Lab Assignment 3: Printing Page Table

I added the vmprint() function in ***kernel/vm.c***, given below the code:-

```
void _pteprint(pagetable_t pagetable, int level)
{
 for(int i = 0; i < 512; i++) {
   pte_t pte = pagetable[i];

   if (pte & PTE_V) {
     for (int j = 0; j <= level; j++)
       printf(".. ");
     // printf("\b%d: pte %p pa %p\n", i, pte, PTE2PA(pte));
     printf("%d: pte %p pa %p\n", i, pte, PTE2PA(pte));
   }
   if ((pte & PTE_V) && (pte & (PTE_R|PTE_W|PTE_W)) == 0) {
       // this PTE points to a lower-level page table.
   uint64 child = PTE2PA(pte);
   _pteprint((pagetable_t)child, level+1);
   }
 }
}

void vmprint(pagetable_t pagetable)
{
 printf("page table %p\n", pagetable);
 _pteprint(pagetable, 0);
}
```

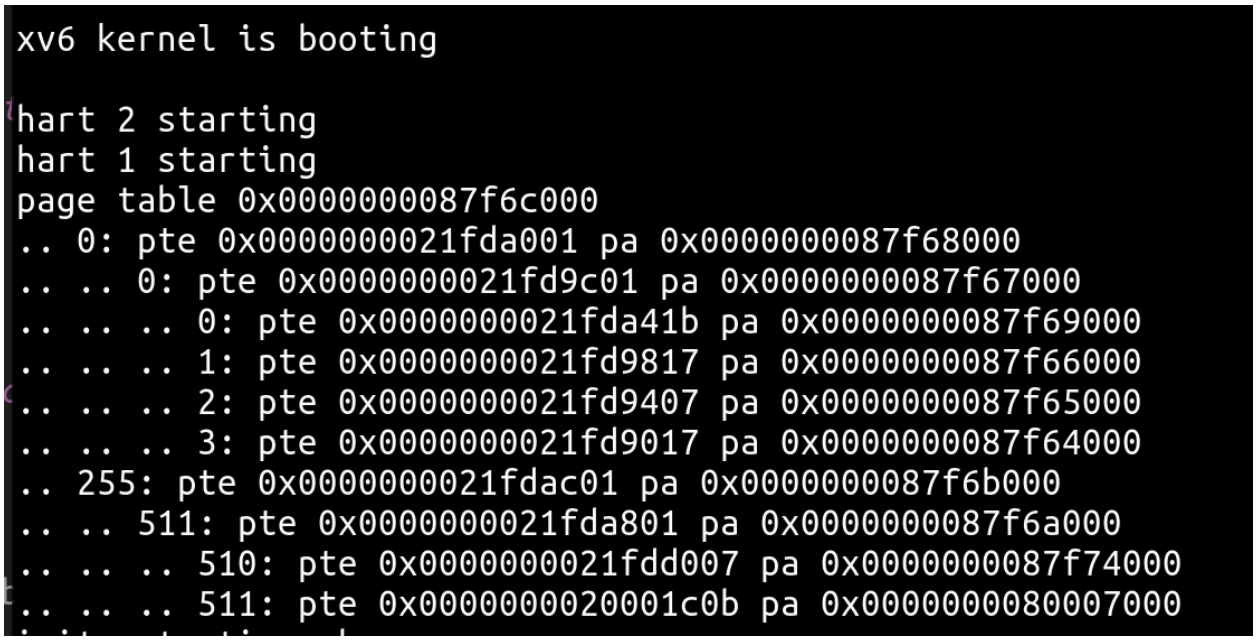
EXPLANATION OF CODE:-

The provided code is a C implementation designed to print the entries of a multi-level page table, a critical data structure used in virtual memory management systems. It begins with the declaration of the `_pteprint` function, which takes a `pagetable_t` data structure representing a page table and an integer `level` to track the current traversal depth within the table. Utilizing a loop iterating over each entry of the page table, the function extracts the page table entry (PTE) at the current index and evaluates its validity using the `PTE_V` flag. If valid, the function prints the index, PTE value, and the corresponding physical address derived from the PTE, also employing indentation to signify the hierarchical structure of the page table. Furthermore, it checks if the PTE points to another page table by examining its permissions and, if so, recursively calls

itself with the child page table and an incremented level. The `vmprint` function serves as the entry point for printing the page table, initiating the process by printing the address of the provided page table and invoking `_pteprint` with the initial level set to 0. Overall, this code provides a comprehensive view of the page table structure, aiding in debugging and understanding the intricacies of virtual memory management within operating systems or similar system-level software.

Then I Inserted **vmprint(p->pagetable)** in exec.c just before the **return argc**, to print the process's page table of your interest.

Q1. Print the page table of init process.



Q2. Print the page table of sh process.

```
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
```

Q3. Print the page table of user-level sleep process.

```
$ sleep-CS22BTECH11057 10
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
$
```

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?
ANSWER:-

```
//
// The risc-v Sv39 scheme has three levels of page-table
// pages. A page-table page contains 512 64-bit PTEs.
// A 64-bit virtual address is split into five fields:
//   39..63 -- must be zero.
//   30..38 -- 9 bits of level-2 index.
//   21..29 -- 9 bits of level-1 index.
//   12..20 -- 9 bits of level-0 index.
//    0..11 -- 12 bits of byte offset within the page.
```

Image is from file **vm.c**

The page frame size is 4KB.

As we can see that the lowest bits for offset are 12 bits, and the page size should be 2^12 Bytes or 4KB.

Number of entries in a page table is 2^9 = 512

Address bits of virtual address space: 64 bits

Address bits of physical address space : 56 bits ((44) -> page + (12) -> offset)

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

ANSWER:-

Ext	L2	L1	L0	Offset
25	9	9	9	12

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.

ANSWER:-

A page table entry (PTE) is a data structure used by the operating system to manage memory and control page faults in a virtual memory system. It stores information about a particular page of memory, including the physical address of the page in memory, whether the page is present in memory or not, and various access permissions such as read, write, and execute .

The validity of a PTE is determined by examining the last bit of the entry, known as the valid or present bit. If this bit is set, the PTE is valid and indicates that the associated page is in the process' logical address space and is thus a legal page. If the bit is not

set, the PTE is invalid and indicates that the page is not in the process' logical address space.

The page frame number in a PTE can point to either a next-level page table or a user process. To distinguish between these two cases, one can examine the read (PTE R), write (PTE W), and execute (PTE X) flags. If these flags are all unset (0), the entry likely points to a next-level page table, indicating an intermediary node in the page table hierarchy. If any of these permissions are set, the PTE points to a physical page being used by a user process, denoting a leaf in the table.

For example, in the given entries, the top entry has the valid bit set, indicating that it is a valid PTE pointing to a physical page in memory. The next-level entry has the valid bit set and all permissions unset, indicating that it is a valid PTE pointing to a next-level page table. The leaf entry has the valid bit set and all permissions set, indicating that it is a valid PTE pointing to a physical page being used by a user process

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	Each process in the system has an outer page table containing two valid entries entry 0 and entry 255. Each of these entries points to two additional page tables. Therefore for each process,there are a total 5 page tables being used.
Internal fragmentation in the page table(s) in bytes	20400 B	20392 B	20400 B	Internal fragmentation occurs when allocated memory space is not fully utilized, leading to wasted memory. In the context of paging systems, this waste occurs when the memory allocated to a process does not align perfectly with page boundaries, leaving unused space within pages. This inefficiency results in the loss of memory that could have been utilized for other processes. Strategies like dynamic memory allocation can help reduce internal fragmentation by allocating memory based on actual process requirements, optimizing memory

				utilization. However, this approach may introduce complexity and overhead, requiring a balance between efficient memory usage and system performance.
No. of page frames allocated for the process	6	7	6	The number of page frames allocated to a process is equivalent to the total count of physical pages mapped by valid entries in the final level of the page table.
No. of page frames allocated for TEXT segment of the process and their physical addresses	1 0x0000000087f69000	2 0x000000087f5c000 0x0000000087f59000	1 0x0000000087f4000	All the pages with PTE_R,PTE_U, PTE _X all should not be 0.
No. of page frames allocated for Data/Stack/Heap segments of the process and their physical addresses	2 0x0000000087f66000 0x0000000087f64000	2 0x0000000087f58000 0x0000000087f56000	2 0x0000000087f3d000 0x0000000087f3b000	All the pages with PTE_R,PTE_U, PTE _W all should not be 0.
Any dirty pages?	0	0	0	Since the page tables are printed before the process begins execution, none of the pages have been modified yet.
Any kernel mode (controlled) page frames?	YES, 3	YES, 3	YES, 3	Because of the PTE_U flag, certain entries in the page table will be restricted, preventing users from accessing them.