xv6 Boot Recap:
Transitioning from 16 bit mode to 32 bit mode

3 November 2018
Aftab Hussain
University of California, Irvine
what it does

Sets up the hardware.
Transfers control to the Boot Loader.
what it does

Sets up the hardware.
Transfers control to the Boot Loader.

how it transfers control to the Boot Loader

Boot loader is loaded from the 1st 512-byte sector of the boot disk.
This 512-byte sector is known as the boot sector.
Boot loader is loaded at 0x7c00.
Sets processor’s ip register to 0x7c00.
BIOS → xv6 Boot loader

2 source source files
  bootasm.S - 16 and 32 bit assembly code.
  bootmain.c - C code.
2 source source files
   bootasm.S - 16 and 32 bit assembly code.
   bootmain.c - C code.

executing bootasm.S
1. **Disable interrupts** using `cli` instruction. ([Code](#)).
   > Done in case BIOS has setup any of its interrupt handlers were initialized while setting up the hardware. Also, BIOS is not running anymore, so better to disable them.
   > Clear segment registers. Use xor for `%ax`, and copy it to the rest ([Code](#)).
2. **Switch from real mode to protected mode.** ([References](#): a, b).
   > Note the difference between processor modes and kernel privilege modes.
   > We do the above switch to increase the size of the memory we can address.
2. Let's expand on this a little bit

Switch from real mode to protected mode.

executing bootasm.S
Addressing in Real Mode
In real mode, the processor sends 20-bit addresses to the memory. (e.g. Intel processors 8086, 8088).

However, it has eight general 16-bit registers + 16 bit segment registers.

How to generate a 20 bit address from a 16 bit register?
Say processor fetches a data read/write instruction. The processor would then use the data segment register (%ds).

To pass the 16-bit address obtained from %ds to the memory (which accepts 20-bit addresses) we left shift the 16-bit register by 4 bits.

This is equivalent to adding the register to itself 16 times.
(try it out in a bit calculator - e.g. gnu calculator)
Say processor fetches a data read/write instruction. The processor would then use the data segment register (%ds).

To pass the 16-bit address obtained from %ds to the memory (which accepts 20-bit addresses) we left shift the 16-bit register by 4 bits.

This is equivalent to **adding the register to itself 16 times**.

(try it out in a bit calculator - e.g. gnu calculator)
So let’s look at how the address translation process takes place in real mode...
CPU

- 16 bits
- segment selector

- 16 bits
- offset
logical address
(also known as a 32-bit segment:offset pair)
16 bits offset
16 bits segment selector

logical address (also known as a 32-bit segment:offset pair)

CPU

xv6 refers to this x86 logical address as a virtual address
16 bits
offset

CPU

Segment translation hardware

32 bits
16 bits
segment selector

16 bits
offset

logical address
(also known as a 32-bit segment:offset pair)
The logical address (also known as a 32-bit segment:offset pair) is composed of a segment selector and an offset. The segment selector is 16 bits long, and the offset is 16 bits long. The logical address is then passed to the segment translation hardware, which performs a left shift by 4 bits. The result is a 32-bit address that is used to access memory.
16 bits
offset
CPU

Segment translation hardware

20 bits
linear address

32 bits
16 bits
segment selector
16 bits
offset

Segment selector
Segment selector
Segment selector
left shift by 4
add

logical address
(also known as a 32-bit segment:offset pair)
16 bits offset

CPU

Segment translation hardware

32 bits

16 bits

segment selector

offset

logical address (also known as a 32-bit segment:offset pair)

directly corresponds to the physical address

20 bits

linear address

Segment translation hardware

16 bits

segment selector

20 bits

left shift by 4

add

16 bits

segment selector

16 bits

offset

32 bits

da
dirrectly corresponds to the physical address
if paging is enabled, this address would go through a further translation process within the paging hardware to generate a physical address.
xv6 configures this hardware such that logical and linear addresses are always the same.
It follows (without paging) in xv6, logical address = linear address = physical address.

xv6 configures this hardware such that logical and linear addresses are always the same.
Why the switch to Protected Mode?
with **20 bit addresses** the maximum size of addressable memory is \(2^{20}\) bytes which is only **1MB**.

In Protected Mode, we can address **32 bit addresses**, which allows us to address a memory of size **4GB**.

Paging hardware is also enabled in Protected Mode.
2. Switch from real mode to protected mode.
   > To setup the protected mode, the boot loader sets up the segment descriptor table, with three entries. Each entry is [a base physical address, max virtual address limit (4GB), permission bits]
executing bootasm.S

lgdt gdtdesc

gdtdesc:
.word (gdtdesc - gdt - 1)  # sizeof(gdt) - 1
.long gdt  # address gdt

the macros used are defined here

the gdt table

BIOS —→ xv6 Boot loader
BIOS → xv6 Boot loader

executing bootasm.S

Assembly Tip: All assembler directives begin with a period. (Ref.)

setting up segment descriptor table

gdtdesc:

```
lgdt gdtdesc
(gdtdesc - gdt - 1)  # sizeof(gdt) - 1
.long gdt  # address gdt
```

the gdt table

the macros used are defined here
2. **Switch from real mode to protected mode.**
   > Now enable protected mode by setting the 1 bit in control register %cr0. *(Code)*.
2. **Switch from real mode to protected mode.**
   > To fully enable protected mode, we need to load a new value into `%cs`.
   > Since `%cs` cannot be modified directly we setup `%cs` here: (Code), which results in `%cs` to refer to the code descriptor entry in gdt.
executing bootasm.S

2. **Switch from real mode to protected mode.**
   > To fully enable protected mode, we need to load a new value into %cs.
   > Since %cs cannot be modified directly we `ljmp (code)`, which results in %cs to refer to the code descriptor entry in gdt (a 32 bit code seg.)

**Assembly Tip:**
Long Jump:
```
ljmp $0xfebc, $0x12345678
```

Use 0xfebc for the CS register and 0x12345678 for the EIP register. ([Ref.](#))
We have completed transitioning from 16 bit (real) mode to 32 bit (protected) mode
To see details on how the 32 bit addresses are translated in the segmentation hardware in protected mode look [here](http://example.com).