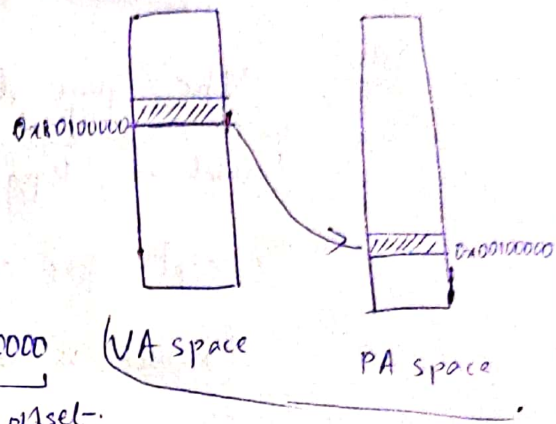
COL331 : HW3

Sumit Singh
2016PH10575

(A.) Assignment : Paging

VA = 0x80100000

PA = 0x00100000



~~VA =~~

VA = 0100 0000 0001 0000 0000 0000 0000 0000

$\underbrace{0100\ 0000\ 00}_{\text{PD offset}}\ \underbrace{01\ 0000\ 0000}_{\text{PT offset}}\ \underbrace{0000\ 0000\ 0000}_{\text{PA page offset}}$

→ Offset in page directory = $(0100000000)_2 = (256)_{10}$

→ Offset in page table = $(0100000000)_2 = (256)_{10}$

→ Entry in page-table → Leftmost 20 bits = leftmost 20 bits of PA address (0x00100000)

Right 12 bits = PRESENT Bit set and other flags 0.

Page table entry (at offset 256) { 0000 0000 0001 0000 0000 0000 0000 0001

$\underbrace{0000\ 0000\ 0001\ 0000\ 0000}_{\text{First 20-bits.}}\ \underbrace{0000\ 0000\ 0001}_{\text{Last 12 bits flags}}$ ← PRESENT bit

→ Page directory entry : Assume that the page-table page is present at 0x00101000

then

PD entry : $\underbrace{0000\ 0000\ 0001\ 0000\ 0001}_{\text{First 20 bits.}}\ \underbrace{0000\ 0000\ 0001}_{\text{Last 12 bit flags}}$ ← PRESENT bit

(B.) Page Table Reload :-

(a) print/x kpgdir[0] → why is this zero?

∵ ~~execution has reached the beginning~~ →

Ans:- The page directory page returned by setupkvm() (which is 'kpgdir' now) has all its entries zero except for the four entries present in 'kmap', (which ~~is at non~~ are not at 0th offset).

Thus kpgdir[0] gives 0.

(b) How do we translate 0x80107beb to pa?

Ans:- At this point, there is identity mapping from (0x80000000 + 4 MB) 'va' to (0 + 4 MB) 'pa'.

∴ Subtract 0x80000000 from 'va' to get 'pa'

∴ pa = 0x00107beb

(c) print/x 0x80107beb >> 22

$4 = 0x200

print/x kpgdir[0x200]

$6 = 0x114007

→ what is this?

– The ~~top~~ leftmost 20 bits of this no. (0x114007) locate the page-table for the (0x200)th page-directory entry.

Scanned with CamScanner

→ what is the PPN ?

PPN = leftmost 20 bits of 0x114 007

= 0x114

→ what does the 7 mean?

∵ hex digit is 7 ∴ last four bits are 0111

rightmost bit = PTE_P = 1 (i.e. page present)

second right bit = PTE_W = 1 (i.e. page is writable)

third right bit = PTE_U = 1 (i.e. page is user-accessible)

('page' refers to page ~~table~~ ~~ind~~ indicated by the entry)

(d) print/x (0x80107beb >>12) & 0xfff

$ 6 = 0x107

print/x ((int *) 0x114000) [0x107]

$12 = 0x107001

what is this ?

— The page-table entry in the (page-table) page referenced by 0x114007.

why 1 is in the low bits ?

— 1 in low bits means last four bits of this entry are 0001. This means the physical page referenced by it is 'present', 'read-only' and 'non-user accessible'.

→ why did the physical address work in the gdb?

- %cr3 has not yet been loaded with the page directory (kpgdir) (∵ switchkvm() has not yet returned). So the physical address is valid.

→ why doesn't 0x107beb work after switchkvm() has executed?

- After switchkvm() has executed and %cr3 has been loaded with kpgdir, the paging hardware treats 0x107beb as a virtual address whose mapping doesn't exist in the page directory (kpgdir). Thus the corresponding memory address can't be accessed.

D. Traps :-

→ Is it possible to have two "context" structures and one "trapframe" structure on kstack ?

Ans:- No.

"Context" is pushed by 'swtch' function. After pushing the "context", the 'swtch' function switches to another kstack and pops off the saved context there. When 'swtch' switches again to the previous stack, it pops off the saved 'context'.

So the "context" can be saved only after the previous one has been popped off. Thus we can't have two "context" structures at the same time.

→ Is it possible to have two trapframe structures and one context structure on the kstack ?

Ans:- ~~Yes~~. Yes.

~~When the interrupt handler itself needs to do context~~

Scenario :

- User process sends a software interrupt
- It's trapframe gets pushed and the corresponding interrupt handler starts running.
- while the interrupt handler is running, the timer interrupt occurs.
- So another trapframe is pushed and the handler of timer interrupt calls the 'scheduler' which then pushes the 'context'. Thus two 'tf' and one 'context'.

→ Is it possible to have more than three sets of saved registers?

Ans:- No.

We can have at most two trapframes. The second trapframe (if exists) will be due to the timer interrupt whose handler will run (in kernel mode) with interrupts disabled. So we can't have any more trapframes.

Also, we can have only one 'context' on a kstack.

Two 'tf' and one 'context' make at most three sets of saved registers.

(E) · Context Switching:-

Suppose a process that is running in the kernel calls sched(), which ends up jumping into scheduler().

→ Where ~~does~~ is the stack that sched() executes on?

~~Ans: the 'kstack' of the process~~

Ans:- sched() executes on the kstack of the calling process which lives in the kernel address space.

→ Where is the stack that scheduler() executes on?

Ans:- scheduler() runs on the same stack on which main() (in /main.c) runs. There's this one initial stack per cpu on which the scheduler() runs and this stack is different from the per-process kstacks.

→ When sched() calls switch(), does switch() ever return? If so, when?

Ans:- switch() returns only when the ~~sc~~ scheduler() switches back to the process (i.e. when scheduler calls switch() with the current process's context as argument).

→ cprintf("a");
switch(&cpu→scheduler, &proc→context);
cprintf("b");
} in scheduler()

cprintf("c");
switch(&proc→context, cpu→scheduler);
cprintf("d");
} in sched()

→ what is the four characters pattern?

Ans:- "cbad".

→ The very first characters are 'ac'? why?

Ans:- when the first process is created, the ~~user init~~ userinit() function prepares the kstack ('tf', 'context' etc.) and scheduler() on it. Thus first 'a' is printed. In the next line & after cprintf("a"), the control switches to the new process. So "c" is printed. when this new process calls "sched()". Again, the switch occurs to the scheduler where switch() returns & "b" is printed, & so on.