OS Homework -3

2016CS10324
R Jayanth Reddy

1. Paging

The given virtual address is 1000 0000 0001 0000 0000 0000 0000 0000 in binary, the linear address is also the same since the base is 0.

The first ten bits i.e 1000000000 (its 512 in decimal) gives offset of the entry in page directory that needs to be modified.This PDE entry contains PTPPN bbbg 0010 0101 (PTPPN - first 20bits is page table PPN)
Here first twenty bits represent page table physical page number which can't be obtained from given information , bbb -used by os , g - ignored, I have set bits based on assumption that page is cached , read only, user mode, page is present in memory.

The next ten digits i.e 0100000000 (its 256 in decimal) gives offset of the entry in page table that needs to be modified. This PTE entry contains 0000 0000 0001 0000 0000 bbb0 0011 0101
Here first twenty bits represent physical page number which is obtained from physical address, bbb -used by os, I have set bits based on assumption that page is not dirty , its cached , read only, user mode, page is present in memory.

2. Page table reload

kpgdir[0] is not mapped as it is general convention to follow that the 0th address is not mapped and it acts like null. This can also be observed from setupkvm code where kpgdir is made zero and starting address is not mapped. (this is not the same in case of entrypgdir)

v.a : 0x80107beb , as we didn't execute switchkvm() yet, we need to use entrypgdir to get p.a so p.a = v.a - 2GB for this 0x80107beb i.e p.a = 0x00107beb

```
(gdb) print/x 0x80107beb >> 22
$4 = 0x200
(gdb) print/x kpgdir[0x200]
```

$6 = 0x114007

0x200 is the first ten bits of address 0x80107beb.
kpgdir[0x200] gives the 512th entry in the page directory.
The first 20 bits of the entry i.e 0x00114007 gives us p.a of page table which has an entry pointing to p.a of 0x80107beb. So 0x00114 (276 in decimal) is page table PPN and p.a is 0x00114000 for page table.
The last 12 bits of 0x00114007 denote flags and especially 0111 as the last 4 bits means that it has write-back policy, user mode , read/write enabled and page is present in memory.

(gdb) print/x (0x80107beb >> 12) & 0xfff
$6 = 0x107
(gdb) print/x ((int*)0x114000)[0x107]
$12 = 0x107001
0x107 are the next 10 bits of 0x80107beb which can be used to find corresponding entry in page table.
As 0x114000 is page table address as described previously, ((int*)0x114000)[0x107] gives entry in the page table to find the physical address of 0x80107beb.
The first 20 bits of 0x00107001 denote PPN of the physical address of 0x80107beb and the next 12 bits denote flags. So 0x107000 + 0xbeb(offset last 12 bits of v.a) = 0x107beb is the actual physical address of 0x80107beb.
0001 in flags denote that it has write-back policy, privilege mode instr, read only enabled and page is present in memory.

gdb) x/i 0x107beb
Why did the physical address work in gdb?

As switchkvm() is not executed and entrypgdir is still being used in which the addresses which correspond to physical addresses in virtual address space are mapped to the same physical address space.
i.e 0 to 4MB in v.a space also 2GB to 2GB +4MB in v.a space both are mapped to 0 to 4MB in p.a space

now 0x107beb won't work:
(gdb) x/i 0x107beb
0x107beb: Cannot access memory at address 0x107beb

As switchkvm() is executed, kpgdir is a new page table in which all memory below the kernel base is used by user processes and it is not mapped yet.

3. Addressing

If we add 0x100000 to the va/pa of each ELF section, then we may have used the last 1MB of pa/va space which is actually allocated to I/O devices.

4. Traps

The process makes a system call and the user registers are saved in kstack as "trapframe" struct. Now just when this trapret (function returning to user mode by popping data in kstack) begins if we have timer interrupt , the kernel registers would be saved below the previous trapframe as "trapframe" struct. Now in handling this timer interrupt if yield() is called (or scheduler determines a context switch) we would have kernel registers saved in "context" struct and the process is suspended.

We can't have two "context" structures and one "trap frame" structure in kstack because "context"
structures are required when there is context switch , but when it returns to this process after some time, context structure present in it will be popped out and we can never have two context structures.

We can have two trapframe structures and one context structure as described earlier.

We can't have more than three sets of saved registers in the kstack due to conventions followed in xv6 to avoid stack overflow of 4KB stacks.
- Ensure that the kernel cannot cause any software interrupt or
  exception while executing a handler.
- Ensure that handlers of external interrupts (e.g., timer, disk, network, etc.)
  always execute with interrupts disabled.

5. Context switching

sched() executes on the process's kernel stack from which it is called.
scheduler() executes on a scheduler kernel stack which is already present in the kernel.
When the swtch we have been tracing returns, it returns not to sched but to scheduler, and it's stack pointer points at the current CPU’s scheduler stack. Also similar thing happens in swtch()

of scheduler() and then program after sched()'s swtch is run. (So it actually returns but not in usual sense)

We can actually get rid of ebp(frame pointer) in struct context and thus avoid storing it by swtch(). We already have esp to get stack pointer and ebb is not necessary in context switch also.

```
# Switch stacks
 movl %esp, (%eax)
 movl %edx, %esp
```

The above code would ensure esp pointing to appropriate places in new stack and ebp is not necessary.

"badc" is the repeating four character pattern.
The very first characters are "ac" as the main() calls mpmain() which is a first calls scheduler(). Then the process scheduled (due to context switch by swtch(&(c−>scheduler), p−>context)) by scheduler() first calls sched(). Hence first "a" (in scheduler()) and then "c" (in sched()) is printed.