Akanshu Gupta
2016EE10418

Paging ~~Test~~ :

Virtual address = 0x 80100000

$$= 0x\,\underbrace{1000\ 0000\ 00}_{\text{page directory offset}}\underbrace{01\ 0000\ 0000}_{\text{page table offset}}\ \underbrace{0000\ 0\,000\ 0000}_{\text{physical page offset.}}$$

Physical address = 0x 00100000

∴ for page diretory:

offset to be changed = 0x200

value = pge_table address ~~&~~ OR 0x 3

for page table :

offset = 0x100

value = 0x 00100001 (for read-only, kernel access mode).

Page Table Reload :

- print/x kpgdir[0]

  → 0x0

  This zero is because, ~~str~~ setup kvm produce mapping only in kernel part of Virtual address space. i.e. above kernel-base.. All values below are mapped to 0x0 i.e invalid address.

- x/i kvmalloc

Q. how would we translate 0x80107beb to physical address?

A. ~~Linear~~ virtual address is already mapped ~~ide~~ identically to physical address through 4MB pages. We can calculate physical address by subtracting kernel-base

i.e. ~~0x80107beb~~ 0x80106c90 − 0x80000000 = ~~0x00107beb~~ 0x00106c90 (PA).

Q. print/x kpgdir [0x200].

→ 0x3fe007

Q what is this ?

A. Virtual address in my case is 0x80106c90.

0x200 is the first ~~2~~ 10 bits. i.e. pde-offset.

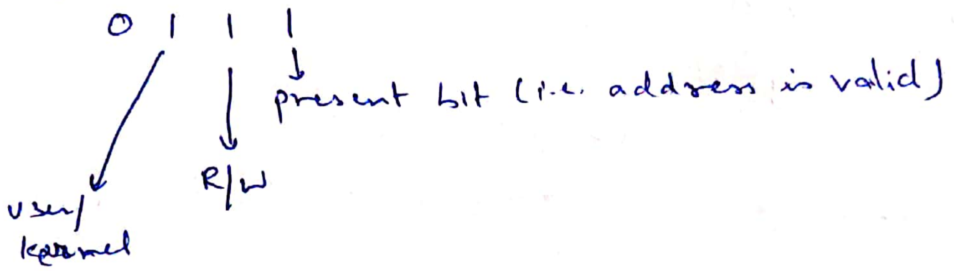
kpgdir[0x200] returns the ~~address~~ physical address of page table. i.e. 0x3fe007 in my case.

Q. what is the physical page number?

A. Physical Page number in my case is
0x3fe000

Q. what does the 7 mean?

A. ~~0~~ 12 bits ~~are~~ represents flags.
So. 0x7 i.e. 0111 represents that Page table is accessible by user, and it is both readable ~~&~~ writable.



(gdb). print/x ((int*)0x3fe000)[0x106]
→ 0x106001

Q what is this?

A. This is physical page number for ~~accessible~~ the virtual address 0x80106c90. we can get the exact physical address at ~~same~~ offset 0xc90 in this page.

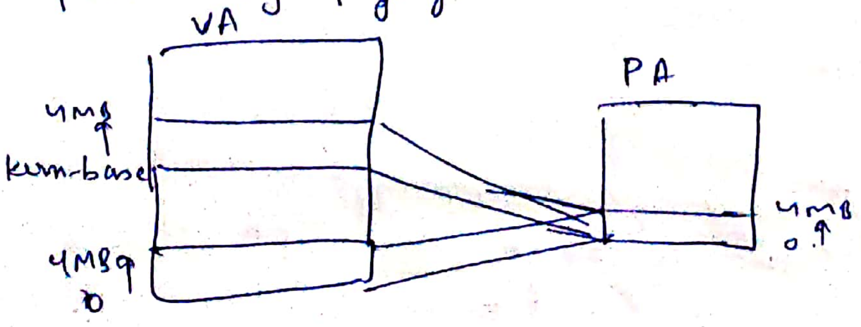
Q. why 1 in the low bits?

A. since requested address lie in kernel space. ie. above kernel-base. so, this physical address is a privileged access.

0001 → ~~u~~kernel privilege, Read only, Present (i.e. address is valid).

Q. why did the physical address work in gdb?

A. Before ~~setup~~ setupkvm was called, we already have mapped. First 4 MB of Physical address to [0-4MB) and [kernel-base, kernelbase + 4MB) in virtual space; ~~to~~, using 4 MB page size mapping. Hence, ~~both~~ Physical address could be parsed by paging.



Q. After switchkvm(), ~~physical~~ direct physical address could not be accessed, why?

A. New page ~~direct~~ mapping has only kernel side mapping in virtual Addresses. ~~hence~~ Addresses below kernel_Base would throw error since they are not mapped linearly to physical addresses.

Addressing:

Q. Suppose you wanted bootmain() to load kernel at 0x80200000 instead of 0x80100000, and you do so by modifying bootmain() to add 0x100000 to va/pa of each ElF section, Something would go wrong. what?

A. Code will throw run-time errors because. ~~these~~ ~~may be~~ functions and variables are named by addresses internally and ~~code~~ ~~of~~ these addresses are already know and stored in code segment. If we change or shift segments, then these pointer will ~~p~~ now point to some garbage value. kernel will not boot at all.

Traps:

Q. xv6 defines two structures that hold saved registers for a process: struct trapframe on sheet 06, and struct context on sheet 20. Explain a situation in which a suspended process will have three sets of saved registers in its kstack.

A. For example, if an exception (page fault, ~~or~~ divide by zero). occurs while running user ~~progr~~ process. and during ~~exception~~. ~~TIMER Interrupt cause context switch~~ exception handling. TIMER Interrupt also occur which cause context switch to another process. In this case, we will have Three sets of saved registers: two trap-frames (one by exception handler + one by TIMER Interrupt) and one context ~~res~~ structure.

Q. Is it possible to have two "context" structures and one "trapframe" structure on the kstack? If so, when? If not, why not?

A. No, we cannot have two context structures because after writing a context structure, control over CPU is passed on to another process. and when same process gets back control, context structure is popped out. So, there is no case in which two context structures can be present.

Q. Is it possible to have 2 "trapframe" and 1 "context" structure on kstack. If so, when? If not, why not?

A. Similar to the ~~[illegible]~~ earlier case, we can have 2 "trapframes". Assume TIMER Interrupt in middle of exception handler. ~~[illegible]~~, context structure will be due to context switch in scheduler called by TIMER Interrupt Handler.

Q. Is it possible to have more than 3 sets of saved registers in the kstack? If so, when? If not, why?

A. Yes. it is possible to have more than 3 sets of saved register in case there are nested interrupts. But as a design choice and to impose a bound on kstack size. OS's generally donot allow more than two level of nested interrupts.. which limits the number of set of saved registers by 3..

Context Switching:

Q. Where is the stack that sched() executes on?

A. sched() run on kernel stack of current process.
sched() is just a function ~~call~~ running in kernel mode.

Q. Where is the stack that scheduler() executes on?

A. scheduler() is a separate process it has its own kernel stack. It runs on its own stack inside kernel mode

Q. when sched() calls switch(), does call to sched() ever return. If so when?

A. sched() suspends current running process and transfer control to another process, when it gets control back, same state is revived and then sched() will return.

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

A. swtch is a function call. as per GCC convention it needs to maintain the callee saved register. in the stack. swtch can do the less work and still be correct as long as the new process that we have switched to, does not change these registers. or the caller function does not need these registers' ~~original~~ old values.

Size of struct context could be reduced ~~of~~ and it can have only one pointer pointing to the top of ~~cont~~ stack instead of set of registers. All addresses above it will still be valid and can be accessed by adding ~~of~~ respective offset.

Q. What is the four ~~patt~~ character pattern?

A. ~~badcba~~ badc. will be the repeating pattern.

Q. The very first characters are ac. why does this happen?

A. "a" represents ~~st~~ process got suspended and control is transferred to scheduler

~~d~~"c" represents scheduler ~~sto~~ has called switch().

when first process calls sched(), "a" gets printed and now since scheduler is called for the first time it prints "c". before ~~wiring~~ doing switch().

when scheduler gets back control, it prints "d" and ~~a~~ then "c" that represents context switch to another process.. when that process gets back into running. it prints "b" as part of sched() code. and ~~agai~~ then "a" to pass control to scheduler.

This cycle continues. and we get

acbadcbadcbadc....

ba → one process has run and now ~~bac~~ control is transferred to scheduler

dc → scheduler has scheduled next process and handed the control to it.