<u>uv2p () System Call</u>

Name of the system Call:

uv2p() – uv2p system call translates the virtual address into Physical address.

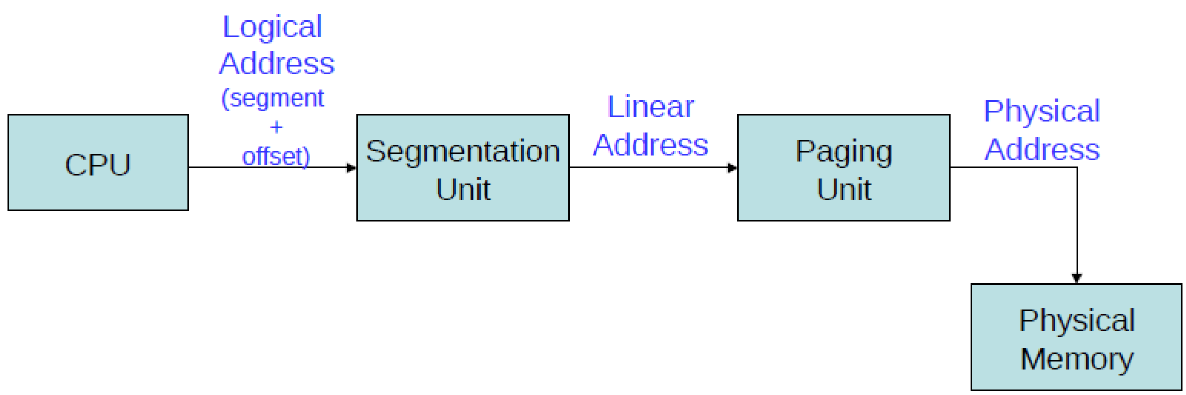
Description:

uv2p() system call displays the physical address when a valid virtual address is sent as a pointer to the uv2p system call. If an invalid virtual address is sent, then it displays the corresponding error messages.

Calling from the user level:

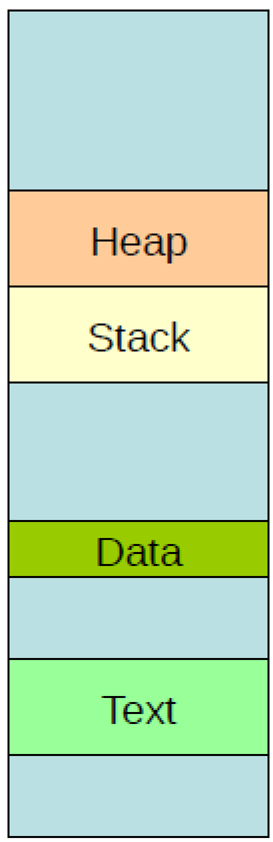
we created a .c file named uv2p.c inside the main program. What this file does is that it simply calls uv2p() system call from it. So whenever we run this file, uv2p() system call is automatically called.

XV6 Address Translation Architecture:



Segmentation Unit:

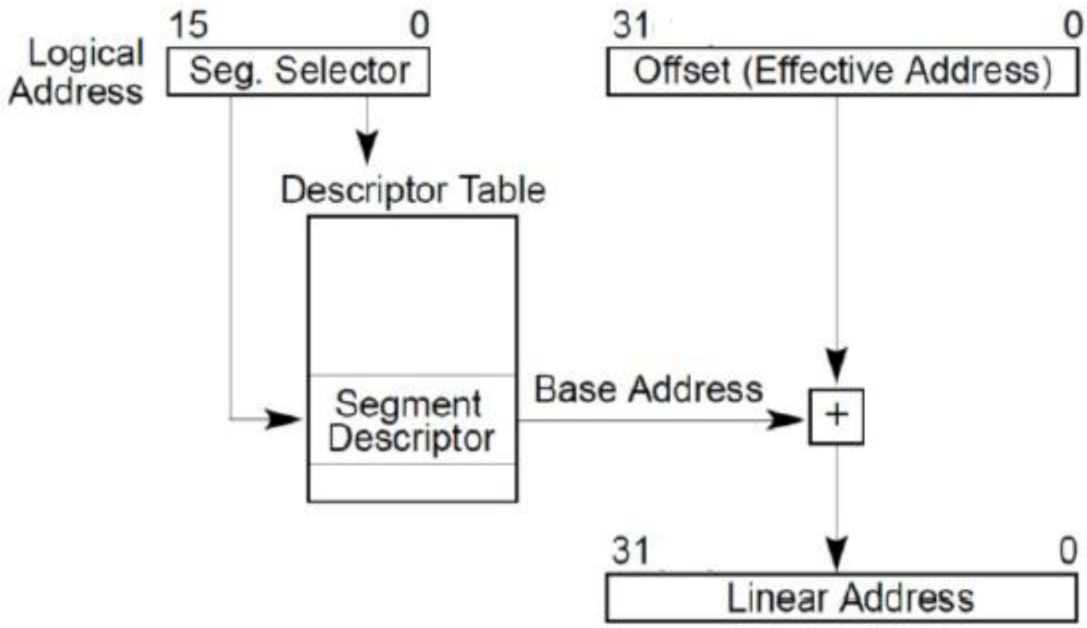
- Virtual address space of a process is divided into separate logical segments.
- Each segment is associated with a segment selector and an offset.



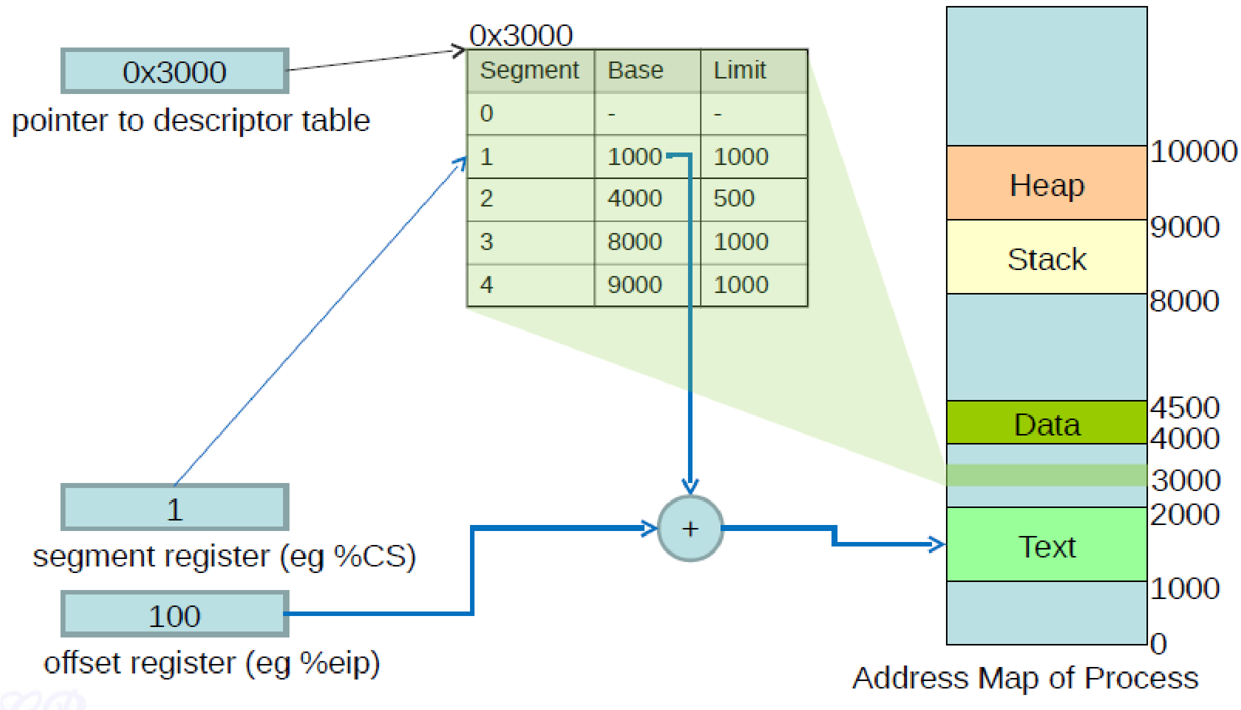
Address Map of Process

Logical and Linear address conversion:

- **We have a segment table and the segment of the given virtual address points to the segment descriptor in the segment descriptor table.**
- **From this location we extract the base address and we will add it to the 32 bit Virtual address(offset).**

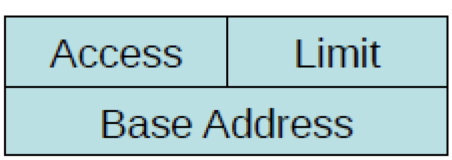


Example of Mapping to the segment descriptor Table:



Note: In the case of Xv6 architecture the base value for all the segments is set to zero. So the Linear address remains the same as the logical address after conversion. This address is called as virtual address and it is sent as an input to the paging unit.

Segment Descriptor:



- Segment descriptor has Base, limit address (0-4Gb) and bits for access rights. (Execute, Read, Write, Privilege level).

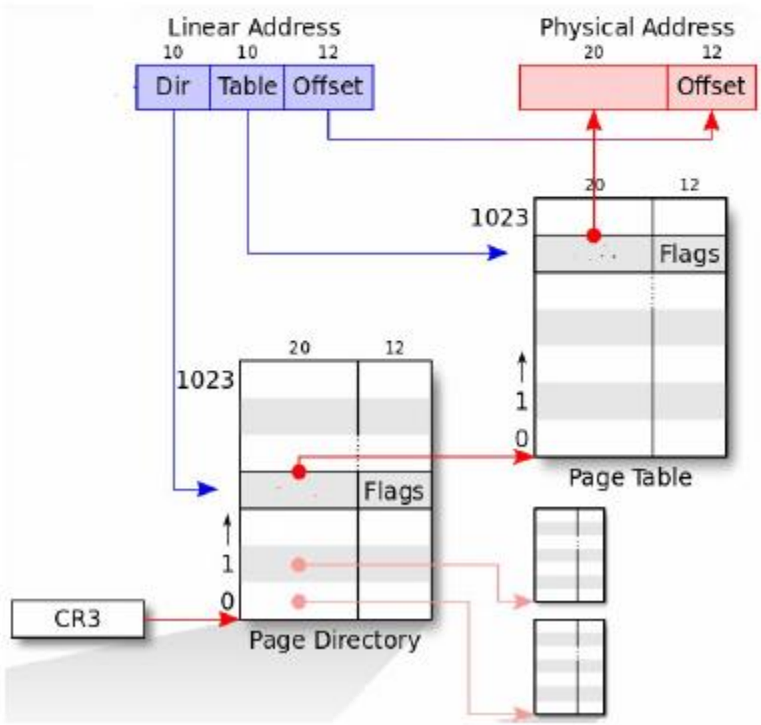
- Holds 16 bit segment selectors
  - Points to offsets in GDT
- Segments associated with one of three types of storage
  - Code
    - %CS register holds segment selector
    - %EIP register holds offset
  - Data
    - %DS, %ES, %FS, %GS registers hold segment selector
  - Stack
    - %SS register holds segment selector
    - %SP register holds stack pointer

    (Note: Only one code segment and stack segment can be accessible at a time. But 4 data segments can be accessed simultaneously)

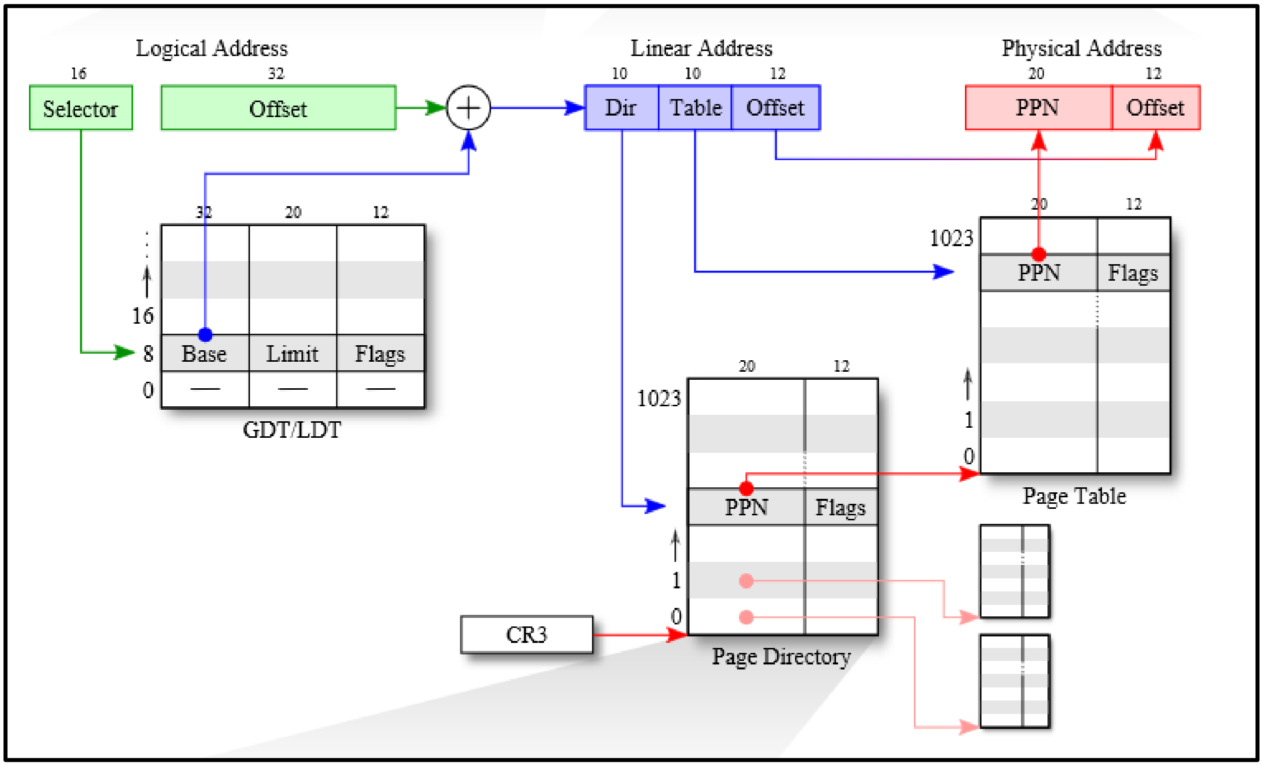
```
// Application segment type bits
#define STA_X       0x8     // Executable segment
#define STA_E       0x4     // Expand down (non-executable segments)
#define STA_C       0x4     // Conforming code segment (executable only)
#define STA_W       0x2     // Writeable (non-executable segments)
#define STA_R       0x2     // Readable (executable segments)
#define STA_A       0x1     // Accessed

// System segment type bits
#define STS_T16A    0x1     // Available 16-bit TSS
#define STS_LDT     0x2     // Local Descriptor Table
#define STS_T16B    0x3     // Busy 16-bit TSS
#define STS_CG16    0x4     // 16-bit Call Gate
#define STS_TG      0x5     // Task Gate / Coum Transmitions
#define STS_IG16    0x6     // 16-bit Interrupt Gate
#define STS_TG16    0x7     // 16-bit Trap Gate
#define STS_T32A    0x9     // Available 32-bit TSS
#define STS_T32B    0xB     // Busy 32-bit TSS
#define STS_CG32    0xC     // 32-bit Call Gate
#define STS_IG32    0xE     // 32-bit Interrupt Gate
#define STS_TG32    0xF     // 32-bit Trap Gate
```

Paging Unit:

- Paging unit has two level page translation
    1. Page Directory.
    2. Page Table.



Xv6 Architecture involving segmentation and Paging:



Details of the Architecture:

- Cr3 is a hardware register which helps to give the base of the page directory. CR3 enables the processor to translate linear addresses into physical addresses by locating the page directory and page tables for the current task.
- Cr3 register can be accessed directly using assembly language as follows.

```
asm ("\t movl %%cr3, %0" : "=r" (page_directory));
```

- We will traverse through the Page Directory using the Cr3 register and the 10 higher order bits of the Virtual address (22-31) (Page Directory Index) to get the corresponding Page Directory Element(PDE).

  Page Directory Element = CR3 + (Page Directory Index * Size of the Page Directory Element).
- The Higher order 20 bits of the PDE (11-31) contains the PPN (Physical page Number and it points to the start of the Page Table.

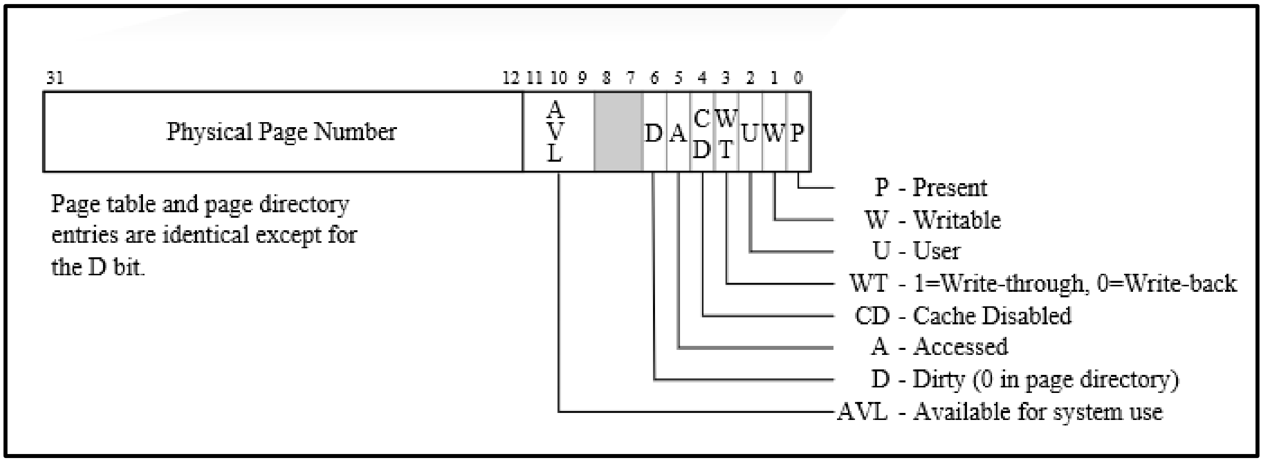
- The Lower 10 bits of the PDE (0-11) contains flags for protection purposes.
- Once the start address of the page table is obtained, we will traverse through the Page Table using the PPN of PDE and the 10 middle bits of the Virtual address (12-21) (Page Table Index) to get the corresponding Page Table Element(PTE).

  Page Table Element = PPN of PDE + (Page Table Index * Size of the Page Table Element).

- The Higher order 20 bits of the PTE (11-31) contains the PPN (Physical page Number.
- The Lower 10 bits of the PTE (0-11) contains flags for protection purposes.
- We will obtain the Physical address by adding the PPN of the Page Table Element (20 bits) with the offset (Lower order 12 bits) of the virtual address.

  Physical Address = PPN of PTE + (Offset of the virtual address).

Sample PDE/PTE:



- P -- (Present bit) Is to check whether a PTE is present or not. If the bit is not set, then this results in a page fault exception.
- W -- (Writable) If W is set to 1 then we can write or read, if it is set to 0 then we can only read.
- U -- (User) If this bit is set to 1 then user program is allowed to access else only the kernel is allowed to access.
- WT – If this bit is set to 1 then we can write through. If it is set to 0 then we can write back.
- CD – Cache disabled bit.
- A- Access bit.
- D – Dirty bit tells how long it is being waiting without getting accessed once it is loaded.
- AVL – Available for system use.

All the corresponding bit details are present in the mmu.h file.

```
// Page table/directory entry flags.
#define PTE_P           0x001   // Present
#define PTE_W           0x002   // Writeable
#define PTE_U           0x004   // User
#define PTE_PWT         0x008   // Write-Through
#define PTE_PCD         0x010   // Cache-Disable
#define PTE_A           0x020   // Accessed
#define PTE_D           0x040   // Dirty
#define PTE_PS          0x080   // Page Size
#define PTE_MBZ         0x180   // Bits must be zero
```

Testing of source Code:

Verification using Gbd:

Steps:

- Open a terminal and get into the folder where xv6 is present using cd command followed by path. Type in the command “**make qemu-nox-gdb**”. Once entered the xv6 will stop as shown below. Because we need to connect to the tcp port at 26000 port number.

```
$ QEMU: Terminated
user@cs3224:~/6.828/xv6-public$ make qemu-nox-gdb
dd if=/dev/zero of=xv6.img count=10000
10000+0 records in
10000+0 records out
5120000 bytes (5.1 MB, 4.9 MiB) copied, 0.0523919 s, 97.7 MB/s
dd if=bootblock of=xv6.img conv=notrunc
1+0 records in
1+0 records out
512 bytes copied, 0.00565175 s, 90.6 kB/s
dd if=kernel of=xv6.img seek=1 conv=notrunc
337+1 records in
337+1 records out
172708 bytes (173 kB, 169 KiB) copied, 0.00255599 s, 67.6 MB/s
sed "s/localhost:1234/localhost:26000/" < .gdbinit.tmpl > .gdbinit

(process:6663): GLib-WARNING **: /build/glib2.0-prJhLS/glib2.0-2.48.2/./glib/gme
m.c:483: custom memory allocation vtable not supported
*** Now run 'gdb'.
```

- Now we need to open a new terminal and enter “**gdb**” command.

```
user@cs3224:~/6.828/xv6-public$ gdb
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.5) 7.11.1
Copyright (C) 2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word".
+ target remote localhost:26000
warning: A handler for the OS ABI "GNU/Linux" is not built into this configuration
of GDB.  Attempting to continue with the default i8086 settings.

The target architecture is assumed to be i8086
[f000:fff0]    0xffff0: ljmp   $0xf000,$0xe05b
0x0000fff0 in ?? ()
+ symbol-file kernel
```

- Once gdb has started connect to the tcp port at 26000 as shown below.

```
(gdb) target remote:26000
A program is being debugged already.  Kill it? (y or n) y
Remote debugging using :26000
[f000:fff0]    0xffff0: ljmp   $0xf000,$0xe05b
0x0000fff0 in ?? ()
(gdb) break proc.c:643
Breakpoint 1 at 0x80104273: file proc.c, line 643.
(gdb) continue
Continuing.
```

- Inorder to stop(breakpoint) at a particular point we use "**break proc.c: line number**".
- Once we have done with setting up the break point we can use "**continue**" to start the xv6.

```
(process:6667): GLib-WARNING **: /build/glib2.0-prJhLS/glib2.0-2.48.2/./glib/gme
m.c:483: custom memory allocation vtable not supported
xv6...
cpu1: starting 1
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap star
t 58
init: starting sh
```

- Once the xv6 has started call system call by entering uv2p command. Since we have given a break point gdb will stop the execution and u can debug the code by giving "**n or next**" command.

```
649                     pde_t pgdir_val = PTE_ADDR(*pgdir_value); // extracting the PPN of PDE using PTE_ADDR(20 bits)
(gdb) n
=> 0x801042a0 <uv2p+112>:       pop    %ecx
650                     cprintf("Points to Page Table Start position: %p\n",pgdir_val); // Display base of page table
(gdb) nn
Undefined command: "nn".  Try "help".
(gdb) n
=> 0x801042a2 <uv2p+114>:       and    $0xfffff000,%edi
649                     pde_t pgdir_val = PTE_ADDR(*pgdir_value); // extracting the PPN of PDE using PTE_ADDR(20 bits)
(gdb) n
=> 0x801042a8 <uv2p+120>:       push   %edi
650                     cprintf("Points to Page Table Start position: %p\n",pgdir_val); // Display base of page table
(gdb) n
=> 0x801042a9 <uv2p+121>:       lea    -0x80000000(%edi,%ebx,1),%ebx
652                     globalpt=P2V_WO(globalpt);
(gdb) n
=> 0x801042b0 <uv2p+128>:       push   $0x80107968
650                     cprintf("Points to Page Table Start position: %p\n",pgdir_val); // Display base of page table
(gdb) n
=> 0x801042ba <uv2p+138>:       add    $0x10,%esp
654                     if(*pee & PTE_P){
(gdb) n
=> 0x801042c2 <uv2p+146>:       sub    $0x8,%esp
656                              cprintf("PTE Address (PPN of PDE +(PTI*size of PTE)):  %x\n",globalpt);//Display PTE address
(gdb) n
=> 0x801042d0 <uv2p+160>:       pop    %eax
660                              cprintf("Physical Address:  %x\n",pa); // Dispay the physical address
(gdb) n
=> 0x801042e9 <uv2p+185>:       lea    -0xc(%ebp),%esp
674         }
(gdb) n
=> 0x80104aea <syscall+42>:     lea    -0x8(%ebp),%esp
syscall () at syscall.c:149
149         }
(gdb)
```

- Once we exit the program after displaying physical address go to the cmd prompt and give ctrl+a and c.
- We get (qemu) now enter xp/x 0Xphysical address. Now we will get the value present in that location in hexa decimal format. Verify the value by converting it to decimal.

```
$ uv2p
50
Virtual Address:  8dfbcf1c
Offset: f1c
Page directory address base (cr3): df76000
Points to PDE (cr3+(PDI*size of PDE)):  8df768dc
Points to Page Table Start position: dff6000
PTE Address (PPN of PDE +(PTI*size of PTE)):  8dff6ef0
Physical Address:  dfbcf1c
QEMU 2.3.0 monitor - type 'help' for more information
(qemu) xp/x 0xdfbcf1c
000000000dfbcf1c: 0x00000032
(qemu)
```

```c
pde_t* get_pagedirectory(void)
{
   pde_t* page_directory;
   asm ("\t movl %%cr3, %0" : "=r" (page_directory));
        return page_directory;
}
```

- Above code is used to get the Cr3 register value from the hardware which points to the base of the page directory.

```c
int uv2p(pde_t val)
{

        pde_t* value = (pde_t*) &val;
        pde_t virtualdd;
        virtualdd = (pde_t) value;  // Virtual address assignment
        cprintf("Virtual Address:  %x\n",virtualdd);  // Virtual address
        int offset = ((virtualdd) &0xFFF);   // Extracting last 12 bits of va to get offset
        cprintf("Offset: %x\n",offset);
        pde_t* pgdir=get_pagedirectory();    // calling get pagedirectory to get the CR3 base register value
        cprintf("Page directory address base (cr3): %p\n",pgdir); // display base of Page Directory
        pde_t globalpt = (pde_t) pgdir +(4*(PDX(value))); // PDX gives first 10 bits of va * size of PDE(4) + Cr3 register
        globalpt=P2V_WO(globalpt);
        pde_t* pdd = (pde_t*) globalpt;
        if(*pdd & PTE_P){   //protection check  present bit
                cprintf("Points to PDE (cr3+(PDI*size of PDE)):  %x\n",globalpt);  // Display PDE
                pde_t* pgdir_value = (pde_t*) globalpt;
                pde_t pgdir_val = PTE_ADDR(*pgdir_value); // extracting the PPN of PDE using PTE_ADDR(20 bits)
                cprintf("Points to Page Table Start position: %p\n",pgdir_val); // Display base of page table
                globalpt = pgdir_val +(4*(PTX(value))); //PTX gives Middle 10 bits(12-21) of va * size of PTE(4) + PPN Of PDE
                globalpt=P2V_WO(globalpt);
                pde_t* pee = (pde_t*) globalpt;
                if(*pee & PTE_P){ //protection check  present bit
                        if(*pee & PTE_U){ //protection check  User mode bit
                                cprintf("PTE Address (PPN of PDE +(PTI*size of PTE)):  %x\n",globalpt);//Display PTE address
                                pde_t* pgtable_value=(pde_t*) globalpt;
                                pde_t pgtable_val = PTE_ADDR(*pgtable_value); // Extract PPN of PTE(20 bits) using PTE_ADDR
                                pde_t pa= pgtable_val|offset; // PPN of the PTE + Offset
                                cprintf("Physical Address:  %x\n",pa); // Dispay the physical address
                        }
```

```c
                        else{
                                cprintf("User process can't access this page - Protection \n");
                        }
                }
                else{
                        cprintf("You don't have the page in Physical memory - Page Fault exception\n");
                }
        }
        else{
                cprintf("You don't have the PTE - Page Fault exception\n");
        }
        return 23;
}
```

Case 1: (Given a valid address)

- Passing a value of 50 and virtual address corresponds to the address of the value 50. Once given a virtual address the user program converts the virtual address into corresponding physical address.

```
$ uv2p
50
Virtual Address:  8dfbcf1c
Offset: f1c
Page directory address base (cr3): df76000
Points to PDE (cr3+(PDI*size of PDE)):  8df768dc
Points to Page Table Start position: dff6000
PTE Address (PPN of PDE +(PTI*size of PTE)):  8dff6ef0
Physical Address:  dfbcf1c
QEMU 2.3.0 monitor - type 'help' for more information
(qemu) xp/x 0xdfbcf1c
000000000dfbcf1c: 0x00000032
(qemu)
```

- We got a physical address of dfbcf1c when we check the value inside the physical address we got 0X00000032 in hexa decimal. (16*3 + 2 =50). So, the hexa decimal number of 50 is 0X00000032.Hence verified.

Case 2:

- If we don't give a valid virtual address and the page table element dosen't exists (i.e) we will check the Present bits of the PDE. Since it is 0 we get an error message stating that "You don't have the PTE - Page Fault exception".

```
$ uv2p
Virtual Address:  8dfbcf40
Offset: f40
Page directory address base (cr3): df76000
You don't have the PTE - Page Fault exception
```

Case 3:

- If we don't give a valid virtual address and the present protection bit in the PTE is not set to 1, then we will get an error message stating that page is not present in the physical memory.

```
$ uv2p
Virtual Address:  8dfbcf68
Offset: f68
Page directory address base (cr3): df76000
Points to PDE (cr3+(PDI*size of PDE)):  8df768dc
Points to Page Table Start position: dff6000
You don't have the page in Physical memory - Page Fault exception
```

Case 4:

- If the user bit in the PTE is not set to 1 then we will get an error message stating that user process cannot access this page.

```
$ uv2p
Virtual Address:  8dfbcf90
Offset: f90
Page directory address base (cr3): df76000
Points to PDE (cr3+(PDI*size of PDE)):  8df768dc
Points to Page Table Start position: dff6000
User process can't access this page - Protection
```

- If any user program tries to do write on a page which can only be read an exception occurs stating that user process can't access.

If an invalid virtual address is given a trap occurs and it is handled by the operating system.