143A: Principles of Operating Systems

Lecture 6: System boot

Anton Burtsev
October, 2019
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/hypervisor
  • System Management Mode (SMM) (ring -2)
    − Runs below the hypervisor/OS
  • Intel Management Engine and Intel Innovation Engine (ring -3)
    − Run on separate CPUs
PC motherboard components

Diagram showing PC motherboard components, including CPUs, DRAM, FLASH, UEFI, ME FW, SPI, USB, SATA, PCH, ME, NIC/PHY, QPI, DDR, PCIe, DMI.
I/O Devices
Dell R830 4-socket server

Dell Poweredge R830 System Server with 2 sockets on the main floor and 2 sockets on the expansion

Multi-socket machines
PC motherboard components
PCH – Platform Controller Hub
ME gets power before CPUs
Intel Management Engine (ME)

- Full-featured computer
  - Intel Quark x86-based 32-bit CPU
  - Internal RAM (1.7MB)
  - Can access all DRAM via DMA
  - Can control boot chain
  - Can access network interface (NIC) on the motherboard
    - Has it’s own MAC and IP address
    - Via System Management Bus (SMBus)
    - Or an ATM compatible NIC
- Connected to the power supply
  - Stays on as long as power is provided to power supply
ME: Theft prevention use-case

• In S5 (computer off) ME cannot access DRAM
  • DRAM is off
  • But ME can use its internal memory
  • ME can disable a stolen laptop equipped with cellular modem remotely
    – As long as power is connected
    – And cell network has signal
Intel Management Engine (ME)

• All modern Intel CPUs contain ME
  • Part of Active Management Technology (AMT)
  • Convenient way for administrators to fix your machine remotely
    – Obviously a huge opportunity for an attack
ME starts first

- Reads its initialization code from the BIOS chip
  - Via the SPI bus
Bootstrap processor (BSP)

- One of the logical processors is chosen as bootstrap processor (BSP)
  - Will start initialization
- Others become “application processors” (AP)
  - Waiting for a special interrupt from the BSP
BSP starts reading BIOS

- Executes instructions stored in the BIOS chip
- An interesting detail is that BSP starts with DRAM disabled
  - Hence there is no stack to call functions
  - What can be done?
BSP starts without DRAM

- Custom-written assembly code that uses no stack
- Or a ROMCC compiler
  - Generates code from C that uses no stack
  - Used in the coreboot project
Cache-as-RAM

• Use CPU caches as temporary replacement for RAM
  • Initialize DRAM
  • Copy BIOS firmware into DRAM and continue
BIOS firmware

• Initialize
  • Interrupt controllers
  • Devices, e.g., network interfaces
  • If one of PCI devices contains “option ROM” load and execute it
    – Network cards may contain iPXE ROM
    – Implement boot from the network host
System Management Mode

- Another compartment that runs underneath your OS or hypervisor
- Initialized by BIOS
- Protected with hardware memory mechanisms
  - OS cannot access this region of memory
- Runs under your OS or hypervisor
  - Receives interrupts periodically, can take over the entire system any time
  - Impossible to disable
BIOS loads the boot loader

• BIOS ends by loading a boot loader
  • Modern BIOSes can load the boot loader from a variety of sources (hard disks, USB drives, optical disks)
  • Default way is to load the first sector (512 bytes) from disk into the memory location at 0x7c00
  • BIOS then starts executing instructions at the address 0x7c00
    - This is exactly what we see when we run xv6 under QEMU
    - QEMU emulates hardware: runs BIOS, follows the same protocol
BIOS loads bootloader

Physical memory layout:

- bootbock: 512B
- Physical memory from 0x0 to 512MB
- CS: 0x0, EIP: 0x7c00
- SS: 0x0, ESP: 0x0
- GDT: 0x0, TSS: 0x0
- IDT: 0x0

Real Mode
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

xv6/bootasm.S
Why start happens to be 0x7c00?

9111 start:

9112 cli # BIOS enabled interrupts; disable

9113

xv6/bootasm.S
Linker is instructed to link the boot block code in 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

xv6/Makefile
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
```

xv6/bootasm.S
Load GDT

Physical

bootbock
512B

0 0x7c00 0x7d00

512MB

GDT

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

Real Mode

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

FIP: 0x7c1d
ESP: 0x0
TSS: 0x0
Recap: complete address translation
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, 0xfffffffff)  # code seg
9185      SEG_ASM(STA_W, 0x0, 0xfffffffff)  # data seg
9186
9187  gdtdesc:
9188      .word (gdtdesc - gdt - 1)  # sizeof(gdt) - 1
9189      .long gdt

xv6/bootasm.S
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

xv6/bootasm.S
Actual switch

- Use long jump to change code segment

9153 1jmp $(SEG_KCODE<<3), $start32

- Explicitly specify code segment, and address

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

- Segment selector:

  15
  |
  3   2   1   0
  |
  Index  T   RPL

Table Indicator
0 = GDT
1 = LDT

Requested Privilege Level (RPL)
Long jump

Physical

bootbock
512B

0
0x7c00
0x7d00

512MB

GDT

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

Protected Mode

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

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

9155  .code32  # Tell assembler to generate 32-bit code now.
9156  start32:
9157  # Set up the protected-mode data segment registers
9158  movw $(SEG_KDATA<<3), %ax  # Our data segment selector
9159  movw %ax, %ds  # −> DS: Data Segment
9160  movw %ax, %es  # −> ES: Extra Segment
9161  movw %ax, %ss  # −> SS: Stack Segment
9162  movw $0, %ax  # Zero segments not ready for use
9163  movw %ax, %fs  # −> FS
9164  movw %ax, %gs  # −> GS
Setup stack

• Why do we need a stack?

9166 movl $start, %esp
9167 call bootmain

xv6/bootasm.S
Setup stack

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

9166 movl $start, %esp
9167 call bootmain

xv6/bootasm.S
First stack

Linear Stack

Code

Data

Physical Stack

0x7c00

0x7d00

GDT

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

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

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

Protected Mode
Invoke first C function

9166 movl $start, %esp
9167 call bootmain
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?

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);
}

xv6/bootmain.c
9250 void
9251 waitdisk(void)
9252 {
9253     // Wait for disk ready.
9254     while((inb(0x1F7) & 0xC0) != 0x40)
9255     ;
9256 }
9257
xv6/bootmain.c
// 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();
}

Call kernel entry
# 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’.

.globl _start
_start = V2P_WO(entry)

# Entering xv6 on boot processor, with paging off.
.globl entry
.entry:

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

entry(): kernel ELF entry
1039 .globl entry
1136 # By convention, the _start symbol specifies the ELF entry point.
1137 # Since we haven’t set up virtual memory yet, our entry point is
1138 # the physical address of ’entry’.
1139 .globl _start
1140 _start = V2P_W0(entry)
1141
1142 # Entering xv6 on boot processor, with paging off.
1143 .globl entry
1144 entry:
1145 # Turn on page size extension for 4Mbyte pages
1146    movl %cr4, %eax
1147    orl $(CR4_PSE), %eax
1148    movl %eax, %cr4

xv6/entry.S entry(): kernel ELF entry
32bit x86 supports two page sizes

- 4KB pages
- 4MB pages
Page translation for 4MB pages

0x410010 = 00 0000 0001|00 0001 0000|0000 0001 0000

Page Table
Directory
(aka Level 1)

Physical memory

CR3 = 0x8192
Page translation for 4MB pages

Linear Address

31  22  21  0
  |    |    |   |
  Directory  Offset

/  /  /  /
10  22  18  32

Page Directory

PDE with PS=1

4-MByte Page

Physical Address

CR3
Set up page directory

1149 # Set page directory
1150 movl $(V2P_W0(entrypgdir)), %eax
1151 movl %eax, %cr3

xv6/entry.S
Our goal: 2GB/2GB address space
First page table

- Two 4MB entries (large pages)
- Entry #0
  - 0x0 – 4MB $\rightarrow$ 0x0:0x400000
- Entry #512
  - 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,
};
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
// 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
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
1406 // The boot page table used in entry.S and entryother.S.
1407 // Page directories (and page tables) must start on page boundaries,
1408 // hence the __aligned__ attribute.
1409 // PTE_PS in a page directory entry enables 4Mbyte pages.
1410
1411 __attribute__((__aligned__(PGSIZE)))
1412 pde_t entrypgdir[NPD ENTRIES] = {
1413   // Map VA’s [0, 4MB) to PA’s [0, 4MB)
1414   [0] = (0) | PTE_P | PTE_W | PTE_PS,
1415   // Map VA’s [KERNBASE, KERNBASE+4MB) to PA’s [0, 4MB)
1416   [KERNBASE>>PDXSHIFT] = (0) | PTE_P | PTE_W | PTE_PS,
1417 };
First page table (cont)

0870 // Page directory and page table constants.
0871 #define NPDENTRIES 1024
Turn on paging

1152 # Turn on paging.
1153 movl %cr0, %eax
1154 orl $(CRO_PG|CRO_WP), %eax
1155 movl %eax, %cr0
1157 # Set up the stack pointer.
1158      movl $(stack + KSTACKSIZE), %esp
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

Protected Mode

GDT

CS : 0x8  EIP: 0x10001a
SS : 0x10  ESP: stack
GDT: 0x7c78  TSS: 0x0
IDT: 0x0  CR3: entrypgdir

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

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

0x80000000
0x80400000

0 0x7c00 0x7f00 0x100000
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\n", cpunum());
1325     ...
1340 }
Recap of the boot sequence

- Setup segments (data and code)
- Switched to protected mode
  - Loaded GDT (segmentation is on)
- Setup stack (to call C functions)
- Loaded kernel from disk
- Setup first page table
  - 2 entries \([0 : 4MB]\) and \([2GB : (2GB + 4MB)]\)
- Setup high-address stack
- Jumped to main()
Conclusion

• We've booted
  • We're running in main()

• Next time:
  • OS interfaces
Thank you!
References

```c
#include <stdio.h>

void func_a(void)
{
    printf("func_a\n");
    return;
}

void func_b(void)
{
    printf("func_b\n");
    return;
}

int main(int ac, char **av)
{
    void (*fp)(void);
    fp = func_b;
    fp();
    return;
}
```
Function pointers

08048432 <func_b>:

8048432:      55                      push   %ebp
8048433:      89 e5                   mov    %esp,%ebp
8048435:      83 ec 18                sub    $0x18,%esp
8048438:      c7 04 24 07 85 04 08    movl   $0x8048507,(%esp)
804843f:      e8 ac fe ff ff          call   80482f0 <puts@plt>
8048444:      90                      nop
8048445:      c9                      leave
8048446:      c3                      ret

08048447 <main>:

8048447:      55                      push   %ebp
8048448:      89 e5                   mov    %esp,%ebp
804844a:      83 e4 f0                and    $0xfffffff0,%esp
804844d:      83 ec 10                sub    $0x10,%esp
8048450:      c7 44 24 0c 32 84 04    movl   $0x8048432,0xc(%esp)
8048457:      08                      
8048458:      8b 44 24 0c             mov    0xc(%esp),%eax
804845c:      ff d0                   call   *%eax
804845e:      90                      nop
804845f:      c9                      leave
8048460:      c3                      ret
08048432 <func_b>:
  8048432:      55                      push   %ebp
  8048433:      89 e5                   mov    %esp,%ebp
  8048435:      83 ec 18                sub    $0x18,%esp
  8048438:      c7 04 24 07 85 04 08    movl   $0x8048507,(%esp)
  804843f:      e8 ac fe ff ff          call   80482f0 <puts@plt>
  8048444:      90                      nop
  8048445:      c9                      leave
  8048446:      c3                      ret

08048447 <main>:
  8048447:      55                      push   %ebp
  8048448:      89 e5                   mov    %esp,%ebp
  804844a:      83 e4 f0                and    $0xfffffff0,%esp
  804844d:      83 ec 10                sub    $0x10,%esp
      # Load pointer to func_p on the stack
  8048450:      c7 44 24 0c 32 84 04    movl   $0x8048432,0xc(%esp)
  8048457:      08                      
  8048458:      8b 44 24 0c             mov    0xc(%esp),%eax
  804845c:      ff d0                   call   *%eax  # Call %eax
  804845e:      90                      nop
  804845f:      c9                      leave
  8048460:      c3                      ret

Function pointers