N TARUN SAI GANESH
2016EE10438

OS Homework 3

Paging: To map Virtual address 0x80100000 to the physical address 0x00100000 with read-only permissions.

— Virtual address in binary form

$$= \underbrace{1000000000}_{\text{gives page directory offset}} \; \underbrace{0100000000}_{\text{gives page table offset}} \; 000000000000$$

offset in the page directory (or index) can be obtained from first 10 bits of VA.

so, Page directory offset = 512

offset in page table is obtained from next ten bits.

so, page table offset = 256.

Since Physical address,

PA = first 20 bits of page table entry + last 12 bits of VA

so, first 20 bits of page table entry should be equal to first 20 bits of PA = 0001000000 0100000000

To set read permission, the 2nd most LSB should be set to '0'.

so, Pte = 0001000000 0100000000 0000000001

$$\begin{tikzcd} 0000000001 \arrow[r] \arrow[d] & \text{present bit} \\ \text{R/W bit set to zero} & \end{tikzcd}$$

Page table reload:

Q) Why is kpgdir[0] zero initially?

This is because although a page table is allocated by setupkvm, its entries are not yet mapped to pages so, entry kpgdir[0] also ~~is~~ has an address value 0x0. stored in it.

Q) How would we translate 0x80106c90 to a physical address?

At the first instruction of kvmalloc, the page table is still not yet allocated, hence we can translate virtual to PA using V2P (virtual to physical) fn.

ie, PA = VA − KERNBASE

= VA − 0x80000000

Q) What is this?

Right shifting 0x80106c90 by 22 bits gives the first 10 bits which tell the index (~~or~~ offset) in the page directory

ie, 0x200 is the offset in the page directory

kpgdir [0x200] gives the address of the page table (2nd level) page.

so, kpgdir[0x200] = 0x3fe007 is the address of page table page

Q) What is the PPN?

Physical page number (PPN) can be obtained as

= Address / page size

For ~~page starting~~ the address 0x3fe007, the PPN is

$$\frac{0x3fe007}{4096} = \underline{\underline{1022}}$$

Q) What does 7 mean?

In 0x3fe007, 7 indicates that the last 3 bits of the entry are 1. This means that

- user is supervisor
- page mode is writable
- page is present.

N TARUN SAI GANESH
2016EE10438

Q) What is this?
(0x80106c90 >> 12) & 0x3ff gives the middle 10 bits.
so, 0x106 is the offset (index) in the page table (2nd level)
0x106001 is the address ~~present~~ value present at that index

Q) Why 1 in the low bits?
1 indicates that the last 4 bits of the entry are 0001.
They mean:
- User is user (not a supervisor)
- page mode is read-only
- page is present.

Q) Why did the physical address in gdb?
This is because paging bit was not enabled in Cr3 register
so physical addresses can still be provided at this point

Q) Why?
After switchkvm, the kpgdir is loaded into Cr3 register and paging bit is enabled. From now, we need to provide virtual addresses only.

Traps: Is it possible to have: ~~two~~
(1) Two "context" structures and one "trapframe" structure on the kstack?
No, because context stores kernel side data of a process & is used during context switching b/w processes. trapframe is used to store user side data of a process & is used ~~to~~ when process transitions from user side to kernel side. Two contexts cannot be stored to same kstack

because interrupts are disabled while pushing context.

(ii) Two trapframes and one context ?

Yes, this happens when a process has already transitioned into Kernel side (ex: syscall) in which case which case there is already a trapframe stored on kstack and then another interrupt occurs (ex: timer interrupt). This again stores another trapframe on kstack & stores the context before switching to another process.

(iii) More than three sets of saved registers ?

No, this is because when a trapframe is stored (i.e. already in Kernel mode), another Interrupt can only be an external interrupt. (XV6 is designed in such a way). Thus when that occurs, interrupts are disabled and no more interrupts occur. Thus a maximum of two trapframes can be stored. Together with one context max, a maximum of 3 sets of saved registers.

Context switching:

Q) on which stack sched() executes on?

sched() is called from the function yield() which runs when a process gets interrupted.

So, sched() executes on the K-stack of the currently interrupted process.

Q) on which stack scheduler() executes ?

scheduler () executes on its own scheduler kstack which is the same as that of the init process (scheduler is called first time from main function).

Q) Does the call to switch() ever return ?

Yes, the switch call returns when the process which is being context switched out is scheduled again. Then it continues executing from switch () statement where the process was context switched out.

Q) Could we reduce the size of a struct context ?

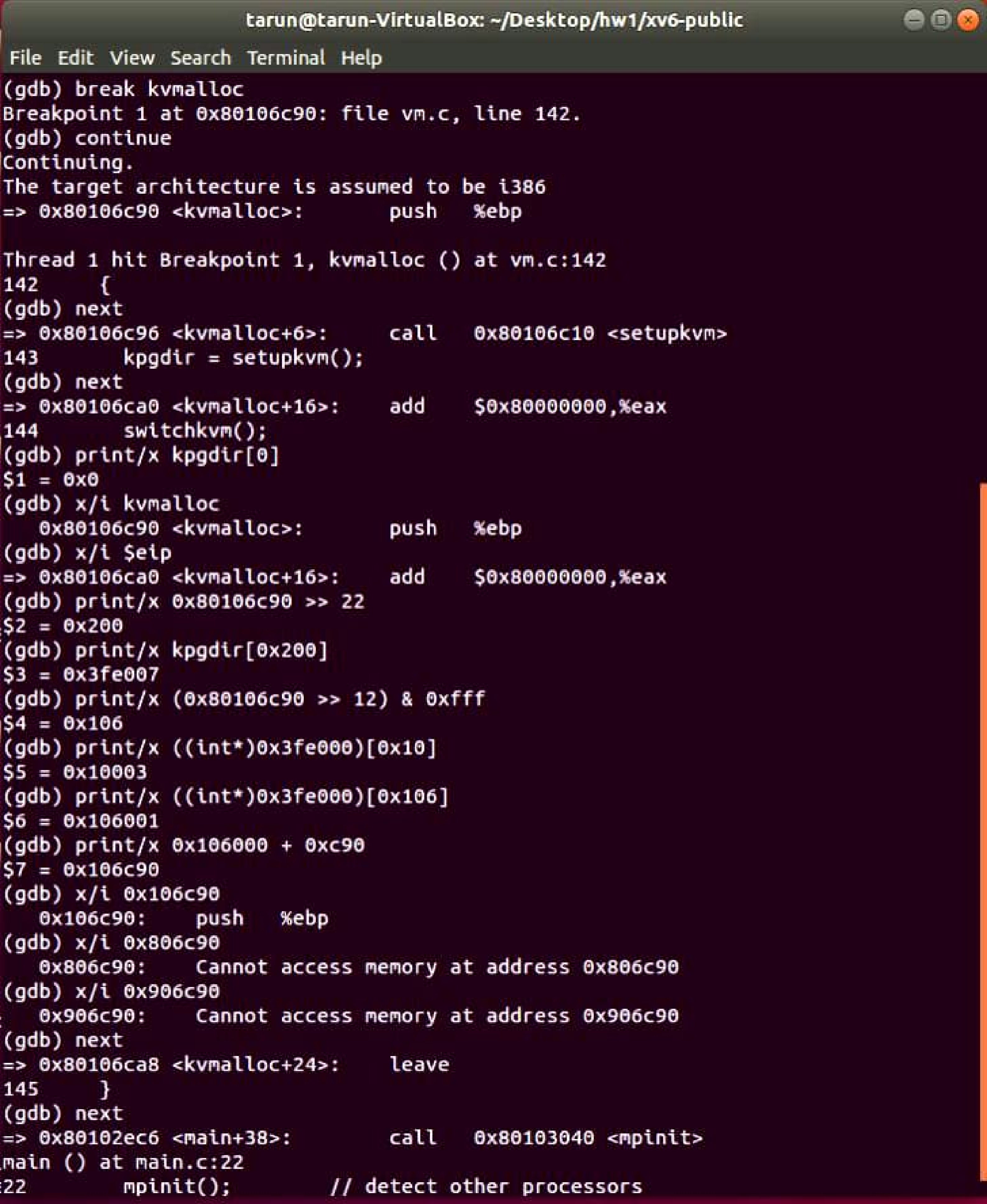
No, we cannot reduce the size further because struct context already stores minimum values reqd.
ebp, ebx, esi, edi are callee save registers which should be saved ~~[illegible]~~ according to GCC calling convns.
Also, eip should be stored. so, storing this much is necessary to follow GCC calling convns. & program to work correctly.

Q) What is the four - character pattern ?

The 4 character pattern printed is "cbad".

Q) Why are first characters "ac" ?

This happens in the beginning because when first time system boots, main function calls scheduler (so "a" printed). Then when a context switch happens, sched gets called (so "c" is printed).


tarun@tarun-VirtualBox: ~/Desktop/hw1/xv6-public
File Edit View Search Terminal Help

```
(gdb) break kvmalloc
Breakpoint 1 at 0x80106c90: file vm.c, line 142.
(gdb) continue
Continuing.
The target architecture is assumed to be i386
=> 0x80106c90 <kvmalloc>:       push   %ebp

Thread 1 hit Breakpoint 1, kvmalloc () at vm.c:142
142     {
(gdb) next
=> 0x80106c96 <kvmalloc+6>:     call   0x80106c10 <setupkvm>
143       kpgdir = setupkvm();
(gdb) next
=> 0x80106ca0 <kvmalloc+16>:    add    $0x80000000,%eax
144       switchkvm();
(gdb) print/x kpgdir[0]
$1 = 0x0
(gdb) x/i kvmalloc
   0x80106c90 <kvmalloc>:       push   %ebp
(gdb) x/i $eip
=> 0x80106ca0 <kvmalloc+16>:    add    $0x80000000,%eax
(gdb) print/x 0x80106c90 >> 22
$2 = 0x200
(gdb) print/x kpgdir[0x200]
$3 = 0x3fe007
(gdb) print/x (0x80106c90 >> 12) & 0xfff
$4 = 0x106
(gdb) print/x ((int*)0x3fe000)[0x10]
$5 = 0x10003
(gdb) print/x ((int*)0x3fe000)[0x106]
$6 = 0x106001
(gdb) print/x 0x106000 + 0xc90
$7 = 0x106c90
(gdb) x/i 0x106c90
   0x106c90:    push   %ebp
(gdb) x/i 0x806c90
   0x806c90:    Cannot access memory at address 0x806c90
(gdb) x/i 0x906c90
   0x906c90:    Cannot access memory at address 0x906c90
(gdb) next
=> 0x80106ca8 <kvmalloc+24>:    leave
145     }
(gdb) next
=> 0x80102ec6 <main+38>:        call   0x80103040 <mpinit>
main () at main.c:22
22        mpinit();        // detect other processors
```