BOOTSTRAP, IA32, AND BIOS/UEFI

CS124 – Operating Systems

Spring 2024, Lecture 5

Bootstrapping

- All computers have the same basic issue:
  - They require a program to tell them what to do…
  - …but when you turn them on, they have no program!
  - How do you get them to start running programs?
- Computers must implement a **bootstrap process** to load the OS
  - A series of one or more boot-loaders
  - Each stage is responsible for loading the next stage of the computer's programming
- Originated from the term "pulling oneself up by one's bootstraps"
  - Was used in the context of computing in the early 1950s
  - (The phrase was extant well before that time)

Bootstrapping (2)

- Modern computers use read-only memory (ROM) containing the initial code to load the operating system
- Pre-ROM computers had to implement various techniques
  - Read a small number of instructions from some external source, then begin executing them!
- Examples:
  - Computers with punched-card readers were designed to read one card, then begin executing that card's instructions
  - Some computers included a bank of switches to specify the first instruction(s) for the computer to execute
  - Other computers used diode matrixes; presence of a diode = 1, absence of a diode = 0
- Result: boot-loaders were very primitive

Bootstrapping (3)

- The advent of read-only memory (ROM) revolutionized boot-loading
  - Manufacturers included sophisticated programs on the computer motherboard to manage the operating system boot process
  - Software even includes basic drivers for disks, video, minimal operating systems, etc.
- The combination of persistent memory and the program stored in it is called **firmware**
- Systems now use electrically-erasable versions of ROM (EPROM, EEPROM), allowing for firmware upgrades

IA32 Bootstrap

- IA32 processors are engineered to start executing instructions at 0xFFFFFFF0 immediately after a reset
- Computer manufacturers place a read-only memory (ROM) in this address range to start the boot process
  - ROM is typically much larger than 16 bytes
  - Modern computers include very sophisticated firmware now
- IA32 CPUs also start off in ring 0 (kernel mode)
  - Gives the bootloader full access to all system facilities, in order to set up the hardware to run the operating system
- Currently there are two categories of PC boot-loaders:
  - PC BIOS – Basic Input/Output System
  - EFI/UEFI – [Unified] Extensible Firmware Interface

PC BIOS

- Original firmware for x86 computers
- Provides two critical features, and a third useful one:
  - A firmware bootloader to start the bootstrap process
  - A library of basic input/output functions for interacting with the computer hardware
    - i.e. device drivers exposed via a standard interface
  - Often includes a simple user interface for hardware configuration
- BIOS functionality emerged as a de-facto standard
  - Certain microcomputers (IBM PC) and operating systems (MS DOS) became very popular
  - Other manufacturers began to clone the hardware…
  - They had to match existing firmware functions for software to work!

PC BIOS Bootloading

- BIOS bootloader follows a very simple process:
- If a hardware reset was performed, run some diagnostics on the hardware
  - e.g. memory check
  - Called a **Power-On Self Test**, a.k.a. POST
- Identify and configure computer peripherals for basic use
- Iterate through bootable devices in some order, trying to load and start the next stage of the bootstrap process
  - The first sector of each bootable device is loaded into memory
  - If sector ends with signature 0x55, 0xAA, it is used as bootloader

PC BIOS Bootloading (2)

- BIOS loads the boot sector at memory address 0x7C00, then jumps to that address
- The boot sector that BIOS loads is only 512 bytes (!!!)
  - Historically, this was the size of x86 disk sectors
  - 0x200 bytes, so bootloader is at addresses 0x7C00 – 0x7DFF
  - Minus the 55AA signature, boot sector has 510 bytes to do its thing
- BIOS passes a few limited details to the bootloader
  - e.g. `%dl` register contains the numeric ID of the boot disk; allows the bootloader to retrieve more data from the boot disk
- Bootloaders are usually written in assembly language
  - Only way to cram the required functionality into the limited space

Bootloading and MBRs

- Picture grows more complex from disk **partitioning**
- First sector of a hard disk is a **master boot record** (MBR)
  - Specifies up to four **partitions** of the hard disk, each with its own format and use
  - e.g. each partition could be used for a different OS
  - An OS might also need multiple partitions, e.g. Linux filesystem partition vs. Linux swap partition
- Issue: MBR doesn't correspond to a specific operating system; the disk may contain multiple OSes
- A partition with an OS often has its own bootloader in the first sector of that partition
- MBR must kick off the next bootloader in the process

Bootloading and MBRs (2)

- Partition details in the master boot record also take up some space…
  - 4 partition-table entries × 16 bytes per entry = 64 bytes
- Older style MBR format:
  - 512 bytes – 2 byte signature – 64 byte partition table = 446 bytes for bootloading
- Newer MBRs include more details, reducing the MBR loader size to 434-436 bytes, broken into two parts
- Advanced bootloaders like LILO and GRUB clearly can't fit in this small space
  - The bootloader itself is broken into multiple stages
  - Stage 1 is 512 bytes, responsible for loading stage 2 into memory
  - Stage 2 is much larger, e.g. 100s of KBs

Bootloading and MBRs (3)

- MBR loaders use a mechanism called **chain loading**
  - Emulate the BIOS mechanism, so partition loader doesn't need to know it wasn't loaded by the BIOS
- MBR loads the next bootloader into address 0x7C00, then jumps to that address and begins running it
  - Also retains other register values from BIOS, e.g. `%dl` contains same value that BIOS passed to the MBR loader
- Of course, the MBR loader was already at 0x7C00…
  - MBR loader copies itself to another location, then jumps to the copy of itself, before it loads the partition boot sector
  - Chain loaders often copy themselves to address 0x0600

BIOS Library Functions

- At this early stage, bootloaders rely on BIOS functions to interact with the computer hardware
- Bootloaders have same issue that user applications have: They don't know the addresses of BIOS operations
- All BIOS functions are invoked via software interrupts
- Example: **`int $0x10`** is used for video services
  - Specify operation to perform in **`%ah`**
  - Other parameters are stored in other registers
  - Invoke **`int $0x10`** to perform the operation
- Example: Print a '*' character in teletype mode:

```
movb    $0x0e, %ah    # Use BIOS teletype function
movb    '*', %al
int     $0x10         # Invoke BIOS video service
```

IA32 Memory Addressing

- A complication of OS loading is that BIOS operations require the use of x86 *real-addressing mode*
  - An ancient memory-addressing mode used by original 8086/8088
- IA32 architecture has several memory-addressing modes
  - Some provide advanced ways for modern operating systems to manage memory
  - Others are required for backward compatibility
- The ultimate goal of the OS is to get into *protected-mode* memory addressing
  - Supports memory protections, virtual memory management, etc.
  - *At that point, BIOS calls will no longer work!*

Protected Mode and BIOS

- Once the system is in protected mode, we can no longer use BIOS functions to interact with the hardware
  - Problem: BIOS uses (and requires) IA32 real-addressing mode
  - Problem: BIOS uses 16-bit Interrupt Vector Table entries

- From this point forward, the operating system must use its own **device drivers** to interact with computer hardware
  - Software components that know how to interact with a specific kind of device, but that also present a simple, generic interface
- OS device drivers often reinitialize the hardware to suit the needs/preferences of the operating system

IA32 BIOS Bootstrap Process – Summary

- At power on, IA32 processor starts executing instructions at the address 0xFFFFFFF0
  - A ROM memory is positioned at this hardware address to jump to BIOS bootstrap code
- BIOS bootstrap code performs a power-on self test if it was a hardware reset (skips if software reset)
- Next, BIOS attempts to load the first 512-byte sector of each bootable device in the system at address 0x7C00
  - On success, BIOS jumps to address 0x7C00 for next stage of boot
- If it's an MBR bootloader, it must chain-load the boot sector for the operating system to start
- Finally, the OS bootloader must load the OS kernel into memory and jump to the kernel bootstrap code

Plug and Play

- Another major issue from BIOS bootstrap: BIOS only exposes very basic device functionality and access
  - “Lowest Common Denominator” type functionality
- To identify other devices in the computer, the OS kernel had to probe various IO ports and see what responded
  - e.g. sound cards, graphics cards, networking cards, etc.
- Sometimes multiple devices would use the same ports…
- Sometimes probing for one kind of device could cause another kind of device to hang the system…
- Over time, several standards were published to make PC systems more “**plug-and-play**” capable
  - Allow the OS to identify and configure hardware devices safely and automatically, via standard mechanisms

Plug and Play (2)

- To support a “plug-and-play” auto-discovery mechanism, the hardware bus protocol must be updated:
  - Devices must include a vendor-specified device ID and type value
  - Must specify a standard mechanism for querying this information off of all bus devices
  - When system buses are initialized, system can enumerate devices connected to the bus and handle each device’s basic initialization
- Example plug-and-play buses:
  - PCI family of buses (PCI, PCI Express, Mini PCI, etc.)
  - USB, FireWire
  - PC Card/PCMCIA (for removable laptop peripherals)

Plug and Play (3)

- With hardware that facilitates device discovery, systems began providing more detailed information to the OS
- Frequently exposed as tables of data set up by the BIOS during bootstrap

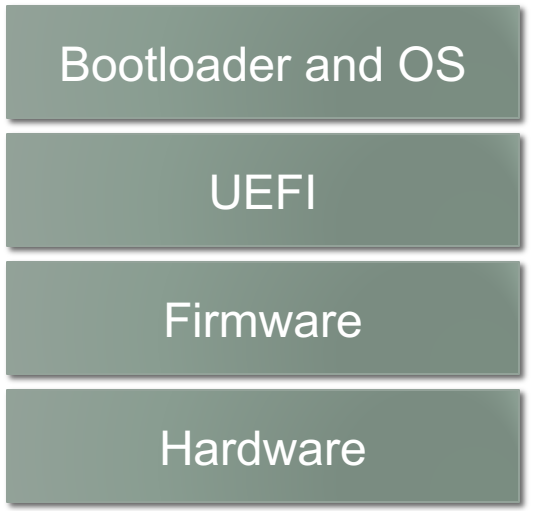
- Example: Intel MultiProcessor Specification (1997)
  - Identifies processor manufacturer, model number, etc.
  - Identifies all system buses, processors, processor APIC IDs, etc.
  - Table is set up by the BIOS at startup time
  - A multiprocessor operating system can locate this table and use it to run processes on all available processors

ACPI Standard

- One of the more notable standards is the ACPI standard
  - Advanced Configuration and Power Interface
  - Defines a platform-independent interface for hardware discovery, configuration, power management and monitoring
  - Replaces several previous standards
- ACPI primarily consists of a large number of tables that contain platform configuration details
- All tables are accessible through a structure called the Root System Description Pointer
- Tables include details for all major aspects of the system
- Tables are initialized by bootstrap firmware
  - e.g. multicore/multiprocessor and APIC details
  - e.g. memory characteristics and memory topology

Unified Extensible Firmware Interface

- Data tables are helpful…
- BIOS is still very limiting for modern OSes to deal with
  - Can't even use it after switching to protected mode
- A new standard has emerged: Unified Extensible Firmware Interface (UEFI)
  - Completely replaces the old BIOS interface with a new, modular, extensible firmware
- Prompted by Intel Itanium processor
  - 64-bit processor, couldn't run x86 BIOS!
  - Still needed to support an operating system
- UEFI is a firmware interface standard
  - Sits on top of lower-level firmware, not directly on computer hardware



Unified Extensible Firmware Interface (2)

- UEFI is a modular system, allowing components to be installed and removed
- Can install UEFI bootloaders for OSes on the computer
  - Knows how to use UEFI services to load and run the OS
- Can install UEFI applications that allow system hardware, boot configuration, etc. to be managed
  - Runs in the “preboot environment” (before the OS is started)
  - e.g. UEFI systems usually have a command shell for basic tasks
  - UEFI bootloaders (a.k.a. OS loaders) are one kind of application
- UEFI drivers provide standardized abstractions for hardware, including buses and devices
  - Used by UEFI applications and OS loaders to perform their tasks

Unified Extensible Firmware Interface (3)

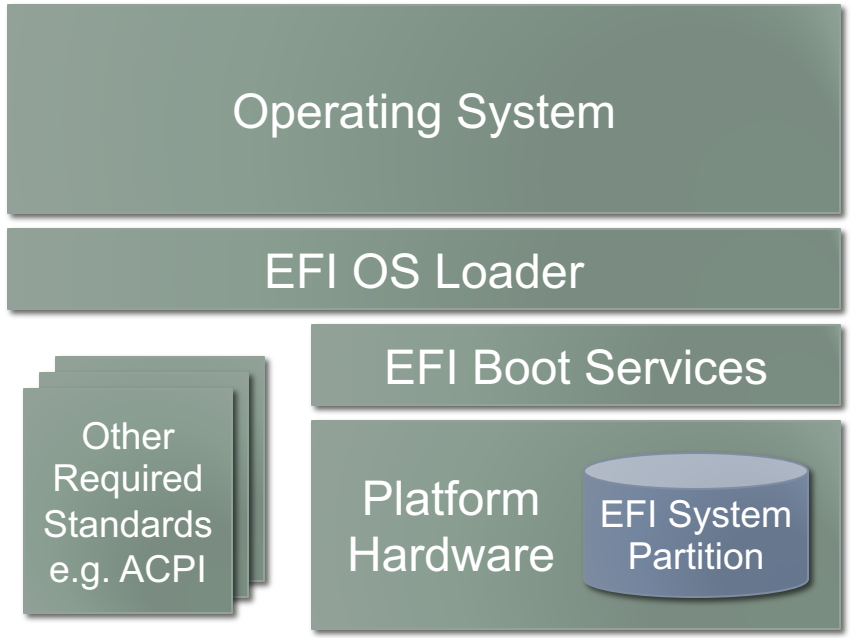
- UEFI also exposes its functionality via tables in memory
- Unlike ACPI and other earlier standards, UEFI includes function-pointers to operations for programs to use
- Example: “Hello World” UEFI application

```
#include <efi.h>
#include <efilib.h>

EFI_STATUS efi_main(EFI_HANDLE ImageHandle,
                    EFI_SYSTEM_TABLE *SystemTable {
    SIMPLE_TEXT_OUTPUT_INTERFACE *console_out;
    InitializeLib(ImageHandle, SystemTable);
    console_out = SystemTable->ConOut;
    uefi_call_wrapper(console_out->OutputString, 2, console_out,
                      (CHAR16 *) L"Hello World\n\r");
    return EFI_SUCCESS;
}
```

Unified Extensible Firmware Interface (4)

- Another picture of UEFI:



- UEFI provides *some* of its functionality in firmware…
- UEFI modules are often stored on a special disk partition
- The EFI System Partition is the first partition of a disk in the system
  - Often a simple format that can be supported in firmware, e.g. FAT32

UEFI and Disk Partitioning

- UEFI addresses another issue: disk partitioning
- Problem: Master Boot Records specify partitions using 32-bit values
  - (32-bit start, 32-bit size, 512-byte sectors)
  - Limits partitions to 2TiB in size!
- Solution: GUID Partition Tables (GPT)
  - GUID = Globally Unique ID, a 128-bit identifier with a high likelihood of being unique
- Partition descriptors use 64-bit values – 9.4 zettabyte (8×10^{21}, or 8 zebibytes = $8\times512\times2^{64}$ byte) partitions!
- Disks, partition types and partitions all identified by GUIDs
  - Partition-type GUIDs are standardized
  - (MBR uses a 1-byte value to indicate partition type)

GUID Partition Tables

- Also allows up to 128 partitions per hard disk
  - GUID partition table occupies 33 sectors at the start of the disk
- A “legacy MBR” occupies first sector (LBA 0)
  - Includes a single partition that covers the entire disk
  - Partition type is set to a value unused by all major OSes
  - Reason: if a legacy MBR tool is used on the disk, it won’t be as likely to mangle the GUID partition table on the disk
- Disks with GUID Partition Tables maintain two GPTs
  - Identical copies, kept at the start and end of the disk
  - Reduces likelihood that corruption will render the disk unusable
- Most modern OSes can use GUID partition tables now
  - Not all of them can boot off of a GPT disk without firmware support

GUID Partition Tables and UEFI

- UEFI specification includes the GUID partition table spec
  - UEFI requires GUID partition tables
- When an OS is installed on a UEFI system, the OS loader is installed into EFI System Partition (using EFI services)
  - Allows the UEFI preboot system to provide multiboot services
- UEFI is large and complex
  - There are definitely bugs in UEFI implementations
  - Not every company follows the UEFI standard precisely
  - Bootloaders definitely still have to jump through hoops with UEFI

For More Information…

- UEFI Standards – https://uefi.org
- TianoCore – https://www.tianocore.org
  - Open-source implementation of UEFI standard
  - Derived from Intel's implementation of EFI for several platforms
  - Includes EDK II (EFI Development Kit) for writing UEFI components
- Windows and Linux both support UEFI
  - GRUB and many other bootloaders understand UEFI
- Can install UEFI bootloaders on macOS (if you dare!)
  - rEFIt – http://refit.sourceforge.net (no longer actively maintained)
  - rEFInd – http://www.rodsbooks.com/refind/ (fork of rEFIt)
- VirtualBox and QEMU can both emulate UEFI hardware

Next Time

- Start exploring the process abstraction