first inner page table, and the last 9 bits guide us to the innermost page table.

Question 6:-

In xv6, to know if a Page Table Entry (PTE) is valid, we check a special bit called PTE_V using a method called bitwise AND. Also, to see if a user process is involved, we look at another bit called PTE_U, which is the 5th smallest bit. Furthermore, to tell if a PTE points to a page table inside the process or if it's part of the process itself, we check if it has any permissions set for reading, writing, or executing. If all these permissions are zero, it points to an inner page table; otherwise, it's part of the process.

Question 7:-

	init process	**sh process**	**sleep process**	**Remarks**
No. of page frames consumed by the page table	**5**	**5**	**5**	**All of them are equal**

Internal fragmentation in the page table(s) in bytes	**20400**	**20392**	**20400**	**Equal for init and sleep process**
No. of page frames allocated for the process	**6**	**7**	**6**	**Equal for init amd sleep processes**
No. of page frames allocated for TEXT segment of the process and their physical addresses	**1** **0x87f69000**	**2** **0x87f69000**	**1** **0x87f3f000**	**The same address for init and sh** **But different page frames**
No. of page frames allocated for Data/Stack/Heap segments of the processand their physical addresses	**2** **0x87f66000** **0x87f65000**	**2** **0x87f69000** **0x87f69000**	**2** **0x87f3c000** **0x87f3a000**	**Same page frames for all processes but different aresses**
Any dirty pages?	**NO**	**NO**	**NO**	**No dirty pages for any cases**
Any kernel mode (controlled) page frames?	**YES**	**YES**	**YES**	**Kernel mode page frames for all of them**
-Blank space-	**-Blank space-**	**-Blank space-**	**-Blank space-**	**Added this row by mistake**