- ROHIT KUMAR SINGH
- 2016ME10080

Q2: <u>ASSIGNMENT</u>: <u>PAGE TABLE RELOAD</u>

* Why is this zero?
→ Virtual address starts from 0 to $2^{31}-1$ (In case of 32-bit machine) and hence when accessing it by kpgdir[0] this gives us 0. It has been ~~seal~~ setup using setupkvm which gives us 0.

* How would we translate 0x80107beb to physical address?

VA			
	10	10	12

- we will use first 10 bits to look ~~dow~~ up the page table in the ~~a~~ page directory
- We will use the next 10 bits to look up for the page from the page table and finally the last 12 bits will tell us the page offset.

Through this we can translate ~~page~~ VA to PA.

* 0x114007
→ What is this?
~~This the address of the~~ page ~~table~~ and an ~~page directory entry~~.
This is a page directory entry which is used to get the address of the page table.

→ What is the PPN?
0x114

_/_/_

→ What does the 7 mean?

0x7 or 0b111 tells us about the flags for the said page table.

1 1 1

→ User Accessible

→ Writeble

→ Present.

Here all three flags are set meaning the page table is present and is pages are writable and user can possibly access ~~some~~ of the pages.

0x 107001

→ What is this?

This is a page table entry which can be used to get address of page containing the data associated with VA. First 20 bits tells us about the address of the page and last 12 bits are flags associated with that page.

→ Why is 1 in the low bits

Here the flags are 001

meaning the page is present and is read only and user has no privilege of accessing it.

→ Why did the physical address work in gdb?

Physical Address worked in gdb because x86 uses flat structure to address the PA and VA hence PA also worked here and when we started the program we made sure that this linear flat structure was followed.

→ ~~Why?~~

~~As the flat linear structure is not being followed in this case hence we are not able~~

→ Why not accessing the memory?

Because the page containing the memory is not user accessible and ~~not~~ re. through malloc.

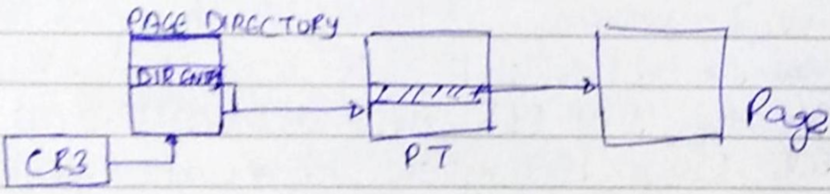
A1 4KB Page

VA 0x80100 $\boxed{000}$

VA | PDE (10 bits) | PTE (10 bits) | Page offset (12 bits) |

↓

PA 0x00100000



0x80100000 → ~~Page Dir Offset 0x200~~
~~Page Table offset 0x200~~
~~Physical Page offset 0x000~~

P

PDE Offset from first 10 bits

0b1000 0000 00 or $\boxed{0x200}$

PDE offset is 0x200 needs to be modified

PTE offset from 11th to 20th bit

0b01 0000 0000 or $\boxed{0x100}$

Here PDE will be 0x300 01 → Present, Read only; here 0x301 is the PT address

Here PTE will be 0x100 01

__/__/__

and the Page table value will be modified to

0x00100(05) → Read only and Present flag
1 User Ram view.   1 Flags 101

PD Entry will be modified 0x------01
Address of the Page Table.

A3 Addressing.

This will be wrong because when the kernel executes first, it will set up page structure to map VA 0X80100000 to its PA starting at 0X00100000. Kernel assumes that physical memory at lower offset is available. When the boot starts paging is enabled. Kernel will specify that ELF starts at 0x00100000 which will cause boot system to write garbage values at these address and hence crashing the system.

A4 Trap

It is not possible for a suspended process' k-stack to have two seperate context structure because once a process is context switched out, it returns to returning original structure using the context structure. ~~Addition~~ Additionally, a process can be context switched out only once. Moreover, context structure is last entry on the process k-stack.

(ii) Yes, it is possible to have two separate trap frames along with context structure on k-stack this can happen if a process' first trap occurs due to software interrupt/exception and ~~as~~ second due to external interrupt while executing in the kernel. While handling it can be context switched out which results in two trap structure and one context structure.

(iii) No, an process k-stack can have at ~~+~~ max one context structure and two trap structure. Hence it can only have at max 3. saved registers.

ASSIGNMENT: CONTEX SWITCHING

* Sched() can execute on the stack of any process thread, as it is allocated from the kernel's heap.

* Scheduler() runs on a separate stack that is not mapped to any process and doesn't have a user space. Each process has a scheduler stak which is also allocated from

* ~~When sched() calls switch()~~