1044414: Advanced Operating Systems and Virtualization **Spring 2018**

Lecture 1: March 2

Lecturer: Alessandro Pellegrini *Scribe: Anxhelo Xhebraj*

1.1 Course Information

1.1.1 Exam

The exam is composed of two parts:

Slides 0.

- [2/5] **Written Part**: 3 theoretical or practical questions
- [3/5] **Practical Project**: same project for everyone to be done singularily. Specifications will be given in the in the middle of the course

The two parts must be done within one year from each other. In the course we will see various versions of the Linux kernel (2.4, 3.0, 4.0). Any version can be used for the project. The more compatible the project is with the various versions, the better.

1.1.2 Course outline

In this course we will try to understand the internals of an operating system. We will use the Linux kernel as reference since it is open source and permits to get some good hands-on experience to understand basic principles that are applied to all operating systems in general.

Slides 0. - pp. 4

Regarding the architecture during the course we will use the Intel architecture as reference.

We will see how to develop kernel modules, to perform kernel debugging and hot patching.

1.2 Boot Sequence

After hitting the powerup button the Boot Sequence starts. It is composed of 6 levels:

BIOS (Basic Input Output System) code stored in flash ROM inside the motherboard to check what hardware devices are connected to the system etc. It calculates how much RAM is available and performs some consistency checks. It creates a memory map and a map of all devices installed in the system. Finally the BIOS loads the Bootloader.

ACPI Table: describes what peripherals are installed in the system

Since BIOS became very convoluted over years a new specification was developed: **UEFI** (Unified Extensible Firmware Interface). UEFI is a tentative of replacing the BIOS firmware interface to give more programming versatility and other features (even security wise).

Bootloader Stage 1 It searches among the various devices to load the Bootloader Stage 2. This is done because BS1 does not have enough space to load the system (less than 512 Bytes in length).

Bootloader Stage 2 loads from the storage the kernel image and executes it.

Kernel Startup is performed between two levels: hardware interaction through assembly code and internal data structures initialization. Spawns the first process: Init.

Init Its goal is to startup and configure the environment known as Runlevels/Targets: Desktop Environment etc.

Runlevels/Targets subset of services to be executed on startup.

1.3 x86 Initial Booting Sequence

Intel processors work on different voltage levels and in order to work 3 reference voltages (3.3V, 5V, 12V) called **rails** must be supplied by the Power Supply Unit (PSU). In addition, voltage regulators on the motherboard or in other components convert these standard voltages to others as necessary (for example DDR2 and DDR3 dual inline memory modules (DIMM) require 1.8 V and 1.5 V). After completing internal tests and determining that the power is ready for use the PSU triggers the Power Good signal informing the motherboard [ATX(2018), PG(2011)].

Finally, clocks are derived from a small number of input clocks and oscillator sources and the reset signal is triggered which gives control to the BIOS.

1.3.1 Real Mode

Slides 1. One CPU is dynamically chosen to be the bootstrap processor (BSP) that runs all of the BIOS and kernel initialization code. The remaining processors, called application processors (AP) at this point remain halted until later. In this primitive power up state the processor is in **real mode** with memory paging disabled (Paging unit disabled) behaving like the original 1978 Intel 8086. In this mode memory is accessed through *Segmentation-based addressing*. The wrangling of the adresses is managed by the **Segmentation Unit** that transforms **Logical Adresses** into **Linear Adresses** (which coincides with the Physical Address in real mode).

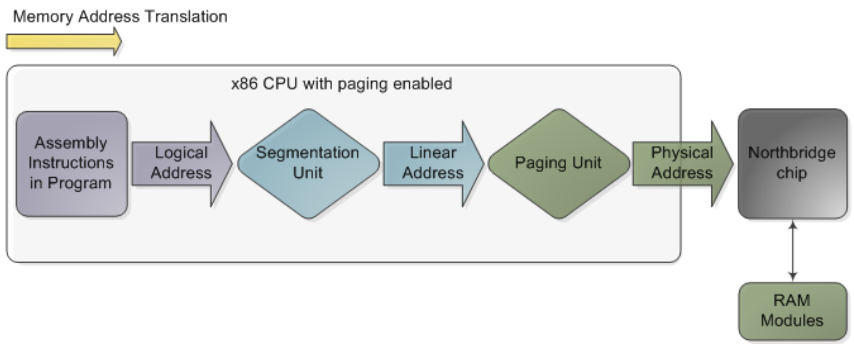


Figure 1.1: Address Translation

Segment Registers

The original 8086 processor had words of 16-bits and this allowed code to work only with 2^{16} bytes (64K). In order to increase addressable memory, segment registers were introduced besides general registers (`AX`, `BX` etc.) to inform the CPU in which of the chunks of memory a program's instructions were going to work on. There were four segment registers: one for the stack (`SS`), one for program code (`CS`) , and two for data (`DS`, `ES`). Nowadays **segmentation** is **still present** and is *always* enabled in x86 processors for backward compatibility. For example, a jump instruction uses the code segment register (`CS`) whereas a stack push instruction uses the stack segment register (`SS`).

Segment registers store 16-bit **segment selectors** which are 16-bit numbers specifying the physical memory address for the start of a segment (this is only in the case of real mode, in protected mode things are different). In this scenario when a physical address needs to be accessed, say for example in one of the data segments, we ask for the word in segment `DS` := `0x1000` with offset `AX` := `0x0012` and the address that is accessed is denoted as `DS:AX`.

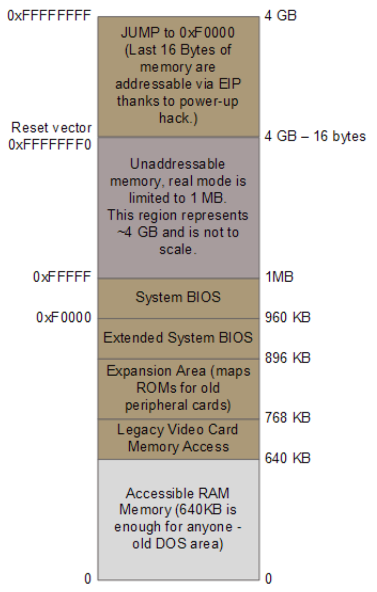
Since physical address pins cost, and at the time 1 MB of memory was thought to be more than sufficient, Intel made the decision to reduce the addressable space to 2^{20} by accessing physical memory through the scheme `DS` $\times 2^4 +$ `AX` instead of using the concatenation of the two registers as address (20 bits, $2^{20} = 1$ MB instead of 32 bits, $2^{32} = 4$GB).

Real mode segment starts range from 0 all the way to `0xFFFF0` (16 bytes short of 1 MB) in 16-byte increments. To these values a 16-bit offset (the logical address) between `0x0` and `0xFFFF` is added. It follows that there are multiple segment/offset combinations pointing to the same memory location, and physical addresses fall above 1MB if your segment is high enough (Memory Wrap-Around).

A20 Line

1.3.2 BIOS operations

The first operation that is fetched is processor dependent and the address at which the operation is found is called **Reset Vector**. In the case of the 8086 processor such address is `F000:FFF0` (`CS:IP`) which corresponds to the physical address `0xF000` $\times 16 +$ `0xFFF0` $=$ `0xFFFF0`, 16 Bytes below the maximum addressable location. The 80386 CPU and later Intel processors have predefined data in some CPU registers after a computer reset: Instruction Pointer (`IP`) set to `0xFFF0`, `CS` to `0xF000` and `Base address` of the "hidden part" of the Code Segment Register set to `0xFFFF0000`. Such addresses are then used to compute the first instruction that is fetched which is `CS Base address + EIP` getting the physical address `0xFFFFFFF0` which is still 16 byte short the maximum addressable memory with 32 bits. The motherboard (Northbridge component) then ensures that the instruction at the Reset Vector is a jump to the memory location mapped to the BIOS entry point. This jump clears the hidden base address present at power up. All of these memory locations have the right contents needed by the CPU thanks to the memory map kept by the chipset. They are all mapped to flash memory containing the BIOS.



0xFFFFFFFF
4 GB
JUMP to 0xF0000 (Last 16 Bytes of memory are addressable via EIP thanks to power-up hack.)
Reset vector 0xFFFFFFF0
4 GB – 16 bytes
Unaddressable memory, real mode is limited to 1 MB. This region represents ~4 GB and is not to scale.
0xFFFFF
1MB
System BIOS
0xF0000
960 KB
Extended System BIOS
896 KB
Expansion Area (maps ROMs for old peripheral cards)
768 KB
Legacy Video Card Memory Access
640 KB
Accessible RAM Memory (640KB is enough for anyone - old DOS area)
0
0

Figure 1.2: Relevant Physical Memory Regions of Later x86 processors.

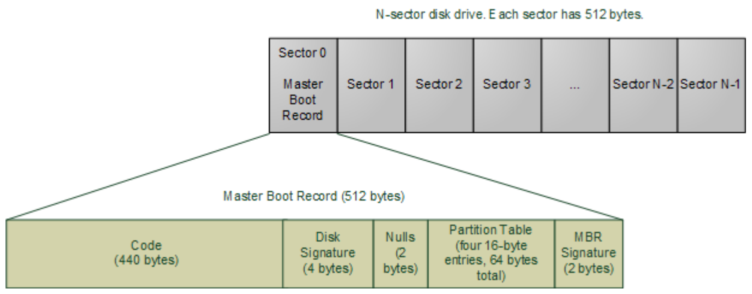
The brown regions are mapped **away** from RAM. When the processor writes/reads such regions the northbridge routes it to the right device.

The CPU then starts executing BIOS code that initializes some of the hardware in the machine and executes **POST** code (Power-On Self Test) that tests and initializes various components in the computer. Lack of a working video card fails the POST and causes the process to halt and beep. A portion of the BIOS is dedicated for communication with legacy video cards.

After the POST the BIOS loads its configuration and then performs **Shadow RAM Initialization**: copies itself on RAM for faster access. The last operation of the BIOS is seeking a boot device from which it loads the first 512-byte sector (sector zero/boot sector) called **Master Boot Record** to the address at `0000:7C00` and performs the jump to that address with `ljmp $0x0000, $0x7C00`

1.3.3 Master Boot Record

The MBR holds the code of the Bootloader Stage 1 that will load the Bootloader Stage 2.


N-sector disk drive. Each sector has 512 bytes.
Sector 0 Master Boot Record
Sector 1
Sector 2
Sector 3
...
Sector N-2
Sector N-1
Master Boot Record (512 bytes)
Code (440 bytes)
Disk Signature (4 bytes)
Nulls (2 bytes)
Partition Table (four 16-byte entries, 64 bytes total)
MBR Signature (2 bytes)

The partition table entries contain offsets telling the beginning of the 4 partitions. At the beginning of each partition there can be one Boot sector. To one partition entry can be designated an *extended partition* which can be subdivided into a number of logical partitions. Each of the logical partitions within the extended partition is preceeded by the Extended Boot Record (EBR) and each EBR has a pointer to the next EBR forming a linked list.

The MBR Signature *must be* `0x55AA`.

The initial bytes of the MBR can contain the **BIOS Parameter Block** (BPB) that is a data structure describing the physical layout of a data storage volume, in order for the BIOS to know how to read it etc. Therefore after the load another `jmp` is performed to skip the BPB.

Finally interrupts are disabled (`cli` instruction) since the stack segment is not initialized yet and all the segment selectors are set to zero to have a linear access on physical addresses.

References

[iap(1985)] *IAPX 286: programmers reference manual.* Intel Corp., 1985.

[ibm(2006)] Inside the linux boot process, May 2006. URL https://www.ibm.com/developerworks/library/l-linuxboot/index.html.

[bio(2007)] Bios startup, 2007. URL https://www.redhat.com/archives/rhl-list/2007-August/msg03384.html.

[win(2009)] Troubleshooting disks and file systems, 2009. URL https://docs.microsoft.com/en-us/previous-versions/windows/it-pro/windows-xp/bb457122(v=technet.10).

[PG(2011)] Power good signal, Dec 2011. URL http://www.tomshardware.co.uk/power-supply-specifications-atx-reference,review-32338-2.html.

[boo(2014)] The hardware boot process: Operating system independent, 2014. URL http://www.tomshardware.com/reviews/pc-repair-upgrade-maintenance-testing,3629-7.html.

[bio(2017)] Bios basics, 2017. URL http://www.bioscentral.com/misc/biosbasics.htm.

[lin(2017)] *Linux Inside.* 0xAX., 2017.

[ATX(2018)] Atx, Feb 2018. URL https://en.wikipedia.org/wiki/ATX.

[wik(2018a)] Boot sector, Feb 2018a. URL https://en.wikipedia.org/wiki/Boot_sector.

[wik(2018b)] Extended boot record, Feb 2018b. URL https://en.wikipedia.org/wiki/Extended_boot_record.

[wik(2018c)] Master boot record, Feb 2018c. URL https://en.wikipedia.org/wiki/Master_boot_record.

[wik(2018d)] Reset vector, Feb 2018d. URL https://en.wikipedia.org/wiki/Reset_vector.

[Duarte(2008a)] Gustavo Duarte. How computers boot up, Jun 2008a. URL https://manybutfinite.com/post/how-computers-boot-up/.

[Duarte(2008b)] Gustavo Duarte. Motherboard chipsets and the memory map, Jun 2008b. URL https://manybutfinite.com/post/motherboard-chipsets-memory-map/.