143A: Principles of Operating Systems

Lecture 7: System boot

Anton Burtsev
January, 2017
Bootloader starts

Physical

bootbock
512B

Real Mode

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

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

512MB
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
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.

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
Load GDT

bootbock
512B

Physical

0 0x7c00 0x7d00

512MB

GDT

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

Real Mode

CS : 0x0  FIP: 0x7c1d
SS : 0x0   ESP: 0x0
GDT: 0x7c78  TSS: 0x0
IDT: 0x0
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)
Long jump

bootbock
512B

Physical

Protected Mode

GDT

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

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

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

0x7c00
0x7d00
0
512MB

NULL: 0x0
CODE: 0 - 4GB
DATA: 0 - 4GB
Why CS is 0x8, not 0x1?

- Segment selector:

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
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

GDT

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

Protected Mode

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

EIP: 0x7c1d
ESP: 0x7c00
TSS: 0x0

Linear

Stack

Code

Data

Physical

GDT

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

Protected Mode

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

EIP: 0x7c1d
ESP: 0x7c00
TSS: 0x0
bootmain(): read kernel from disk
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.

```
entry = (void(*)(void))(elf->entry);
entry();
```
How one reads 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);
}

void waitdisk(void)
{
    // Wait for disk ready.
    while((inb(0x1F7) & 0xC0) != 0x40)
    ;
}
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’. 

Entering xv6 on boot processor, with paging off.

# Turn on page size extension for 4Mbyte pages
movl %cr4, %eax
orl $(CR4_PSE), %eax
movl %eax, %cr4

entry(): kernel ELF entry
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
  - $0x0 - 4MB \rightarrow 0x0:0x400000$
- Entry #960
  - $0x0 - 4MB \rightarrow 0x8000000:0x80400000$
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__(PGSIZE)))
pde_t entrypgdir[NPDENTRIES] = {
    // 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
First page table (cont)

0870 // Page directory and page table constants.

0871 #define NPDENTRIES 1024
# Turn on paging.

movl %cr0, %eax

orl $(CR0_PG|CR0_WP), %eax

movl %eax, %cr0
# Set up the stack pointer.

movl $(stack + KSTACKSIZE), %esp

... 

.comm stack, KSTACKSIZE

#define KSTACKSIZE 4096 // size of per-process kernel stack
Jump to main()

1160 # Jump to main(), and switch to executing at 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
// Bootstrap processor starts running C code here.
// Allocate a real stack and switch to it, first
// doing some setup required for memory allocator to work.

int main(void)
{
    kinit1(end, P2V(4*1024*1024)); // phys page allocator
    kvmalloc(); // kernel page table
    mpinit(); // detect other processors
    lapicinit(); // interrupt controller
    seginit(); // segment descriptors
    cprintf("\\ncpu%d: starting xv6\\n\\n", cpunum());
    ...
}

Running in main()
Questions?