143A: Principles of Operating Systems

Lecture 7: System boot

Anton Burtsev
October, 2017
Outline for today

Boot operating system

- Setup segments (data and code)
- Switch to protected mode
  - Load GDT (turn segmentation on)
- Setup stack (needed to call C functions)
- Load the kernel from disk into memory
- Setup first page table
  - 2 entries [ 0 : 4MB ] and [ 2GB : (2GB + 4MB) ]
- Setup high-address stack
- Jump to main()
  - Start executing kernel code
What happens when we turn on the power?

● Well it's complicated
  ● Intel SGX Explained is a good start (Section 2.13 [1])
● At a high-level a sequence of software pieces initializes the platform
  ● Microcode, firmware (BIOS), bootloader
• The most important thing, the OS is not the only software running on the machine
  • And not the most privileged
• Today, at least two layers sit underneath the OS/hypervizor
  • System Management Mode (SMM) (ring -2)
    – Runs below the hypervisor/OS
  • Intel Management Engine (ring -3)
    – Runs on a separate CPU
Bootloader starts

Physical memory layout:
- Bootbock: 512B
- Memory map: 0x7c00 to 0x7d00

Real Mode:
- CS: 0x0
- SS: 0x0
- GDT: 0x0
- IDT: 0x0
- EIP: 0x7c00
- ESP: 0x0
- TSS: 0x0

512MB physical memory range.
Bootloader starts

9111 start:
9112  cli # BIOS enabled interrupts; disable
9113
9114  # Zero data segment registers DS, ES, and SS.
9115  xorw %ax,%ax # Set %ax to zero
9116  movw %ax,%ds # -> Data Segment
9117  movw %ax,%es # -> Extra Segment
9118  movw %ax,%ss # -> Stack Segment
Why start happens to be 0x7c00?

9111 start:

9112 cli # BIOS enabled interrupts; disable

9113
Linker are told so through the Makefile

9111 start:

9112 cli # BIOS enabled interrupts; disable

9113

bootblock: bootasm.S bootmain.c

$(CC) $(CFLAGS) -fno-pic -O -nostdinc -I. -c bootmain.c
$(CC) $(CFLAGS) -fno-pic -nostdinc -I. -c bootasm.S
$(LD) $(LDFLAGS) -N -e start -Ttext 0x7C00 -o bootblock.o bootasm.o bootmain.o
$(OBJDUMP) -S bootblock.o > bootblock.asm
$(OBJCOPY) -S -O binary -j .text bootblock.o bootblock
./sign.pl bootblock
Switch to protected mode

- Switch from real to protected mode
  - Use a bootstrap GDT that makes virtual addresses map directly to physical addresses so that the effective memory map doesn’t change during the transition.

```汇编
9141 lgdt gdtdesc
9142 movl %cr0, %eax
9143 orl $CR0_PE, %eax
9144 movl %eax, %cr0
```
How GDT is defined

9180  # Bootstrap GDT
9181  .p2align 2  # force 4 byte alignment
9182  gdt:
9183   SEG_NULLASM  # null seg
9184   SEG_ASM(STA_X|STA_R, 0x0, 0xffffffff)  # code seg
9185   SEG_ASM(STA_W, 0x0, 0xffffffff)  # data seg
9186
9187  gdtdesc:
9188   .word (gdtdesc - gdt - 1)  # sizeof(gdt) - 1
9189   .long gdt
How GDT is defined

9180 # Bootstrap GDT
9181 .p2align 2 # force 4 byte alignment
9182 gdt:

9183 SEG_NULLASM # null seg
9184 SEG_ASM(STA_X|STA_R, 0x0, 0xffffffff) # code seg
9185 SEG_ASM(STA_W, 0x0, 0xffffffff) # data seg

9186

9187 gdtdesc:

9188 .word (gdtdesc - gdt - 1) # sizeof(gdt) - 1
9189 .long gdt
Load GDT

bootbock
512B

Physical

0 0x7000 0x7d00

512MB

Real Mode

CS: 0x0
SS: 0x0
GDT: 0x7c78
IDT: 0x0

FIP: 0x7c1d
ESP: 0x0
TSS: 0x0

GDT

NULL: 0x0
CODE: 0 - 4GB
DATA: 0 - 4GB
Actual switch

- Use long jump to change code segment

9153 ljmp $(SEG_KCODE<<3), $start32

- Explicitly specify code segment, and address

- Segment is 0b1000 (0x8)
Why CS is 0x8, not 0x1?

- Segment selector:

  ![Segment Selector Diagram]

  - Table Indicator:
    - 0 = GDT
    - 1 = LDT
  - Requested Privilege Level (RPL)
.code32 # Tell assembler to generate 32-bit code now.

start32:

# Set up the protected-mode data segment registers
movw $(SEG_KDATA<<3), %ax # Our data segment selector
movw %ax, %ds # -> DS: Data Segment
movw %ax, %es # -> ES: Extra Segment
movw %ax, %ss # -> SS: Stack Segment
movw $0, %ax # Zero segments not ready for use
movw %ax, %fs # -> FS
movw %ax, %gs # -> GS
Segments

Linear

Code

Data

bootbocK
512B

Physical

0

0x7c00

0x7d00

GDT

NULL: 0x0
CODE: 0 - 4GB
DATA: 0 - 4GB

Protected Mode

CS: 0x8
SS: 0x10
GDT: 0x7c78
IDT: 0x0

EIP: 0x7c1d
ESP: 0x0
TSS: 0x0

512MB

4GB
Setup stack

- Why do we need a stack?

```
9166 movl $start, %esp
9167 call bootmain
```
Setup stack

- Need stack to use C
  - Function invocations
  - Note, there were no stack instructions before that

9166 movl $start, %esp

9167 call bootmain
First stack

Linear

Stack

Code

Data

Physical

0x7c00

0x7d00

GDT

NULL: 0x0
CODE: 0 - 4GB
DATA: 0 - 4GB

CS: 0x8
SS: 0x10
GDT: 0x7c78
IDT: 0x0

EIP: 0xc1d
ESP: 0x7c00
TSS: 0x0

Protected Mode
Invoke first C function

9166 movl $start, %esp

9167 call bootmain
bootmain(): read kernel from disk

```c
void bootmain(void) {
    struct elfhdr *elf;
    struct proghdr *ph, *eph;
    void (*entry)(void);
    uchar* pa;

    elf = (struct elfhdr*)0x10000; // scratch space

    // Read 1st page off disk
    readseg((uchar*)elf, 4096, 0);

    // Is this an ELF executable?
    if(elf->magic != ELF_MAGIC) {
        return; // let bootasm.S handle error
    }
}
```
// Load each program segment (ignores ph flags).
ph = (struct proghdr*)((uchar*)elf + elf->phoff);
eph = ph + elf->phnum;
for(; ph < eph; ph++){
    pa = (uchar*)ph->paddr;
    readseg(pa, ph->filesz, ph->off);
    if(ph->memsz > ph->filesz)
        stosb(pa + ph->filesz, 0, ph->memsz - ph->filesz);
}

// Call the entry point from the ELF header.
// Does not return!
entry = (void(*)(void))(elf->entry);
entry();

bootmain(): read kernel from disk
How do we read disk?

// Read a single sector at offset into dst.
void readsect(void *dst, uint offset)
{
    // Issue command.
    waitdisk();
    outb(0x1F2, 1); // count = 1
    outb(0x1F3, offset);
    outb(0x1F4, offset >> 8);
    outb(0x1F5, offset >> 16);
    outb(0x1F6, (offset >> 24) | 0xE0);
    outb(0x1F7, 0x20); // cmd 0x20 - read sectors

    // Read data.
    waitdisk();
    insl(0x1F0, dst, SECTSIZE/4);
}

How do we read disk (cont)?

9250  void
9251  waitdisk(void)
9252  {
9253       // Wait for disk ready.
9254       while((inb(0x1F7) & 0xC0) != 0x40)
9255          ;
9256  }
9257
By convention, the _start symbol specifies the ELF entry point. Since we haven’t set up virtual memory yet, our entry point is the physical address of ’entry’.

 global _start
 _start = V2P_W0(entry)

 # Entering xv6 on boot processor, with paging off.
 global entry
 entry:
 # Turn on page size extension for 4Mbyte pages
    movl %cr4, %eax
    orl $(CR4_PSE), %eax
    movl %eax, %cr4
Set up page directory

1149  # Set page directory
1150  movl  $(V2P_WO(entrypgdir)),  %eax
1151  movl  %eax,  %cr3
First page table

- Two 4MB entries (large pages)
- Entry #0
  - 0x0 – 4MB → 0x0:0x400000
- Entry #960
  - 0x0 – 4MB → 0x8000000:0x80400000
The boot page table used in entry.S and entryother.S.

Page directories (and page tables) must start on page boundaries,

hence the __aligned__ attribute.

PTE_PS in a page directory entry enables 4Mbyte pages.

__attribute__((__aligned__((___aligned__((PGSIZE)))

pde_t entrypgdir[NPENTRIES] = {

// Map VA’s [0, 4MB) to PA’s [0, 4MB)
[0] = (0) | PTE_P | PTE_W | PTE_PS,

// Map VA’s [KERNBASE, KERNBASE+4MB) to PA’s [0, 4MB)
[KERNBASE>>PDXSHIFT] = (0) | PTE_P | PTE_W | PTE_PS,

};
First page table (cont)

0870 // Page directory and page table constants.

0871 #define NPDENTRIES 1024
# First page table

## Linear

<table>
<thead>
<tr>
<th>Stack</th>
<th>Kernel</th>
</tr>
</thead>
<tbody>
<tr>
<td>Code</td>
<td></td>
</tr>
<tr>
<td>Data</td>
<td></td>
</tr>
</tbody>
</table>

0 4MB 2GB 4GB

## Virtual

0 4MB 2GB 2GB+4MB

## Physical

0 0x7c00 0x7d00 0x100000

<table>
<thead>
<tr>
<th>Page table</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 - 4MB</td>
</tr>
<tr>
<td>0x0</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>2GB - 2GB + 4MB</td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Protected Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>CS: 0x8</td>
</tr>
<tr>
<td>SS: 0x10</td>
</tr>
<tr>
<td>GDT: 0x7c78</td>
</tr>
<tr>
<td>IDT: 0x0</td>
</tr>
<tr>
<td>GDT</td>
</tr>
<tr>
<td>NULL: 0x0</td>
</tr>
<tr>
<td>CODE: 0 - 4GB</td>
</tr>
<tr>
<td>DATA: 0 - 4GB</td>
</tr>
</tbody>
</table>

Protected Mode

---

EIP: 0x10001a
ESP: 0x7c00
TSS: 0x0
CR3: entrypgdir
Turn on paging

1152 # Turn on paging.
1153 movl %cr0, %eax
1154 orl $(CR0_PGE|CR0_WP), %eax
1155 movl %eax, %cr0
High address stack (4K)

1157  # Set up the stack pointer.
1158        movl $(stack + KSTACKSIZE), %esp
1159
1159 ...
1167  .comm stack, KSTACKSIZE

0151  #define KSTACKSIZE 4096  // size of per-process kernel stack
High address stack (4K)

Linear

Stack

Kernel

Virtual

Physical

Page table

GDT

Protected Mode

CS : 0x8
SS : 0x10
GDT: 0x7c78
IDT: 0x0
EIP: 0x10001a
ESP: stack

0 - 4MB
0x0
...
2GB - 2GB + 4MB
...

NULL: 0x0
CODE: 0 - 4GB
DATA: 0 - 4GB
Jump to main()

1160 # Jump to main(), and switch to executing at
1161 # high addresses. The indirect call is
needed because
1162 # the assembler produces a PC-relative
instruction
1163 # for a direct jump.
1164 mov $main, %eax
1165 jmp *%eax
1166
Running in main()

1313 // Bootstrap processor starts running C code here.
1314 // Allocate a real stack and switch to it, first
1315 // doing some setup required for memory allocator to work.
1316 int
1317 main(void)
1318 {
1319    kinit1(end, P2V(4*1024*1024)); // phys page allocator
1320    kvmalloc(); // kernel page table
1321    mpinit(); // detect other processors
1322    lapicinit(); // interrupt controller
1323    seginit(); // segment descriptors
1324    cprintf("\ncpu%d: starting xv6\n", cpunum());
1325    ...
1340 }
References

Questions?