**PROJECT 3 DEMO**

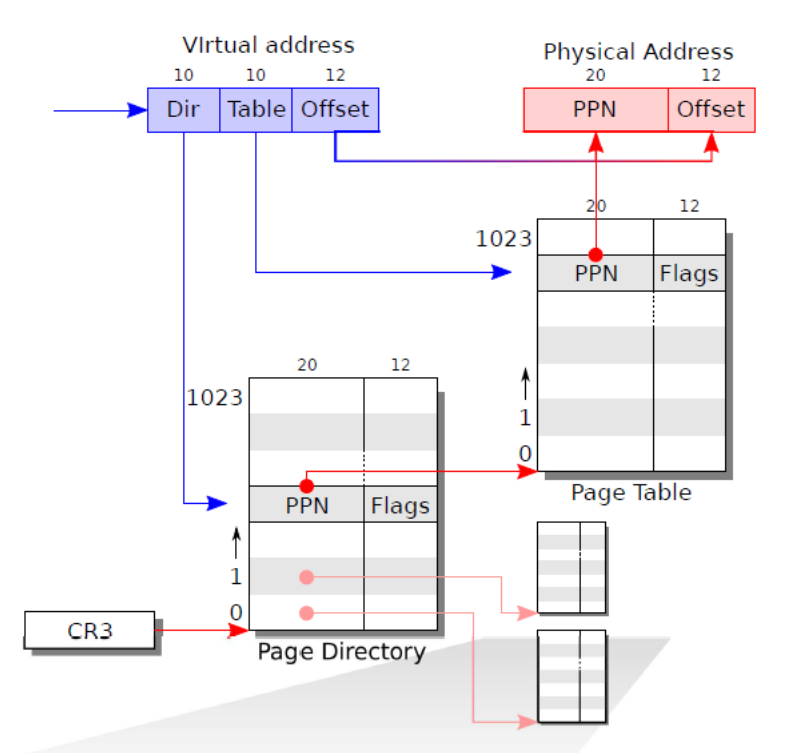
Q1: What is the page size with xv6? Use xv6’s code to support your answer.

* Page size with xv6 is 4 KB. Code in mmu.h file defines PGSIZE which is a macro which has 4096 bytes.

#define PGSIZE 4096 // bytes mapped by a page

Q2: Does xv6 use linear paging or multi-level paging? If multi-level paging, how many levels are there? Use xv6’s code to support your answer.

* xv6 makes use of multilevel paging and has 2 levels. In file named mmu.h, the operations involve page lookups and also the shift mask definitions.
* Page Directory and Page Table constants are also defined as macros
* #define NPDENTRIES 1024 // # directory entries per page directory
* #define NPTENTRIES 1024 // # PTEs per page table
* // A virtual address 'la' has a three-part structure as follows:
* //
* // +--------10------+-------10-------+---------12----------+
* // | Page Directory | Page Table | Offset within Page |
* // | Index | Index | |
* // +----------------+----------------+---------------------+
* // \--- PDX(va) --/ \--- PTX(va) --/



Q3 :

1. What is the address range of virtual memory that is used for the OS kernel?

* Virtual Memory address for the OS Kernel starts from [ KERNBASE to KERNBASE+PHYSTOP ]
* The virtual addresses within the range – [ KERNBASE to KERNLINK i.e. (KERNBASE+EXTMEM) ] are used for IO devices
* All the OS code and data are stored from virtual address [ KERNLINK to KERNBASE+PHYSTOP ]

1. Where is the above range of virtual addresses mapped to in physical memory?

* OS Kernel Physical memory – [ 0 to PHYSTOP ]
* IO Devices – [ 0 – 0x100000 ]

// Memory layout

#define EXTMEM 0x100000 // Start of extended memory

#define PHYSTOP 0xE000000 // Top physical memory

#define DEVSPACE 0xFE000000 // Other devices are at high addresses

// Key addresses for address space layout (see kmap in vm.c for layout)

#define KERNBASE 0x80000000 // First kernel virtual address

#define KERNLINK (KERNBASE+EXTMEM) // Address where kernel is linked

#define V2P(a) (((uint) (a)) - KERNBASE)

#define P2V(a) (((void \*) (a)) + KERNBASE)

#define V2P\_WO(x) ((x) - KERNBASE) // same as V2P, but without casts

#define P2V\_WO(x) ((x) + KERNBASE) // same as P2V, but without casts

**kmap[]** = {

{ (void\*)KERNBASE, 0, EXTMEM, PTE\_W}, // I/O space

{ (void\*)KERNLINK, V2P(KERNLINK), V2P(data), 0}, // kern text+rodata

{ (void\*)data, V2P(data), PHYSTOP, PTE\_W}, // kern data+memory

{ (void\*)DEVSPACE, DEVSPACE, 0, PTE\_W}, // more devices

};

1. How is the kernel’s virtual memory mapped to physical memory (using linear mapping or using page table)? Show the xv6’s which achieves this mapping.

* XV6 uses Linear mapping for the kernel memory. The conversion is done using a one-to-one mapping by adding the offset of KERNBASE to Physical address for Virtual Address and subtracting KERNBASE from Virtual Address to get Physical Address.

Q4 : The original xv6 use the “direct copying” approach when forking. In other words, the forking needs to copy every page of the parent process for the child process. Show the code which achieve this functionality and explain this code.

* The direct copying happens with the help of copyuvm function in vm.c file.
* Here the function takes 2 parameters from the fork function, first being the Page Directory and the second being the size of the process memory.
* The function first creates the kernel part of the page. If it is not successful, it returns.
* Then, it loops through the pages used by the process one by one with the help of a for loop and the function **walkpgdir()**. Returns if the page is not found or if the page table entry is empty.
* While iterating, it creates temporary variables for copying the PTE content and the Flag data which were taken with the help of **PTE\_ADDR()** and **PTE\_FLAGS()**. And then moves the data from these variables to a newly allocated page with the help of **kalloc().**

pde\_t\* copyuvm(pde\_t \*pgdir, uint sz)

{

pde\_t \*d;

pte\_t \*pte;

uint pa, i, flags;

char \*mem;

if((d = setupkvm()) == 0)

return 0;

for(i = 0; i < sz; i += PGSIZE){

if((pte = walkpgdir(pgdir, (void \*) i, 0)) == 0)

panic("copyuvm: pte should exist");

if(!(\*pte & PTE\_P))

panic("copyuvm: page not present");

pa = PTE\_ADDR(\*pte);

flags = PTE\_FLAGS(\*pte);

if((mem = kalloc()) == 0)

goto bad;

memmove(mem, (char\*)P2V(pa), PGSIZE);

if(mappages(d, (void\*)i, PGSIZE, V2P(mem), flags) < 0)

goto bad;

}

return d;

bad:

freevm(d);

return 0;

}

Q5: Explain with code about how your CoW implementation works.

* At first, we make page table not writable by adding **~PTE\_W;** in **copyuvm()** in **vm.c** file
* And when process tries to write into read only or unwritable page table then it will throw an exception which will be caught in **trap.c** written as **T\_PGFLT:**
* From **trap.c,** it will be redirected to function **void pgFault(uint err\_code){}** written in **vm,c** to handle page faults exception to implement write logic on unwritable page table entries.
* It’s possible that there could be multiple forks. So, we have used reference counting to return the count of number of references to that particular page; which is written as uint **get\_pgRef\_cnt(uint pa)** in **kalloc.c** file.