Lab Assignment 3: Printing Page Table

Task: Write a function that prints the contents of page table of a process

Name: Aditya Waghmare
Roll number: CS22BTECH11061

Code-

```c
//Recursively Reads Page Table Entries
void print_entries(pagetable_t pagetable, int level){

   //Each Table has 512 Entries(given)
   for(int i = 0; i < 512; i++){

       if(pagetable[i] & PTE_V){//If Entry is valid

           //print .. for each level
           for(int j = 0; j <= level; j++){
               printf(".. ");
           }

           //Printing the index, pte, and physical address
           printf("%d: pte %p pa %p\n", i, pagetable[i],
PTE2PA(pagetable[i]));

           //The PTE points to a lower-level page table if process cannot
read or write this page
           if((pagetable[i] & (PTE_R|PTE_W)) == 0) {
               //recurse for the next level table
               print_entries((pagetable_t)PTE2PA(pagetable[i]), level+1);
           }

       }
   }
}
void vmprint(pagetable_t pagetable){

   printf("page table %p\n", pagetable);
```

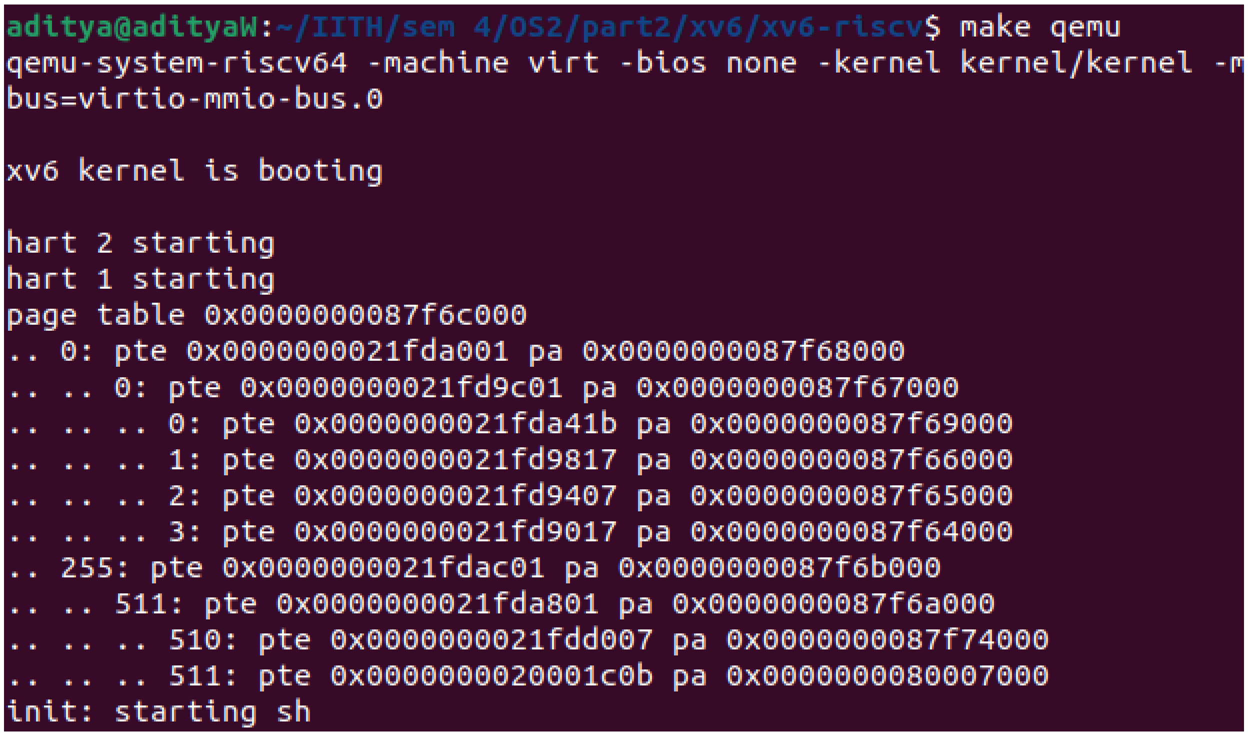
```
  print_entries(pagetable, 0);
}
```

Methodology-

1. vmprint() takes a page table as input and prints its address. It then prints its entries recursively with print_entries().
2. It is mentioned in riscv.h that each page table has 512 entries. We check which of this entries are valid with PTE_V flag.
3. We print the valid entries and if those entries are page tables we recursively read their entries as well.
4. We check if an entry is a page table with read and write permission flags. Process does not have permission to read or write page tables. So if an entry does not have read or write permission then it is a page table.
5. PTE2PA() is used to get the Physical Address from PTE.

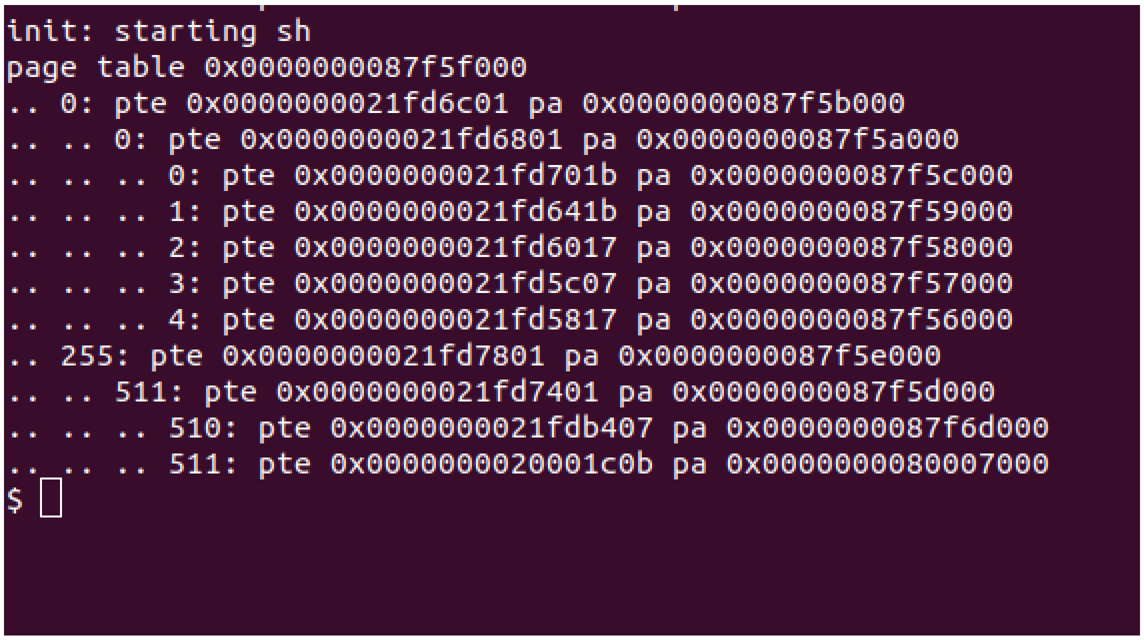
Questions-

Q1. Print the page table of init process
Ans:



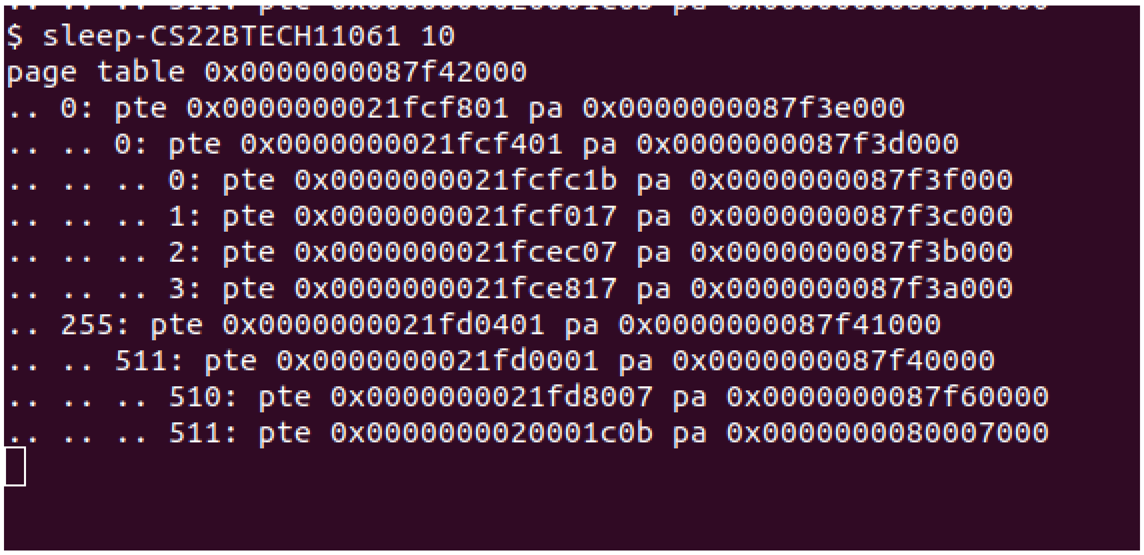
```
aditya@adityaW:~/IITH/sem 4/OS2/part2/xv6/xv6-riscv$ make qemu
qemu-system-riscv64 -machine virt -bios none -kernel kernel/kernel -m
bus=virtio-mmio-bus.0

xv6 kernel is booting

hart 2 starting
hart 1 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
```

Q2. Print the page table of sh process
Ans:



```
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 (refer to Lab Assignment 2)
Ans:



```
$ sleep-CS22BTECH11061 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?
Ans:

1. Page Frame Size = 4096 Bytes
2. Number of Entries in a page table = 512
3. Virtual Address = 64 bits
4. Physical address = 64 bits

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

Only the bottom 39 bits of a 64-bit virtual address are used. The top 25 bits are not used.

i) Offset = 12 bits
ii) 1st level page table = 9 bits
iii) 2nd level page table = 9 bits
iv) 3rd level page table = 9 bits

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.
Ans:

1. Given a PTE, we check if it is valid with PTE_V flag.
2. We determine page frame number in PTE is of next level page table's or user process's with PTE_R and PTE_W permission flags. If it is next level page table's then process does not have Read and Write permissions. If it is user process's then it has at least one of the two permissions.

From Q1 -

PTE pointing to next level table -

```
.. .. 0: pte 0x0000000021fd9c01 pa 0x0000000087f67000
```

PTE pointing to user process's Frame -

```
.. .. .. 3: pte 0x0000000021fd9017 pa 0x0000000087f64000
```

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**	**Number of Page Tables Used**
Internal fragmentation in the page table(s) in bytes	**20400**	**20392**	**20400**	**By Adding internal fragmentation in each table**
No. of page frames allocated for the process	**6**	**7**	**6**	**Number of Valid Entries in Last Level (level 3) Page Table**
No. of page frames allocated for TEXT segment of the process and their physical addresses	**Number = 2** **PA = 0x0000000087f69000, 0x0000000080007000**	**Number = 3** **PA = 0x0000000087f5c000, 0x0000000087f59000, 0x0000000080007000**	**Number = 2** **PA = 0x0000000087f3f000, 0x0000000080007000**	**4th last bit in PTE(X bit) indicates Text segment**
No. of page frames allocated for Data/Stack/Heap segments of the process and their physical addresses	**Number = 4** **PA = 0x0000000087f66000, 0x0000000087f65000, 0x0000000087f64000, 0x0000000087f74000**	**Number = 4** **PA = 0x0000000087f58000, 0x0000000087f57000, 0x0000000087f56000, 0x0000000087f6d000**	**Number = 4** **PA = 0x0000000087f3c000, 0x0000000087f3b000, 0x0000000087f3a000, 0x0000000087f60000**	**Page frames other than Text Segement**
Any dirty pages?	**No**	**No**	**No**	**8th last bit in PTE(D bit) indicates Dirty Page/ Dirty Bit**
Any kernel mode (controlled) page	**Yes**	**Yes**	**Yes**	**5th last bit in PTE(U bit)**

frames?				**indicates User Mode Access. Kernel Mode is Negation of this.**