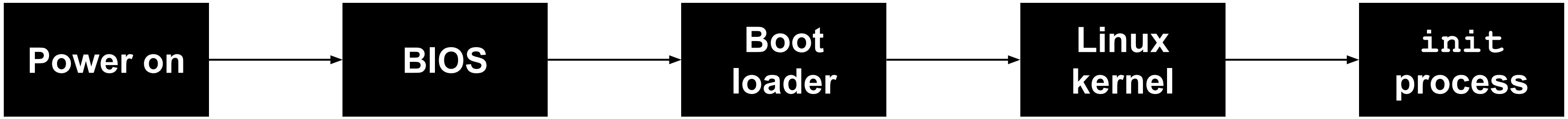
The Linux boot process

Karl Trygve Kalleberg

Hackeriet 21.01.16

Overview



Overview

Power on → **BIOS** → **Boot loader** → **Linux kernel** → **`init` process**

I take an x86-centric view today

Power on

- RAM is disabled
  - don’t even know how much RAM is available yet
- Video is disabled
  - don’t even know if we have video at all
- All buses are disabled
  - don’t even know if we have USB, SATA, serial, parallel, Bluetooth, PATA, Trønderbolt, whatever
- Challenge
  - How do we execute anything if we don’t have any memory?
  - How does the CPU know what to do at all?

CPU initialization

- Cold boot: CPU starts with all registers “hardcoded”
  - CPU always does the exact same thing on every reset
  - CPU starts execution at the same address location every time.
    - This location is called the *reset vector*
- BIOS: mapped into the memory space as read-only
  - Remember, we don’t have RAM yet

Traditional x86 memory map

`0x00000000 – 0x000003FF` - Real Mode Interrupt Vector Table
`0x00000400 – 0x000004FF` - BIOS Data Area
`0x00000500 – 0x00007BFF` - Unused
`0x00007C00 – 0x00007DFF` - Our Bootloader
`0x00007E00 – 0x0009FFFF` - Unused
`0x000A0000 – 0x000BFFFF` - Video RAM (VRAM) Memory
`0x000B0000 – 0x000B7777` - Monochrome Video Memory
`0x000B8000 – 0x000BFFFF` - Color Video Memory
`0x000C0000 – 0x000C7FFF` - Video ROM BIOS
`0x000C8000 – 0x000EFFFF` - BIOS Shadow Area
`0x000F0000 – 0x000FFFFF` - System BIOS

BIOS initialization

- On 16-bit x86:
  - CPU wakes up with CS = `0xFFFF`, IP = `0x0000`
    - => first CPU instruction is at address (1MB - 16 bytes)
    - this instruction is always a `JMP` instruction
    - the `JMP` instruction leads the CPU into the first BIOS instruction
- Modern x86 (32 / 64 bit):
  - CPU wakes up with CS = `0xF000` (selector), EIP = `0xFFF0`
    - selector `0xF000` has *base* `0xFFFF0000` and *limit* `0xFFFF`
      - I.e. it points to final 64kb memory region, just before 4GB
    - => first CPU instruction is at address (1MB - 16 bytes)
    - rest is same as above (`JMP` → BIOS)

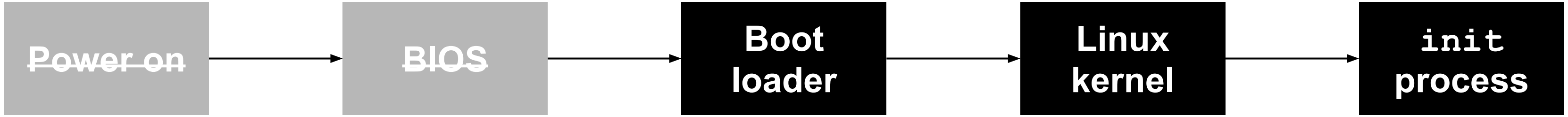
BIOS initialization example

- coreboot 4.2:
  - `src/cpu/x86/16bit/reset16.inc`
  - `src/cpu/x86/16bit/entry16.inc`

BIOS bootstrap sequence

- Power-on-self-test (POST)
  - Check, identify and initialize
    - CPU
    - RAM
    - interrupt and DMA controllers
    - rest of chipset
    - video display card (own BIOS)
    - buses (USB, SATA/PATA)
    - detect keyboard
    - detect bootable devices on the various buses
- Search bootable devices, in some order, for first available bootloader

Overview



BIOS hands over to bootloader

- Switch CPU into 16-bit mode
  - Because backwards compatibility (e.g. all primary bootloaders are 16-bit)
- For bootloader on hard drive
  - Load master boot record (MBR)
    - This always on the 0th sector of the drive
    - First 440 bytes are primary boot loader (= stage 0 bootloader)
    - Next 64 bytes are the partition table
    - Final two bytes are MBR magic (`0xAA`, `0x55`)
  - After MBR is loaded into memory
    - JMP to first instruction of primary boot loader
- For bootloaders on other mediums
  - Out of scope today, but very similar

BIOS hands over to bootloader

- Switch CPU into 16-bit mode
  - Because backwards compatibility (e.g. all primary bootloaders are 16-bit)
- For bootloader on hard drive
  - Load master boot record (MBR)
    - This always on the 0th sector of the drive
    - First 440 bytes are primary boot loader (= stage 0 bootloader)
    - Next 64 bytes are the partition table
    - Final two bytes are MBR magic (`0xAA`, `0x55`)
  - After MBR is loaded into memory
    - JMP to first instruction of primary boot loader
- For bootloaders on other mediums
  - Out of scope today, but very similar

Intermezzo: BIOS vs UEFI

- BIOS design is ~~mature~~ old
  - Older than *Knight Rider* (yes, with Hasselhoff!)
  - 16-bit was enough for everyone
- Practically impossible to change without breaking the world
- Solution:
  - Break the world → UEFI
- But:
  - UEFI is rather complicated
  - Better to show you a simple boot process with the BIOS instead

Stage 1 bootloader constraints

- Problem: Primary bootloader is < 440 bytes
  - Not enough room for all tasks the bootloader needs to perform
- Solution: Bootloader loads another bootloader
  - First partiton (usually) starts at sector 63 (or, later)
    - => free space in sector 1 - 62
    - plenty of room for another bootloader, (GRUB: stage 1.5 bootloader)
- Primary bootloader
  - loads the secondary bootloader into memory
  - JMPs to first instruction in secondary bootloader

Stage 1.5 bootloader

- Contains code for
  - Reading the `/boot` filesystem
    - E.g. relevant file system driver that matches the filesystem on `/boot`
  - Necessary I/O drivers for reading from the type of drive at hand
    - Harddrive, CDROM, etc
- Loads the tertiary boot loader (stage 2 bootloader)
  - Contained in `/boot/grub` (e.g. `/boot/grub/i386-pc`)

Stage 2 bootloader

- Contains code for
  - Video / text mode drivers
  - Menu handling
  - Editor / “debugger”
  - A small REPL
  - Loading + extracting Linux kernel + initrd image
  - File system drivers
  - Chain loader (for loading yet another boot loader, e.g. Windows’)

Stage 2 bootloader

- User selects desired Linux kernel, then:
  - Bootloader loads kernel
    - usually from `/boot/vmlinuz-`*`something`*
  - Bootloader loads initrd image
    - usually from `/boot/initrd.img-`*`something`*

Overview

~~Power on~~ → ~~BIOS~~ → ~~Boot loader~~ → Linux kernel → `init` process

Linux binary + initrd image

- Examples
  - `/boot/vmlinuz-4.2.0-25-generic`
    - bzip2-compressed
    - MZ header??
  - `/boot/initrd.img-4.2.0-25-generic`
    - gzipped
    - cpio archive

Linux binary + initrd image

- Examples
  - `/boot/vmlinuz-4.2.0-25-generic`
    - bzip2-compressed
    - MZ header??
  - `/boot/initrd.img-4.2.0-25-generic`
    - gzipped
    - cpio archive

Load Linux kernel into memory

- Linux has a boot protocol
    - https://github.com/torvalds/linux/blob/master/Documentation/x86/boot.txt
- Specifies
    - where in memory the various parts of the kernel may be loaded
    - how to supply boot parameters to the kernel
    - where to place the `initrd` image (if any)
    - how to avoid overwriting other system software (e.g. BIOS)
- Various revisions exist
    - Tracks evolution of both Linux kernel features + hardware features
        - e.g. version 2.12 (kernel 3.8) adds support for loading > 4GB on 64bit machines

Memory layout for Linux kernel loading

```
       |  Protected-mode kernel  |
100000 +------------------------+
       |  I/O memory hole        |
0A0000 +------------------------+
       |  Reserved for BIOS      |  Leave as much as possible unused
       ~                         ~
       |  Command line           |  (Can also be below the X+10000 mark)
X+10000+------------------------+
       |  Stack/heap             |  For use by the kernel real-mode code.
X+08000+------------------------+
       |  Kernel setup           |  The kernel real-mode code.
       |  Kernel boot sector     |  The kernel legacy boot sector.
X       +------------------------+
       |  Boot loader            |  <- Boot sector entry point 0000:7C00
001000 +------------------------+
       |  Reserved for MBR/BIOS  |
000800 +------------------------+
       |  Typically used by MBR  |
000600 +------------------------+
       |  BIOS use only          |
000000 +------------------------+
```

Kernel parameters (except cmdline)

- These are found at X + `0x01F1`
- X was shown on the previous slide
- X is the base address where the real mode part of the Linux kernel is loaded

```
01F1/1  ALL    setup_sects         The size of the setup in sectors
01F2/2  ALL    root_flags          If set, the root is mounted readonly
01F4/4  2.04+  syssize             The size of the 32-bit code in 16-byte paras
01F8/2  ALL    ram_size            DO NOT USE - for bootsect.S use only
01FA/2  ALL    vid_mode            Video mode control
01FC/2  ALL    root_dev            Default root device number
01FE/2  ALL    boot_flag           0xAA55 magic number
0200/2  2.00+  jump                Jump instruction
0202/4  2.00+  header              Magic signature "HdrS"
0206/2  2.00+  version             Boot protocol version supported
0208/4  2.00+  realmode_swtch      Boot loader hook (see below)
020C/2  2.00+  start_sys_seg       The load-low segment (0x1000) (obsolete)
020E/2  2.00+  kernel_version      Pointer to kernel version string
0210/1  2.00+  type_of_loader      Boot loader identifier
0211/1  2.00+  loadflags           Boot protocol option flags
0212/2  2.00+  setup_move_size     Move to high memory size (used with hooks)
0214/4  2.00+  code32_start        Boot loader hook (see below)
0218/4  2.00+  ramdisk_image       initrd load address (set by boot loader)
021C/4  2.00+  ramdisk_size        initrd size (set by boot loader)
0220/4  2.00+  bootsect_kludge     DO NOT USE - for bootsect.S use only
0224/2  2.01+  heap_end_ptr        Free memory after setup end
0226/1  2.02+  ext_loader_ver      Extended boot loader version
...
```

Handover from bootloader to Linux kernel

- Boot loader loads
  - Linux kernel in two parts
    - Real-mode parts < 1 MB (= below `0x100000`)
    - Protected mode parts > 1 MB (= above `0x100000`, potentially > 4GB)
  - Initrd
    - At free location > 1 MB
- Boot loader initializes kernel parameters
  - cmdline string (at X + `0x10000`)
  - real-mode header (at X + `0x01F1`)
  - switches to 16-bit
  - jumps to first kernel instruction

First kernel instructions

- `arch/x86/boot/header.S`
  - Sets up registers, including stack
  - Check for signature (simple anti-corruption check)
  - Zero out bss (static variables section)
  - jump to `main()`
- Rest of kernel startup story is mostly written in C

The Linux kernel main() function

```c
void main(void)
{
      copy_boot_params();
      console_init();
      if (cmdline_find_option_bool("debug")) puts("early console in setup code\n");
      init_heap();
      if (validate_cpu()) { puts("Unable to boot - please use a kernel appropriate"
         "for your CPU.\n"); die(); }
      set_bios_mode();
      detect_memory();
      keyboard_init();
      query_ist();
      set_video();
      go_to_protected_mode();
}
```

The Linux kernel main() function

```
void main(void)
{
      copy_boot_params();
      console_init();
      if (cmdline_find_option_bool("debug")) puts("early console in setup code\n");
      init_heap();
      if (validate_cpu()) { puts("Unable to boot - please use a kernel appropriate"
         "for your CPU.\n"); die(); }
      set_bios_mode();
      detect_memory();
      keyboard_init();
      query_ist();
      set_video();
      go_to_protected_mode();
}
```

All of this still happens in 16-bit mode

In protected mode

- Protected mode = 32/64 bit mode
  - We have a plenty of address space
- Decompress kernel
  - Jump to decompressed kernel

In protected mode

- Decompress initrd
- `init/main.c: kernel_init()`
    - `kernel_init_freeable()`

```
if (!ramdisk_execute_command)
    ramdisk_execute_command = "/init";
```

    - back in `kernel_init()`:

```
if (ramdisk_execute_command) {
    ret = run_init_process(ramdisk_execute_command);
    if (!ret) return 0;
    pr_err("Failed to execute %s (error %d)\n",
            ramdisk_execute_command, ret);
}
```

Overview

~~Power on~~ → ~~BIOS~~ → ~~Boot loader~~ → ~~Linux kernel~~ → `init` process

The `/init` on `initrd`

- On Ubuntu
  - creates a minimal root file system
  - mounts `/proc`, `/sys`, `/dev`, `/dev/pts`
  - parses `/proc/cmdline`
    - (provided by the bootloader a few slides ago)
  - loads “essential” modules
  - mounts root (e.g. /`dev/sdaX`) file system on `/root`
    - (yes, that *is* confusing!)
    - checks that this filesystem has a valid init
  - moves earlier mounts into to upcoming root
  - does a `switch_root` or `run-init`
    - => the “real” `init` is finally run

The `/init` on `initrd`

- On Ubuntu
  - creates a minimal root file system
  - mounts `/proc`, `/sys`, `/dev`, `/dev/pts`
  - parses `/proc/cmdline`
    - (provided by the bootloader a few slides ago)
  - loads “essential” modules
  - mounts root (e.g. /`dev/sdaX`) file system on `/root`
    - (yes, that *is* confusing!)
    - checks that this filesystem has a valid init
  - moves earlier mounts into to upcoming root
  - does a `switch_root` or `run-init`
    - => the “real” `init` is finally run

Overview

~~Power on~~ → ~~BIOS~~ → ~~Boot loader~~ → ~~Linux kernel~~ → `init` process

The `/sbin/init` on root

- After initrd has finished
  - most file systems are mounted
  - the console is somewhat prepared
  - the most essential drivers are loaded
    - storage, network, nfs
- Handover is done to `/sbin/init`
  - This may be Upstart, System V init,
  - … even `systemd`
    - the subject of our next talk!

Resources

- http://www.ibm.com/developerworks/library/l-linuxboot/
- https://www.coreboot.org/releases/
- https://www.gitbook.com/book/0xax/linux-insides/details
  - Recommended!
- https://github.com/torvalds/linux/blob/master/Documentation/x86/boot.txt