HOMEWORK 3

2016CS10348

HIMANSHU

1. On x86 using 4KB pages only, we wish to map virtual address 0x80100000 to physical address 0x00100000 with read-only permissions. Assume that all segments are setup to [0:0xffffffff] and use only paging. You will need to describe the offset at which the page directory and the page table need to be modified and to what values.

Ans:- Page directory

Offset:- Last 10bits of virtual address represent offset of page directory.

Last 10 bit of 0x80100000 -> In Binary (1000 0000 00) -> 0x200

Flagbit

present bit =1

Writable bit = 0 (Read only mode)

user = 0 (kernal mode as VA is above KERNBASE)

other bit =0

PPN : can be any valid address

So we need to modify 0x200th entry of Page directory by flag-bit 0x001 and with valid PPN

Page table

Offset:- 12th to 21st bit of virtual address represent offset of page table.

12th to 21st bit of 0x80100000 -> In Binary (01 0000 0000) -> 0x100

Flagbit

present bit =1

Writable bit = 0 (Read only mode)

user = 0 (kernal mode as VA is above KERNBASE)

other bit =0

PPN : 12th to 31st bit of physical address represent PPN of page table.

12th to 31 bit of 0x00100000 -> 0x00100

So we need to modify 0x100th entry of Page Table by 0x00100001

2. (gdb) print/x kpgdir[0]

why is this zero?

Ans:-Kpgdir is just initialised and its page-directory value is not updated at this page.

The last bit is 0 that signifies that this page is absent and don't contain any correct value. As the system is boot-up so it don't contain cache, so other bits are zero.

How would we translate 0x80107beb to a physical address?
By subtracting KERNBASE from the virtual address.
Physical Address = 0x80107beb - KERNBASE(0x80000000)

```
(gdb) print/x kpgdir[0x200]
$6 = 0x114007
```
Q: what is this?
Ans: This is 0x200 entry of kpgdir and contain address of second level page table.

Q: What is the PPN?
Ans:- 0x114000

Q: What does the 7 mean?
The 7 represent:
Present =1
Writable=1
User mode = 1
This means that a given entry in a page is present with writable mode and can be accessed by the user.

```
(gdb) print/x ((int*)0x114000)[0x107]
$12 = 0x107001
```
Q: what is this?
Ans: It is 0x107 entry of a page Table which pointed to 0x200th page directory. In other words, it is starting address in physical address corresponding to two level page table of 0x80107beb.

Q: why 1 in the low bits?
1 represent flag with:
Present =1
Writable=0
User mode = 0
Page is present. And as it is physical address so only readable and can accessed in kernel mode.

```
(gdb) x/i 0x107beb
```
why did the physical address work in gdb?
Ans: Till now, paging is not enabled. Hence till now virtual address is same as physical address. As a result, pa is accessible.

0x107beb:	Cannot access memory at address 0x107beb
why?
Now, it is called setupkvm to create a page table. As page table is created, physical address can’t be accessed.

3.	Suppose you wanted bootmain() to load the kernel at 0x80200000 instead of 0x80100000, and you did so by modifying bootmain() to add 0x100000 to the va/pa of each ELF section. Something would go wrong. What?
Ans:- In this, we are asked to add 0x100000 to each ELF section. This effectively means that it allocates new pages to each ELF section.
This is something which would not work. Because Actually, there is shared memory using pointers at various sections. In such shared memory, we are not required to add 0x100000, as this new x100000 is not linked in different ELF sections . So we don’t need to add pages at each section.

4.1	Is it possible to have two "context" structures and one "trapframe" structure on the kstack? If so, when? If not, why not?
Ans:- No, there can’t be two contexts on the same kstack. By the time you push “context” , you have already switched. Also while pushing context, there is interrupt disabled mode, lock is acquired and this prevents it from interrupting in the middle of context-call.

4.2	Is it possible to have two trapframe structures and one context structure on the kstack. If so, when? If not, why not?
Ans:- Yes, This will happen when there is an interrupt in the user process and it starts running instruction of kstack. While running in kernel mode, a timer interrupt will come. Kernal store values and switches to any other process. In such cases, there are two trapframe for one for user system call and another one for kernel mode interrupt but it has only one context structure(one context which is obtained during timer interrupt.)

4.3	Is it possible to have more than three sets of saved registers in the kstack? If so, when? If not, why not?
Ans:-No. In xv6, there cannot be three sets of saved registers because there can be a maximum two trapframe. It is ensured by using lock and external interrupt disabled which prevent from interrupting the set of instructions in the middle.
In theoretical case, Yes there can be more than three set in the case when external interrupt are not disabled. In such case, there are multiple external device interrupts.

5.1 Where is the stack that sched()executes on?

Ans:- Sched() execute on the kstack of the current running process. Sched() is not a user process and it doesn't have a stack of its own.

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

Ans:- Scheduler is running on the same stake on which main() function is running. Scheduler don't have their own stack.

5.3 When sched() calls swtch(), does that call to swtch() ever return? If so, when?

Ans:- Yes swtch() function return ultimately in next switch. Although, Swtch function is not going to return immediately but It will return in next swtch.

5.4 Could swtch do less work and still be correct? Could we reduce the size of a struct context? Provide concrete examples if yes, or argue for why not.

Ans:- At present, swtch is storing and loading only callee-saved register. It can't be reduced further because these are require to maintain gcc calling convention and these register are used in future.
"Struct context" contains callee saved registers edi, esi, ebx ebp and eip. These can't be reduced in size as we need to follow gcc calling convention.

5.5 What is the four-character pattern?

Ans:- Pattern= cbad
The output is "acbadcbad..." so clearly after entering through "a", it flow pattern of "cbad" pattern

5.6 The very first characters are ac. Why does this happen?

Ans:- During the overall setup (in the main.c file) it calls mpmain(). Mpmain() is the setup code of a common CPU. In this mpmain(), for running processes, scheduler() is initialized for first time and we print "a"

While, New process is using the CPU, it uses yield, sleep, exit. This yield() function is called the sched() function and 'c' is printed.