Lab Assignment 3: Printing Page Table

Name : Rishitha Surineni
Roll No: CS22BTECH11050

Methodology:

In the kernel/vm.c the definition of the function vmprint() is written.
The pagetable_t attribute of the required page table is passed as the argument to this function. In vmprint(), the physical address of the page frame holding the outermost (level-1) page table of the process(which is passed as the argument) is printed and then the function vmprint_level() is called with arguments as this address of page table and level(which is 1).

In the vmprint_level() function, a loop is run for 512 iterations (as each page table contains 512 entries). In each iteration pte of data type pte_t is defined and it is given the value of i th row of page table.
if(pte && PTE_V) checks if the pte is valid or not and if it is valid then a loop is run for level number of iterations and .. is printed in each iteration of the loop. This gives the level of the pte.
pte is converted to the corresponding next level entry using PTE2PA.
A print statement is written to print the index value, pte and pa(i.e PTE2PA)
Then it is checked if (PTE_R|PTE_W|PTE_X)) == 0, if the condition is not true then it is the last level of the page table (i.e leaf) and it contains PPN.
If the condition is true then it is not leaf and we have to traverse to the next level this is done by calling this function on child(which is obtained by PTE2PA(pte)) this will take it to the next level of the page table.

Outputs:

Q1) Print the page table of init process

```
pagetable 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
```

Q2)Print the page table of sh process

```
init: starting sh
pagetable 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
```

Q3) Print the page table of user-level sleep process (refer to Lab Assignment 2)

```
$ sleep 10
pagetable 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
$
```

Q4) What is the size of the page frame, number of entries in a page table and address bits of virtual address space and physical address space in xv6?

Size of page frame = 4KB
Number of entries in each page table = 2^9=512
Address bits of virtual address space = 39 (in 64 address space only the lower 39 bits are used and the upper 25 are not used in sv39 configuration)
Address bits of physical address space = 54

Q5) How does xv6 allocate bits of virtual address space for multi-level paging and offsetting?

xv6 uses a three level paging system. The virtual address space is effectively 39 bits (the upper 25 bits of the 64 bits address are not used).

The lower 12 bits of virtual address space are allocated for page offsetting.

In the upper 27 bits,
The upper 9 bits are used to select a Page Table Entry in the root page table, the next 9 bits to select PTE in the middle level page table and last 9 bits to select the PTE in third level.

Q6) Given a pte, how do you determine whether it's valid (present) or not and how do you determine page frame number in pte is of next level page table or user process'? List out some valid pte entries from one of the Q1/Q2/Q3 which fall under above two categories.

We can determine if the page table entry is valid or not by looking at the Valid bit in the address. In implementation we can check it by using PTE_V . This indicates whether the page table entry is valid or not, if it is not set then its reference throws an exception.

If anyone of PTE_R,PTE_W,PTE_X is set to 0 then it means that the page frame number in pte is of next level but if all of them are non zero then it means the address formed from the page frame number in pte has read,write and execute permissions which implies it of user process. By these flags we can recognize if the page frame number is of next level page table or of user process.

page frame number in pte is of next level page table :

0x0000000021fda001
0x0000000021fd9c01
0x0000000021fd6c01
0x0000000021fd0401

page frame number in pte is of user process:

0x0000000021fd9017
0x0000000021fd641b
0x0000000021fd5c07
0x0000000021fcf017

Q7) By referring to the pagetables of init/sh/sleep processes, fill out the following table. Please explain your response as remarks, in brief.

	init process	**sh process**	**sleep process**	**Remarks**
No. of page frames consumed by the page table	5	5	5	Three level paging is followed and in all the above processes the number of page frames consumed is the same . It is one page frame for outer level,2 middle level and 2 inner level

Internal fragmentation in the page table(s) in bytes	20400	20392	20400	The internal fragmentation is almost same in all the three process. It is because of the invalid page table entries in the all the page frames occupied by the page table.
No. of page frames allocated for the process	6	7	6	These are page frames corresponding to physical memory and we get these from last level of page table.
No. of page frames allocated for TEXT segment of the process and their physical addresses	4	5	4	Text section is located at 0
No. of page frames allocated for Data/Stack/Heap segments of the process and their physical addresses	2 0x0000000087f74000 0x0000000080007000	2 0x0000000087f6d000 0x0000000080007000	2 0x0000000087f60000 0x0000000080007000	These are located in between
Any dirty pages?	0	0	0	8th bit from the end is 0 for all the addresses
Any kernel mode (controlled) page frames?	1	1	1	