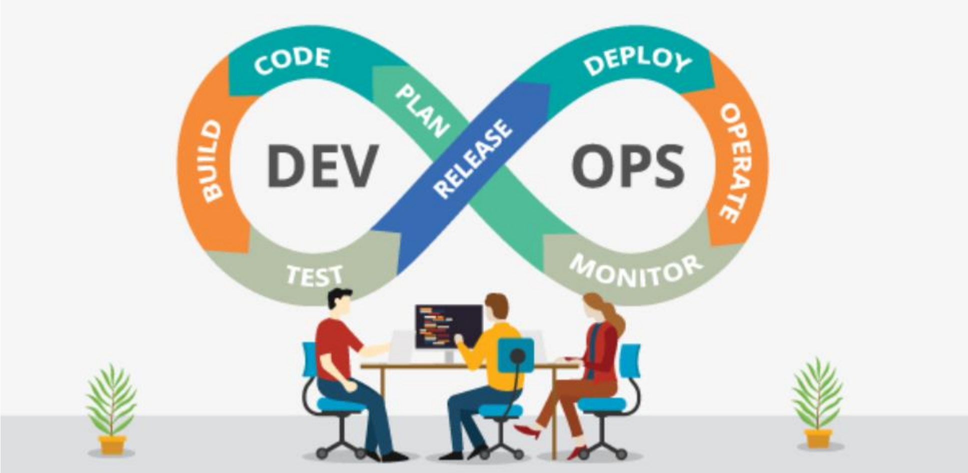
Lecture 3

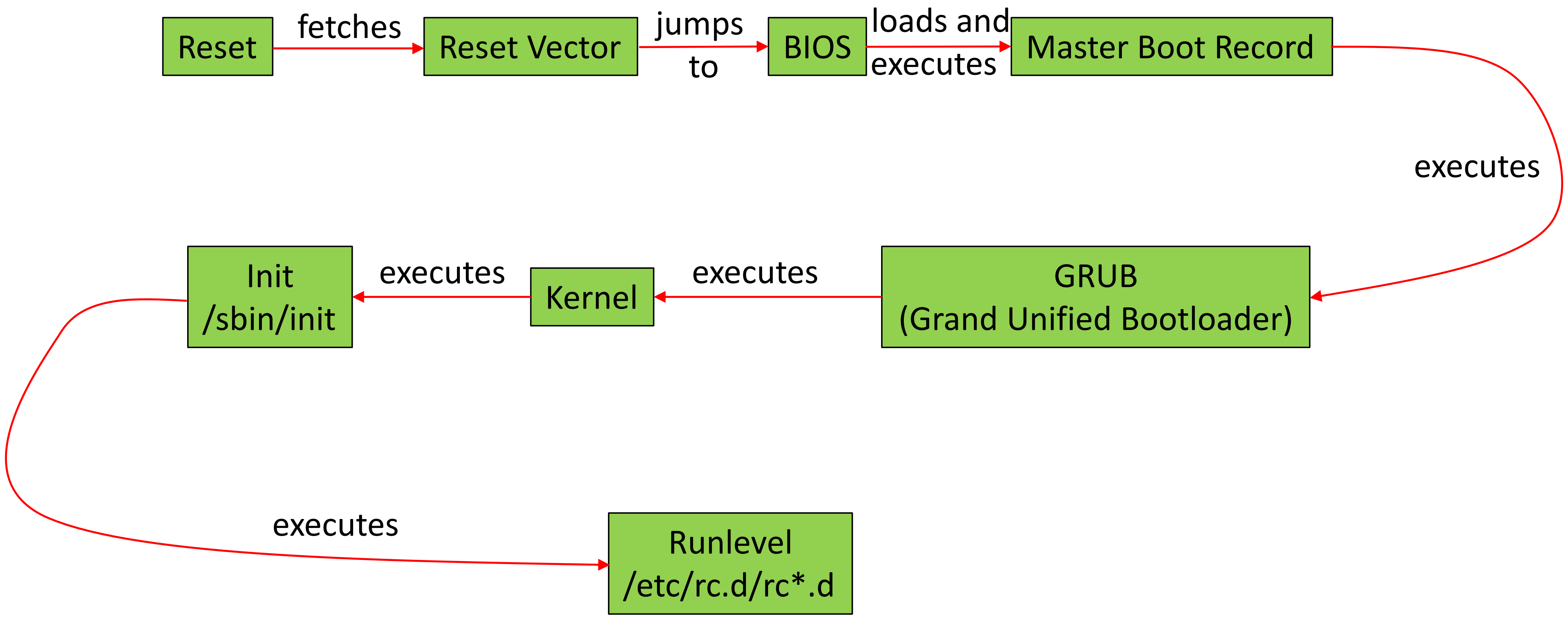
The Boot Process

Learning Objectives

- Understand the boot sequence
- First baby step to being “DevOps”



The Agents in the (Linux) Booting Process



What happens with the x86 processor boot?

- Power sequencing: hardware ensures various power sources are stable at their required voltages
- A variety of hardware subsystems starts self-checks and initialization
- Processor core's PC will be set to the reset vector

Reset vector

- The default value of the PC immediately after reset
  - After critical power and hardware checks
- Hence it is the first instruction fetched
- The chipset (the chips supporting the processor) must map this into the BIOS
- First instruction is a long jump into the real entry of the BIOS code

Reset vector

- 8086 - physical address 0xFFFF0 (16 bytes below 1 MB). CS = 0xFFFF, IP = 0x0000
- 80286 - physical address 0x000FFFF0 (16 bytes below 1 MB). CS = 0xF000, IP = 0xFFF0
- 80386 and later x86 processors – 0xFFFFFFF0 (16 bytes below 4 GB). CS selector =0xF000, CS base = 0xFFFF0000, IP = 0xFFF0h

What about multiprocessor systems?

- During the early reset stage, and at the end of the processors internal self-check, a hardware protocol is used to determine the bootstrap processor (BSP)
  - The protocol differs from processor generation and implementation but basically involve a competition for some kind of semaphore
  - One and only one of the cores will be the bootstrap processor
- Only the BSP executes the boot sequence

BIOS

- “Basic Input/Output System”

- A firmware for hardware initialization and booting that resides in non-volatile memory
  - Updating by “flashing” the EEPROM

- As of 2014, new PCs are shipped with its successor Universal Extensible Firmware Interface (UEFI)

BIOS Implementations

Comparison of different BIOS implementations

	AwardBIOS	AMIBIOS	Insyde	SeaBIOS
License	Proprietary	Proprietary	Proprietary	LGPL v3
Maintained / developed	No	Yes	Yes	Yes
32-bit PCI BIOS calls	Yes	Yes	Yes	Yes
AHCI	Yes	Yes	Yes	Yes
APM	Yes	Yes	Yes (1.2)	Yes (1.2)
BBS	Yes	Yes	Yes	Yes
Boot menu	Yes	Yes	Yes	Yes
Compression	Yes (LHA[30])	Yes (LHA)	Yes (RLE)	Yes (LZMA)
CMOS	Yes	Yes	Yes	Yes
EDD	Yes	Yes	Yes	Yes (3.0)
ESCD	Yes	Yes	?	No
Flash from ROM	?	Yes	?	No
Language	Assembly	Assembly	Assembly	C
LBA	Yes (48)	Yes (48)	Yes	Yes (48)
MultiProcessor Specification	Yes	Yes	Yes	Yes
Option ROM	Yes	Yes	Yes	Yes
Password	Yes	Yes	Yes	No
PMM	?	Yes	?	Yes
Setup screen	Yes	Yes	Yes	No
SMBIOS	Yes	Yes	Yes	Yes (2.4)
Splash screen	Yes (EPA)[31]	Yes (PCX)	Yes	Yes (BMP, JPG)
TPM	Unknown	Unknown	Unknown	Some
USB booting	Yes	Yes	Yes	Yes
USB hub	?	?	?	Yes
USB keyboard	Yes	Yes	Yes	Yes
USB mouse	Yes	Yes	Yes	Yes

Source: Wikipedia

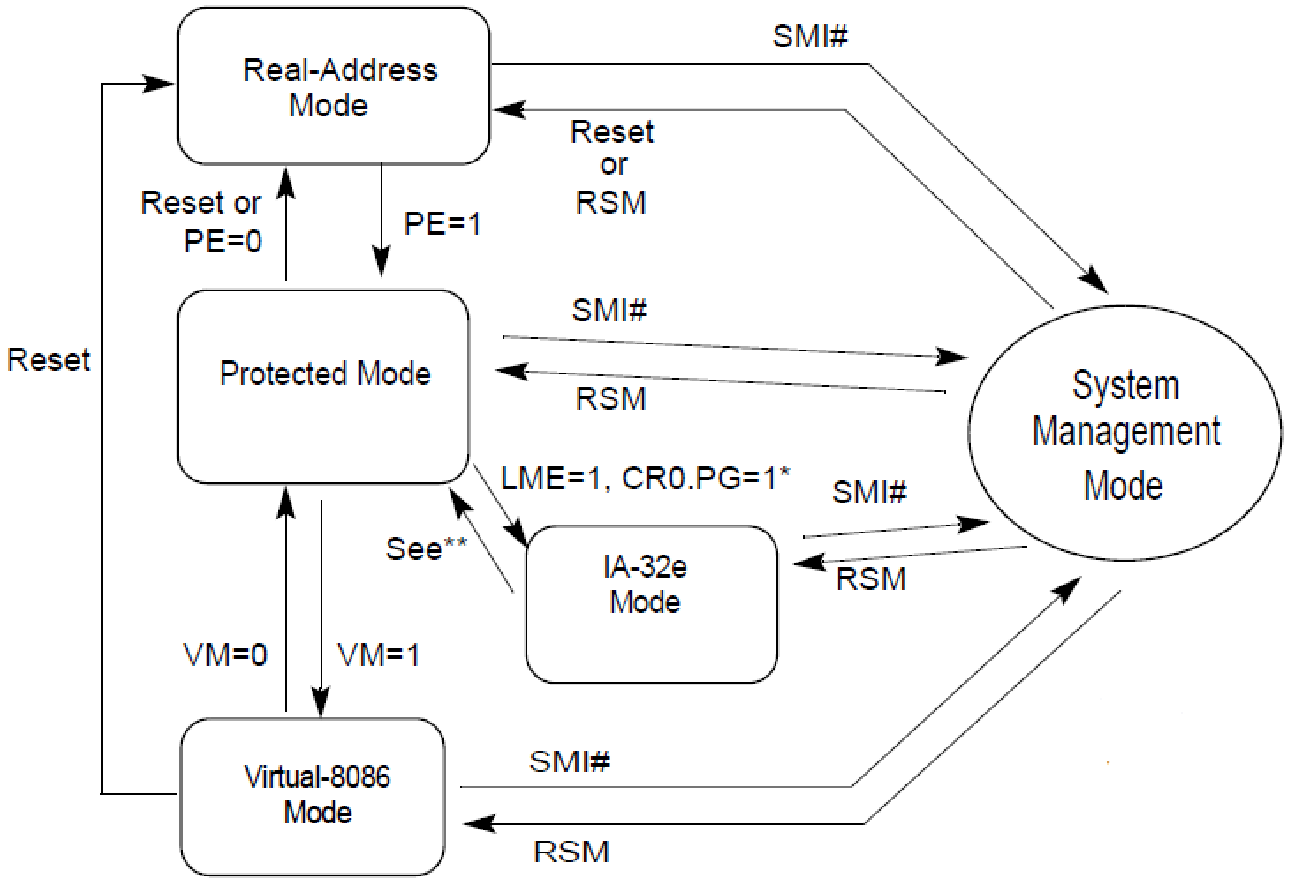
Power-on Self-test (POST)

- Executed very early in the BIOS code
- Typical steps:
  - verify CPU registers
  - verify the integrity of the BIOS code itself
  - verify some basic components like DMA, timer, interrupt controller
  - find, size, and verify system main memory
  - initialize BIOS
  - pass control to other specialized extension BIOSes (if installed)
  - identify, organize, and select which devices are available for booting, organize, and select which devices are available for booting

Real mode

- In real mode, memory is limited to 1MB
- Upon power up, top 12 address lines asserted high, forcing address to 0xFFFxxxxx
- If no mode transition after first long jump, enters real mode
  - Memory limited to 1MB

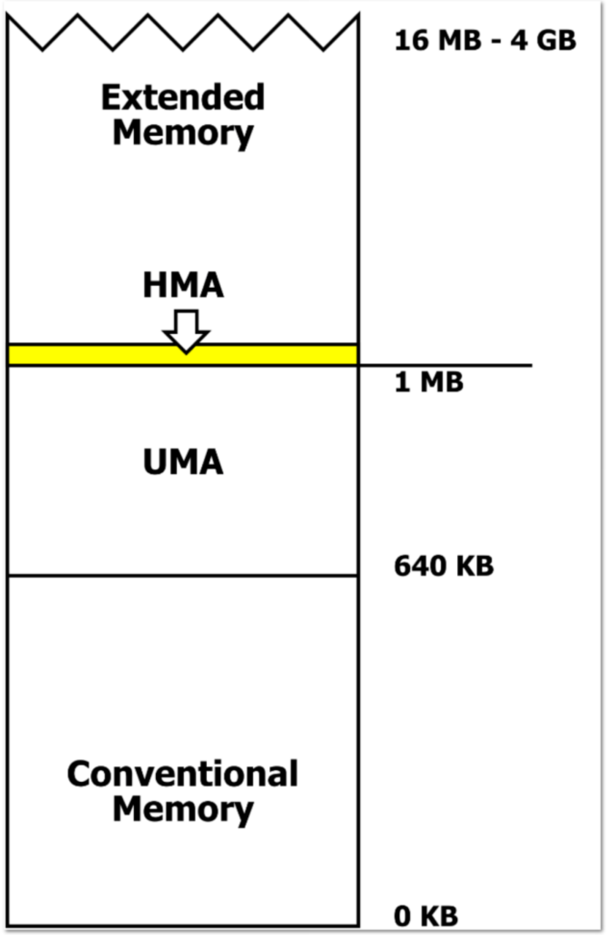
X86 mode changes



DOS memory management (Real mode)

- Upper memory area (UMA) refers to the address space between 640 KB and 1024 KB (0xA0000–0xFFFFF).
- Three 128 KB regions were defined in this area.
  - The 128 KB region between 0xA0000 and 0xBFFFF was reserved for video adapter screen memory.
  - The 32 KiB region between 0xC0000 and 0xC7FFF was reserved for video adapter Video BIOS memory.
  - The 160 KiB region between 0xC8000 and 0xEFFFF was reserved for device Option ROMs, and special RAM usually shared with physical devices

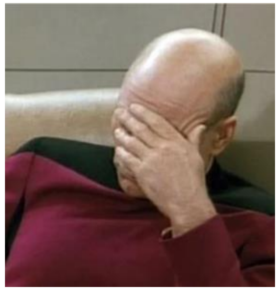
- Wikipedia



Digression – a pain from the past: A20

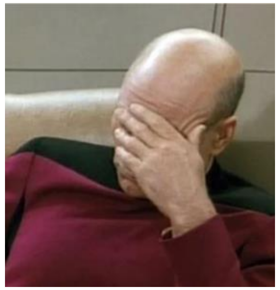
- The original 8086 supports 1MB of memory.
- But with real-address mode segmentation, one can have a segment register with value 0xFFFF and an offset of 0xFFFF
  - Written as 'FFFF:FFFF'
- FFFF:FFFF gives an effective address of 0x10FFEF (which is 21 bits!)
  - On 8086, it is 'wrapped around' to yield the address 0x0FFEF
- 80286 onwards gives an additional A20 (the 21st) address line
  - AND THE OPTION TO TURN IT ON OR OFF
  - VIA PROGRAMMING THE KEYBOARD!

https://wiki.osdev.org/A20_Line



The "Unreal" Mode

A small digression



“Unreal” Mode

- Real mode 1MB memory constraint is too restrictive
- A clever exploit of the Intel ISA allows for real mode addressing to go beyond 1MB
- The main hurdle is the limit of using only 20 bit addresses
- To get into “unreal” mode, need to do a real mode to protected mode switch and then back to real mode

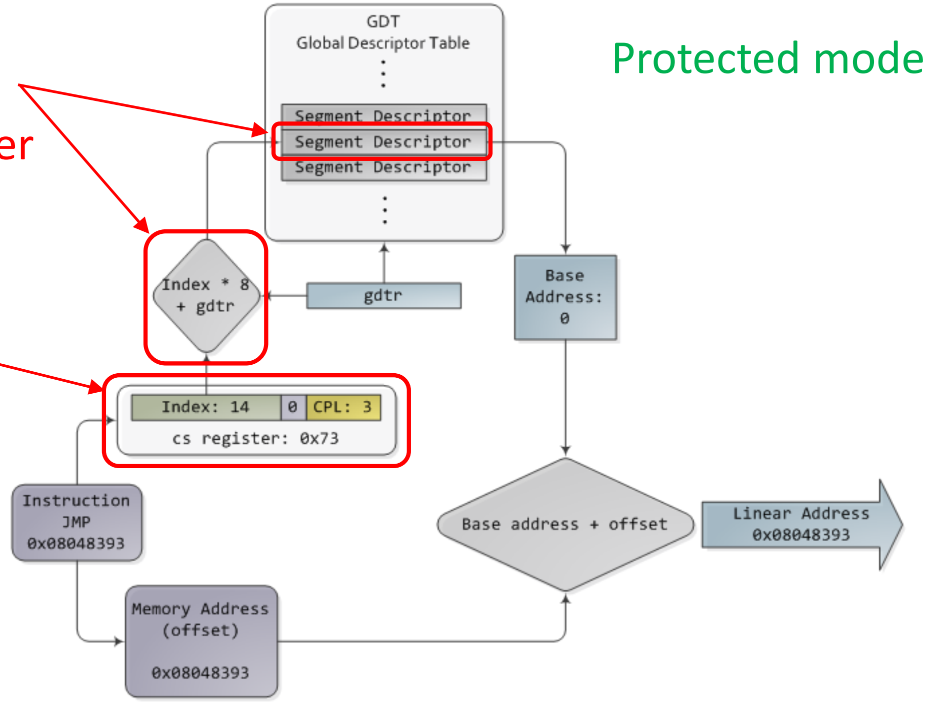
Segment Descriptor Cache Register

- Used to speed up protected mode segment addressing

Cache it!

then no change in these either

If no change



Protected mode addressing

Segment Descriptor Cache

Table 1 - 80286 Descriptor Cache Entry

Bit Position	47..32	31	30..29	28	27..24	23..00
Description	16-bit Limit	P	DPL	S	Type	24-bit Base

Table 2 - 80386 and 80486 Descriptor Cache Entry

Bit Position	95..64	63..32	31..24	23	22..21	20	19..16	15	14	13..0
Description	32-bit Limit	32-bit Base	0	P	DPL	S	Type	0	D / B	0

Table 3 - Pentium Descriptor Cache Entry

Bit Position	95..79	78	77..72	71	70..69	68	67..64	63..32	31..00
Description	0	D/B	0	P	DPL	S	Type	32-bit Base	32-bit Limit

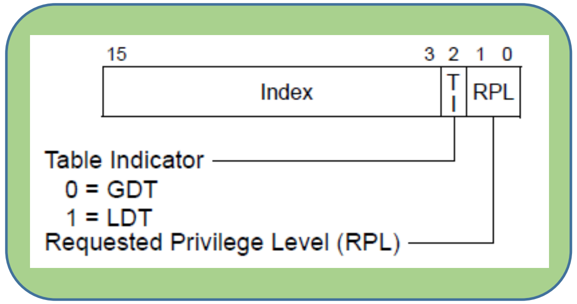
Table 4 - Pentium Pro Descriptor Cache Entry

Bit Position	95..64	63..32	31	30	29..24	23	22..21	20	19..16	15..00
Description	32-bit Base	32-bit Limit	0	D/B	0	P	DPL	S	Type	Segment Selector

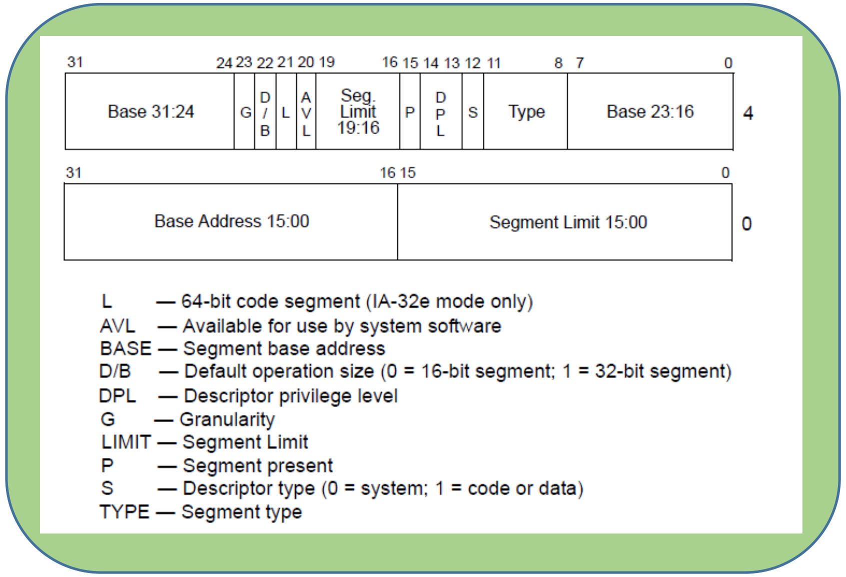
Format varies but… after 80386 all segment limits are 32 bits

Using “Unreal” Mode - 1

- Start in real mode. Do a switch to protected mode.
- Set up the descriptors, selectors and GDT. Most important, set limit to 4GB. Make sure privilege level set appropriately.



Segment Selector



Segment Descriptor

Using "Unreal" Mode - 2

- Switch back to real mode. Cache limit remains and takes precedence.
- Use the "operand size prefix" (0x66) in instructions to enable the use of longer than 16 bit offset registers.
- In (normal) real mode, address = seg * 16 + offset. If total sum exceeds 1MB, instruction fault.
- In "unreal" mode, offset can now be full 32 bits (using the operand size prefix for some instructions) so while still address = seg * 16 + offset, since offset can be 32 bits, address can be 32 bits.

“Unreal” mode - caveat

- Real mode interrupt assumes 16 bits IP. Code segment above 64KB needs special attention.
- Unreal mode can be used by boot routines or for writing/supporting backward compatible applications
  - E.g. DOSBox
- Requires special compiler support to generate the prefix

Back to our boot story…

BIOS software setup

- At least must load an Interrupt Descriptor Table (IDT), a Global Descriptor Table (GDT), a Task-State Segment (TSS), and, optionally, a Local Descriptor Table (LDT).
- Must also initialize some critical control registers and the memory type range registers (MTRR)
- Once these set up, ready to boot the OS.
  - Note: still in real mode!

Memory Type Range Registers (MTRR)

- MTRRs provide a mechanism for associating the memory types with physical-address ranges in system memory
- Allows up to 96 memory ranges to be defined in physical memory

Table 11-2. Memory Types and Their Properties

Memory Type and Mnemonic	Cacheable	Writeback Cacheable	Allows Speculative Reads	Memory Ordering Model
Strong Uncacheable (UC)	No	No	No	Strong Ordering
Uncacheable (UC-)	No	No	No	Strong Ordering. Can only be selected through the PAT. Can be overridden by WC in MTRRs.
Write Combining (WC)	No	No	Yes	Weak Ordering. Available by programming MTRRs or by selecting it through the PAT.
Write Through (WT)	Yes	No	Yes	Speculative Processor Ordering.
Write Back (WB)	Yes	Yes	Yes	Speculative Processor Ordering.
Write Protected (WP)	Yes for reads; no for writes	No	Yes	Speculative Processor Ordering. Available by programming MTRRs.

"Pre-memory"

- Memory itself is a hardware resource and the memory controller itself must be set up
- Prior to the set up (including testing), no DRAM is available
  - Thus far we are in *read-only* ROM BIOS and register land
- In particular, no stack – hence no procedure call
- To circumvent this, one way is to use the cache as memory
  - "No evict mode" – no eviction on miss
  - By setting the appropriate MTRR
  - Must "undo" after memory is available

“Post-memory”

- With memory up and available, BIOS is ready to load in OS
- Initialize secondary storage devices, scan boot sequence to find the master boot record (MBR) and verify its integrity
- If all goes well, load the bootstrap loader – the code section of the MBR
    - For Intel, physical address 0x7C00
- Does a jump to the bootstrap loader

Intel Memory Map at Boot up

```
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
```

Master Boot Record

Offset	Size (bytes)	Description
0	436 (to 446, if you need a little extra)	MBR **Bootstrap** (flat binary executable code)
0x1b4	10	Optional "unique" disk ID[1]
0x1be	64	MBR **Partition Table**, with 4 entries (below)
0x1be	16	First partition table entry
0x1ce	16	Second partition table entry
0x1de	16	Third partition table entry
0x1ee	16	Fourth partition table entry
0x1fe	2	(0x55, 0xAA) "Valid bootsector" signature bytes

Master Boot Record

Element (offset)	Size	Description
0	byte	Boot indicator bit flag: 0 = no, 0x80 = bootable (or "active")
1	byte	Starting head
2	6 bits	Starting sector (Bits 6-7 are the upper two bits for the Starting Cylinder field.)
3	10 bits	Starting Cylinder
4	byte	System ID
5	byte	Ending Head
6	6 bits	Ending Sector (Bits 6-7 are the upper two bits for the ending cylinder field)
7	10 bits	Ending Cylinder
8	uint32_t	Relative Sector (to start of partition -- also equals the partition's starting LBA value)
12	uint32_t	Total Sectors in partition

Partition Table Entry

Disk partitions

- Disk partitioning is a way of organizing a physical disk.
- A physical disk can contain multiple partitions, each a logical disk
- Advantages:
  - Separate uses: OS, swap disk, user files
  - Some protection from misuse and corruption
  - “Short stroking”: reduce seek time as files are more localized
- Disadvantages:
  - Increases fragmentation
  - Can restrict use

And in comes GRUB

- For Linux, MBR is the first stage of GRUB, boot.img (GRUB stage 1)
  - See http://www.funtoo.org/Boot_image for annotated GRUB MBR
- Up to now, everything is in real mode (or possibly unreal mode)
- Boot.img locates and reads in diskboot.img and jumps to it (at location 0x2000)
- Diskboot.img loads in the rest of GRUB (GRUB Stage 1.5)
  - Including drivers for handling (key) file systems
- Control is transferred to grub_main() finally

The Linux/x86 Boot Protocol

- https://www.kernel.org/doc/Documentation/x86/boot.txt

- Dictates the memory layout, structures and initial values expected by the Linux kernel before and during the boot process

grub_main() – GRUB Stage 2

- Initializes the console
- Gets the base address for modules
- Sets the root device
- Loads/parses the grub configuration file
- Loads modules
- Fill in the Linux kernel header

The Linux Kernel Header

Offset /Size	Proto	Name	Meaning
01F1/1	ALL(1	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+(2	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+(3	ext_loader_ver	Extended boot loader version
0227/1	2.02+(3	ext_loader_type	Extended boot loader ID
0228/4	2.02+	cmd_line_ptr	32-bit pointer to the kernel command line
022C/4	2.03+	initrd_addr_max	Highest legal initrd address
0230/4	2.05+	kernel_alignment	Physical addr alignment required for kernel
0234/1	2.05+	relocatable_kernel	Whether kernel is relocatable or not
0235/1	2.10+	min_alignment	Minimum alignment, as a power of two
0236/2	2.12+	xloadflags	Boot protocol option flags
0238/4	2.06+	cmdline_size	Maximum size of the kernel command line
023C/4	2.07+	hardware_subarch	Hardware subarchitecture
0240/8	2.07+	hardware_subarch_data	Subarchitecture-specific data
0248/4	2.08+	payload_offset	Offset of kernel payload
024C/4	2.08+	payload_length	Length of kernel payload
0250/8	2.09+	setup_data	64-bit physical pointer to linked list of struct setup_data
0258/8	2.10+	pref_address	Preferred loading address
0260/4	2.10+	init_size	Linear memory required during initialization
0264/4	2.11+	handover_offset	Offset of handover entry point

arch/x86/boot/header.S

Memory map after kernel load

GRUB loads in both the kernel and RAMdisk. It will exceed the 1MB boundary. So it will transit to protected mode then load.

```
          | 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 0x7C00
001000    +------------------------+
          | Reserved for MBR/BIOS  |
000800    +------------------------+
          | Typically used by MBR  |
000600    +------------------------+
          | BIOS use only          |
000000    +------------------------+
```

GRUB notation

```
(hd0,1)
```

- hd stands for hard disk; alternatively, fd stands for floppy disk, cd stands for CD-ROM etc.
- The first number refers to the physical hard drive number, starting from 0.
- The second number refers to the partition number of the selected hard drive; again, starting from 0.

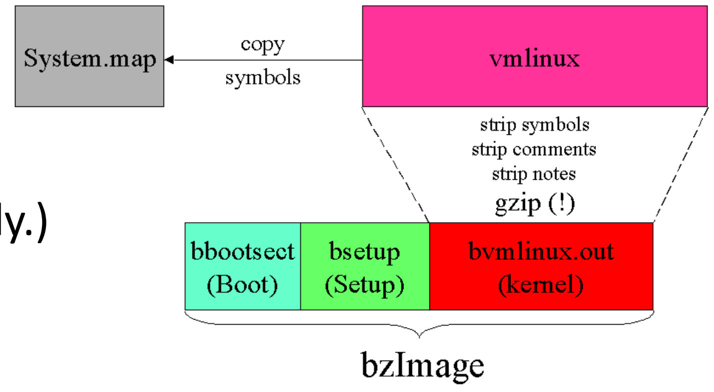
GRUB Configuration

```
# grub.conf generated by anaconda
#
# Note that you do not have to rerun grub after making changes to this file
# NOTICE:  You have a /boot partition.  This means that
#          all kernel and initrd paths are relative to /boot/, eg.
#          root (hd0,0)
#          kernel /vmlinuz-version ro root=/dev/sda2
#          initrd /initrd-version.img
#boot=/dev/sda
default=0
timeout=5
splashimage=(hd0,0)/grub/splash.xpm.gz
hiddenmenu

#####First Operating System#####

title Red Hat Enterprise Linux Server (2.6.18-8.el5)
     root (hd0,0)
     kernel /vmlinuz-2.6.18-8.el5 ro root=LABEL=/ rhgb quiet
     initrd /initrd-2.6.18-8.el5.img

#####Second Operating System#####

title RedHat Operating System 2
     root(hd1,0)
     kernel /vmlinuz-2.6.18-8.el5 ro root=/dev/sdb2 rhgb quiet
     initrd /initrd-2.6.18-8.el5.img
```

The Linux kernel

- vmlinux - a statically linked executable file that contains the entire kernel
- vmlinuz – compressed vmlinux
- zImage – old kernel below 512KB that is loaded into low memory
- bzImage – larger kernel loaded at above 1MB

Anatomy of bzImage



(These days use vmlinuz directly.)

initrd

- Zen koan: “Before you were born, what was your face?”
- Before the OS gets booted, there is no drivers or file system, hence no files
- But the boot process requires an environment to work in
- Answer: a pre-laid out minimum file system that is so small that it can fit in RAM
  - Recall: BIOS brought DRAM on board, albeit in real mode

initrd

- "Initial RAMdisk": a temporary root file system in memory
  - Minimal set of directories and executables to support 2nd stage booting

```
# ls -la
#
drwxr-xr-x  10 root root    4096 May 7 02:48 .
drwxr-x---  15 root root    4096 May 7 00:54 ..
drwxr-xr-x  2  root root    4096 May 7 02:48 bin
drwxr-xr-x  2  root root    4096 May 7 02:48 dev
drwxr-xr-x  4  root root    4096 May 7 02:48 etc
-rwxr-xr-x  1  root root     812 May 7 02:48 init
-rw-r--r--  1  root root 1723392 May 7 02:45 initrd-2.6.14.2.img
drwxr-xr-x  2  root root    4096 May 7 02:48 lib
drwxr-xr-x  2  root root    4096 May 7 02:48 loopfs
drwxr-xr-x  2  root root    4096 May 7 02:48 proc
lrwxrwxrwx  1  root root       3 May 7 02:48 sbin -> bin
drwxr-xr-x  2  root root    4096 May 7 02:48 sys
drwxr-xr-x  2  root root    4096 May 7 02:48 sysroot
#
```

initrd vs initramfs

Initrd:

- A full file system image (may be compressed)
- Driver for that file system (ext2, cramfs) must be compiled statically into the kernel
- Can use `pivot_root` command to change it later

Initramfs:

- A cpio file archive (like tar)
- Unpacked into a file system of OS's choice (usually tmpfs)
- No need for special drivers compiled into kernel

Kernel boots

- GRUB finally transfers control to `_start` in `arch/x86/boot/header.S`
- Still in real mode!
  - Even though GRUB has loaded kernel and initrd into the higher addresses
- More initializations
  - Make sure that all segment register values are equal
  - Set up a correct stack, if needed
  - Set up bss
- Jump to the C code in main.c

arch/x86/boot/main.c

```
void main(void)
{
        /* First, copy the boot header into the "zeropage" */
        copy_boot_params();

        /* Initialize the early-boot console */
        console_init();
        if (cmdline_find_option_bool("debug"))
                puts("early console in setup code\n");

        /* End of heap check */
        init_heap();

        /* Make sure we have all the proper CPU support */
        if (validate_cpu()) {
                puts("Unable to boot - please use a kernel appropriate "
                     "for your CPU.\n");
                die();
        }

        /* Tell the BIOS what CPU mode we intend to run in. */
        set_bios_mode();

        /* Detect memory layout */
        detect_memory();

        /* Set keyboard repeat rate (why?) and query the lock flags */
        keyboard_init();

        /* Query Intel SpeedStep (IST) information */
        query_ist();

        /* Query APM information */
#if defined(CONFIG_APM) || defined(CONFIG_APM_MODULE)
        query_apm_bios();
#endif

        /* Query EDD information */
#if defined(CONFIG_EDD) || defined(CONFIG_EDD_MODULE)
        query_edd();
#endif

        /* Set the video mode */
        set_video();

        /* Do the last things and invoke protected mode */
        go_to_protected_mode();
}
~
```

Enters protected mode!

Switching to Protected Mode

```c
void go_to_protected_mode(void)
{
        /* Hook before leaving real mode, also disables interrupts */
        realmode_switch_hook();

        /* Enable the A20 gate */
        if (enable_a20()) {
                puts("A20 gate not responding, unable to boot...\n");
                die();
        }

        /* Reset coprocessor (IGNNE#) */
        reset_coprocessor();

        /* Mask all interrupts in the PIC */
        mask_all_interrupts();

        /* Actual transition to protected mode... */
        setup_idt();
        setup_gdt();
        protected_mode_jump(boot_params.hdr.code32_start,
                            (u32)&boot_params + (ds() << 4));
}
```

arch/x86/boot/pm.c

arch/x86/boot/pmjump.S

```asm
/*
 * void protected_mode_jump(u32 entrypoint, u32 bootparams);
 */
GLOBAL(protected_mode_jump)
        movl    %edx, %esi              # Pointer to boot_params table

        xorl    %ebx, %ebx
        movw    %cs, %bx
        shll    $4, %ebx
        addl    %ebx, 2f
        jmp     1f                      # Short jump to serialize on 386/486
1:

        movw    $__BOOT_DS, %cx
        movw    $__BOOT_TSS, %di

        movl    %cr0, %edx
        orb     $X86_CR0_PE, %dl        # Protected mode
        movl    %edx, %cr0

        # Transition to 32-bit mode
        .byte   0x66, 0xea              # ljmpl opcode
2:      .long   in_pm32                 # offset
        .word   __BOOT_CS               # segment
ENDPROC(protected_mode_jump)
```

Decompressing the kernel

- Depending on whether the kernel is 32 bits or 64 bits, one of the following will be (eventually) called (in `arch/x86/boot/compressed/head_{32|64}.S`):
  - `startup_32`
  - `startup_64`
- Build page tables
  - Linux uses 4-level paging
- Decompress kernel via `__decompress()`

We are in!

- After kernel decompression, control is transferred to `startup_{32|64}` of `arch/x86/kernel/head_{32|64}.S`
  - Note that this is a different routine from the ones that triggered decompression

- More initialization
- Eventually `start_kernel()` of `init/main.c` is called

initrd kicks in

- Kernel mounts initrd as root, i.e., “/”

- After initrd is mounted successfully, the function “`run_init_process()`” of `init/main.c` is called. This will execve the first user level process – **init**.
  - /sbin/init
  - /etc/init
  - /bin/init
  - /bin/sh

The init process

- Distro dependent
- Switches from initrd over to a disk root file system
  - Need to first load all the necessary drivers
- init is the parent of all processes
- Bulk of the boot messages originate from this point on
- Culminates in the user selecting a bootlevel

runlevel	directory	meaning
N	none	System bootup (NONE). There is no `/etc/rcN.d/` directory.
0	/etc/rc0.d/	Halt the system.
S	/etc/rcS.d/	Single-user mode on boot. The lower case `s` can be used as alias.
1	/etc/rc1.d/	Single-user mode switched from multi-user mode.
2 ... 5	/etc/rc{2,3,4,5}.d/	Multi-user mode.
6	/etc/rc6.d/	Reboot the system.
7 ... 9	/etc/rc{7,8,9}.d/	Valid multi-user mode but traditional Unix variants don't use. Their `/etc/rc?.d/` directories are not populated when packages are installed.

systemd

systemd

- Alternative to init
- Starts processes in parallel
- Supports interactive booting
- Simpler API
- x86 only (for now)
- Still controversial

Features	init	systemd
Quota Management	No	Yes
Automatic Service Dependency Handling	No	Yes
Kills users Process at logout	No	Yes
Swap Management	No	Yes
SELinux integration	No	Yes
Support for Encrypted HDD	No	Yes
Static kernel module loading	No	Yes
GUI	No	Yes
List all the child processes	No	Yes
Interactive booting	No	Yes
Portable to non x86	Yes	No
Adopted on	Several Distro	Several Distro
Parallel service startup	No	Yes
Resource limit per service	No	Yes
Easy extensible startup script	Yes	No
Separate Code and Configuration File	Yes	No
Automatic dependency calculation	No	Yes
Size	560 KB	N/A
Number of Files	75 files	900 files + glib + DBus
Lines of code – LOC	≈15,000	≈224,000

Source: https://www.tecmint.com/systemd-replaces-init-in-linux/

Configuration directories

- `/etc/systemd` – Local configuration information
- `/run/systemd` – Temporary files (testing, etc.)
- `/usr/lib/systemd` – Vendors

Units

- In systemd, everything is a unit
- Units have
  - A Type
  - A State
  - May have a Status
- A unit has a (usually unique) base-name
- Base-name + Type = complete unit name

Unit types

Service	Mount	Automount	Target
Swap	Socket	Device	Snapshot
Timer	Path	Slice	Scope

Dependencies and Ordering

- In systemd, dependencies and ordering determines when to start and stop units

- Dependencies:
  - "`Wants`" (soft)
  - "`Requires`" (hard)

- Ordering:
  - "`After`"
  - "`Before`"

The Daemons

- `systemd` - The main systemd process. This is the replacement for "/etc/init".
- `journald` - Event logging.
- `networkd` - Manages network configuration.
- `logind` - Manages user logins.
- `udevd` - Manages devices.

Initialization

- When running as “`init`” (PID = 1), the following configuration files are read and acted on accordingly:
  - `/etc/systemd/system.conf`
  - `/etc/systemd/system.conf.d/*.conf`
  - `/run/systemd/system.conf.d/*.conf`
  - `/usr/lib/systemd/system.conf.d/*.conf`

Initialization

- When running as a user processor (PID ≠ 1), the following configuration files are used:
  - `/etc/systemd/user.conf`
  - `/etc/systemd/user.conf.d/*.conf`
  - `/run/systemd/user.conf.d/*.conf`
  - `/usr/lib/systemd/user.conf.d/*.conf`

Unit files

- Each unit has a file in one or more of the following directories:
  - `/etc/systemd/system`
  - `/usr/lib/systemd/system`

- Some units have “drop in” files in:
  - `/etc/systemd/system/<unit>.d/*.conf`
  - `/run/systemd/system/<unit>.d/*.conf`
  - `/usr/lib/systemd/system/<unit>.d/*.conf`

- See: `man systemd.unit`

Using systemd

- Management tools (see their man pages):
  - `systemctl`
  - `journalctl`
  - `systemd-cgls`

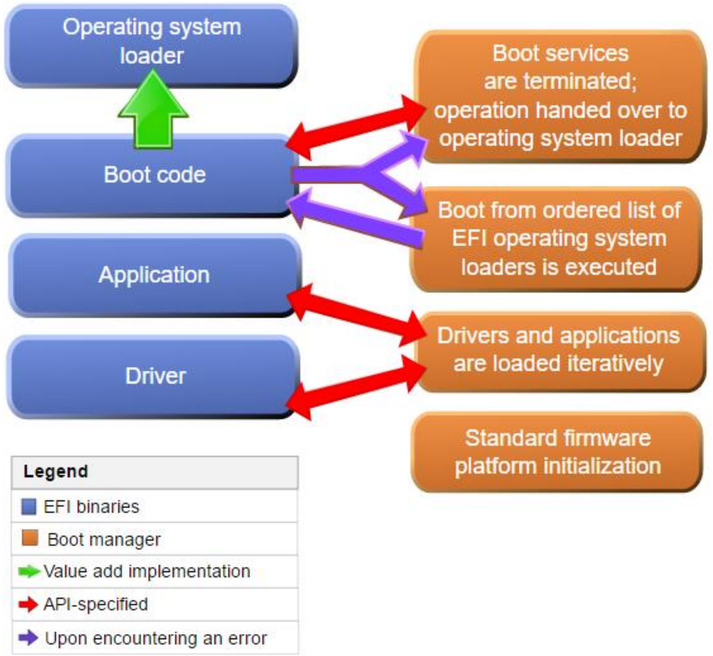
- See: https://www.tecmint.com/systemd-replaces-init-in-linux/

- See: http://0pointer.de/blog/projects/systemd-for-admins-1.html

UEFI

Unified Extensible Firmware Interface (UEFI)

- Successor of traditional BIOS
  - Legacy support usually provided in implementations
- Able to boot from large disks (over 2TB)
- CPU independence
- Modular
- Uses GUID Partition Tables



LBA vs CHS

- Traditionally, disks are addressed using cylinder/head/sector (CHS) tuples
  - Dependent on the physical device
- Logical Block Addressing (LBA) introduced in the SCSI standard as an abstraction
- Current standard is to use 48-bit LBA
  - Access to up to 128 PiB (pebibyte = 1024^5 bytes)

LBA ↔ CHS

$$LBA = (C \times HPC + H) \times SPT + (S - 1)$$

- C, H and S are the cylinder number, the head number, and the sector number
- LBA is the logical block address
- HPC is the maximum number of heads per cylinder (reported by disk drive)
- SPT is the maximum number of sectors per track (reported by disk drive)

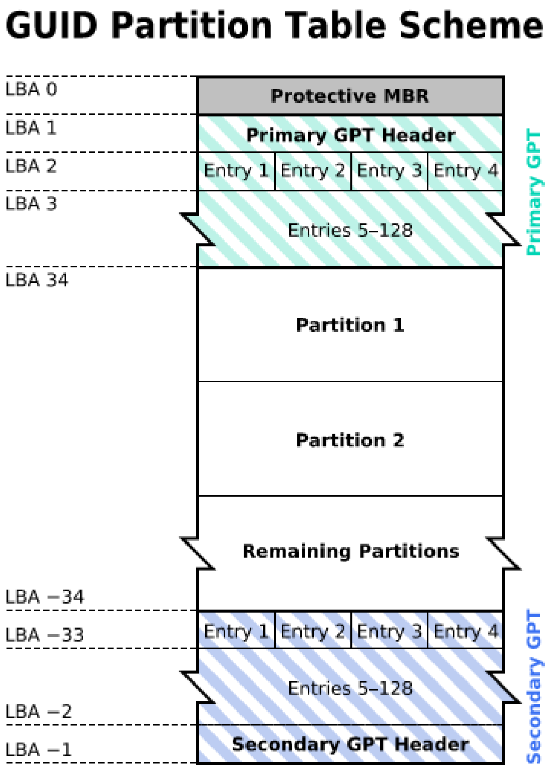
$$C = LBA \div (HPC \times SPT)$$

$$H = (LBA \div SPT) \bmod HPC$$

$$S = (LBA \bmod SPT) + 1$$

GUID Partition Table (GPT) format

- A new partition structure for UEFI
  - Traditional MBR uses 32 bits to store LBA and worse, partitions of up to 2 TiB size
- Now supported by all OSes
- For backward compatibility, LBA 0 is basically the same as MBR
  - Protective MBR – can only boot via EFI
  - Hybrid MBR – contains modified bootloader that knows how to handle GPT in BIOS



Primary GPT Header

Offset	Length	Contents
0 (0x00)	8 bytes	Signature ("EFI PART", 45h 46h 49h 20h 50h 41h 52h 54h or 0x5452415020494645ULL[a] on little-endian machines)
8 (0x08)	4 bytes	Revision (for GPT version 1.0 (through at least UEFI version 2.3.1), the value is 00h 00h 01h 00h)
12 (0x0C)	4 bytes	Header size in little endian (in bytes, usually 5Ch 00h 00h 00h or 92 bytes)
16 (0x10)	4 bytes	CRC32 of header (offset +0 up to header size), with this field zeroed during calculation
20 (0x14)	4 bytes	Reserved; must be zero
24 (0x18)	8 bytes	Current LBA (location of this header copy)
32 (0x20)	8 bytes	Backup LBA (location of the other header copy)
40 (0x28)	8 bytes	First usable LBA for partitions (primary partition table last LBA + 1)
48 (0x30)	8 bytes	Last usable LBA (secondary partition table first LBA - 1)
56 (0x38)	16 bytes	Disk GUID (also referred as UUID on UNIXes)
72 (0x48)	8 bytes	Starting LBA of array of partition entries (always 2 in primary copy)
80 (0x50)	4 bytes	Number of partition entries in array
84 (0x54)	4 bytes	Size of a single partition entry (usually 80h or 128)
88 (0x58)	4 bytes	CRC32 of partition array
92 (0x5C)	*	Reserved; must be zeroes for the rest of the block (420 bytes for a sector size of 512 bytes; but can be more with larger sector sizes)

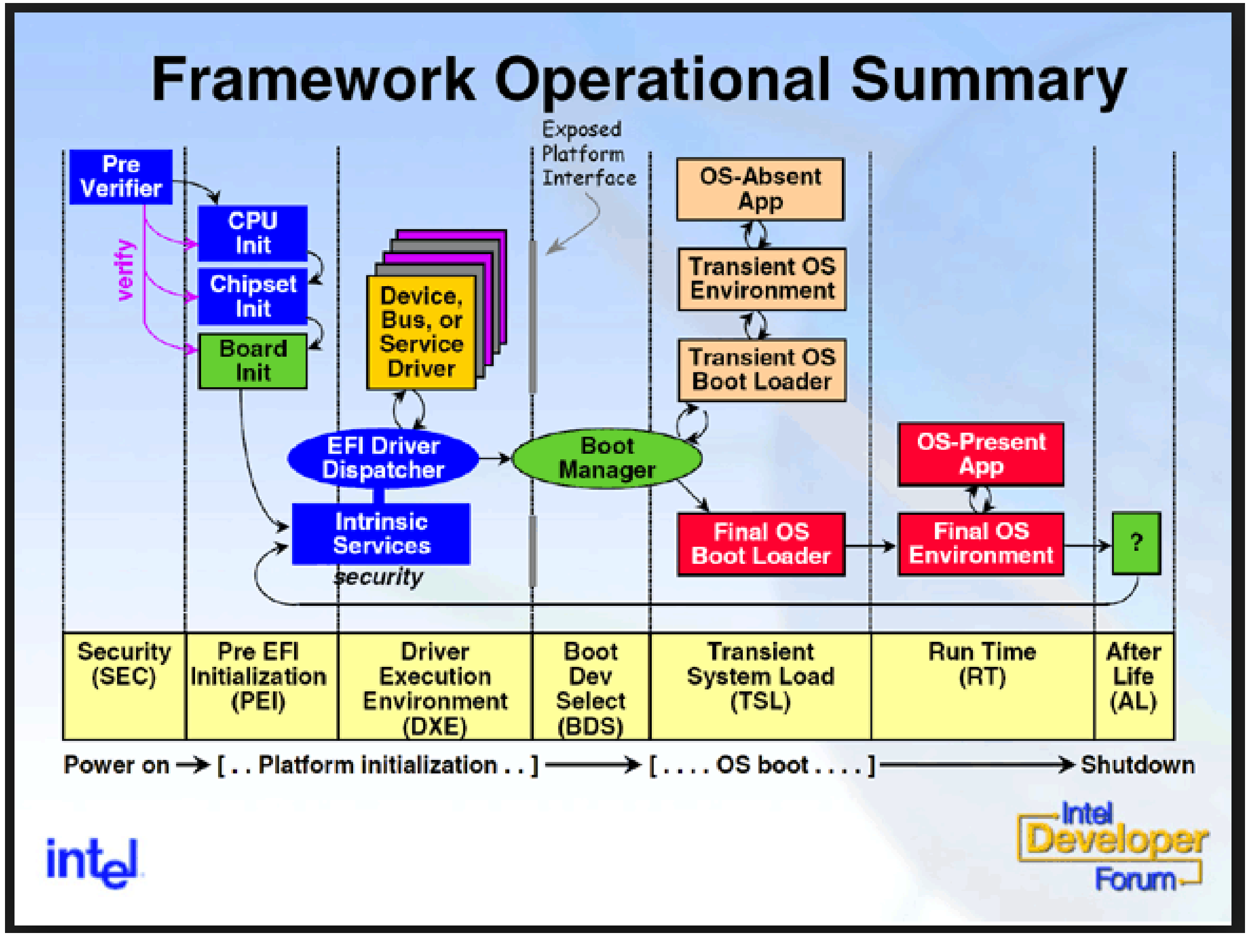
GUID Partition Entry Format

Offset	Length	Contents
0 (0x00)	16 bytes	Partition type GUID
16 (0x10)	16 bytes	Unique partition GUID
32 (0x20)	8 bytes	First LBA (little endian)
40 (0x28)	8 bytes	Last LBA (inclusive, usually odd)
48 (0x30)	8 bytes	Attribute flags (e.g. bit 60 denotes read-only)
56 (0x38)	72 bytes	Partition name (36 UTF-16LE code units)

Global Unique Identifier (GUID)

		Partition Type GUID
(None)	Unused entry	00000000-0000-0000-0000-000000000000
	MBR partition scheme	024DEE41-33E7-11D3-9D69-0008C781F39F
	EFI System partition	C12A7328-F81F-11D2-BA4B-00A0C93EC93B
	BIOS boot partition[e]	21686148-6449-6E6F-744E-656564454649
	Intel Fast Flash (iFFS) partition (for Intel Rapid Start technology)[22][23]	D3BFE2DE-3DAF-11DF-BA40-E3A556D89593
	Sony boot partition[f]	F4019732-066E-4E12-8273-346C5641494F
	Lenovo boot partition[f]	BFBFAFE7-A34F-448A-9A5B-6213EB736C22
Windows	Microsoft Reserved Partition (MSR)	E3C9E316-0B5C-4DB8-817D-F92DF00215AE
	Basic data partition[g]	EBD0A0A2-B9E5-4433-87C0-68B6B72699C7
	Logical Disk Manager (LDM) metadata partition	5808C8AA-7E8F-42E0-85D2-E1E90434CFB3
	Logical Disk Manager data partition	AF9B60A0-1431-4F62-BC68-3311714A69AD
	Windows Recovery Environment	DE94BBA4-06D1-4D40-A16A-BFD50179D6AC
	IBM General Parallel File System (GPFS) partition	37AFFC90-EF7D-4E96-91C3-2D7AE055B174
	Storage Spaces partition	E75CAF8F-F680-4CEE-AFA3-B001E56EFC2D
Linux	Linux filesystem data[g]	0FC63DAF-8483-4772-8E79-3D69D8477DE4
	RAID partition	A19D880F-05FC-4D3B-A006-743F0F84911E
	Root partition (x86)[26]	44479540-F297-41B2-9AF7-D131D5F0458A
	Root partition (x86-64)[26]	4F68BCE3-E8CD-4DB1-96E7-FBCAF984B709
	Root partition (32-bit ARM)[26]	69DAD710-2CE4-4E3C-B16C-21A1D49ABED3
	Root partition (64-bit ARM/AArch64)[26]	B921B045-1DF0-41C3-AF44-4C6F280D3FAE
	Swap partition	0657FD6D-A4AB-43C4-84E5-0933C84B4F4F
	Logical Volume Manager (LVM) partition	E6D6D379-F507-44C2-A23C-238F2A3DF928
	`/home` partition[26]	933AC7E1-2EB4-4F13-B844-0E14E2AEF915
	`/srv` (server data) partition[26]	3B8F8425-20E0-4F3B-907F-1A25A76F98E8
	Reserved	8DA63339-0007-60C0-C436-083AC8230908

UEFI Boot Process



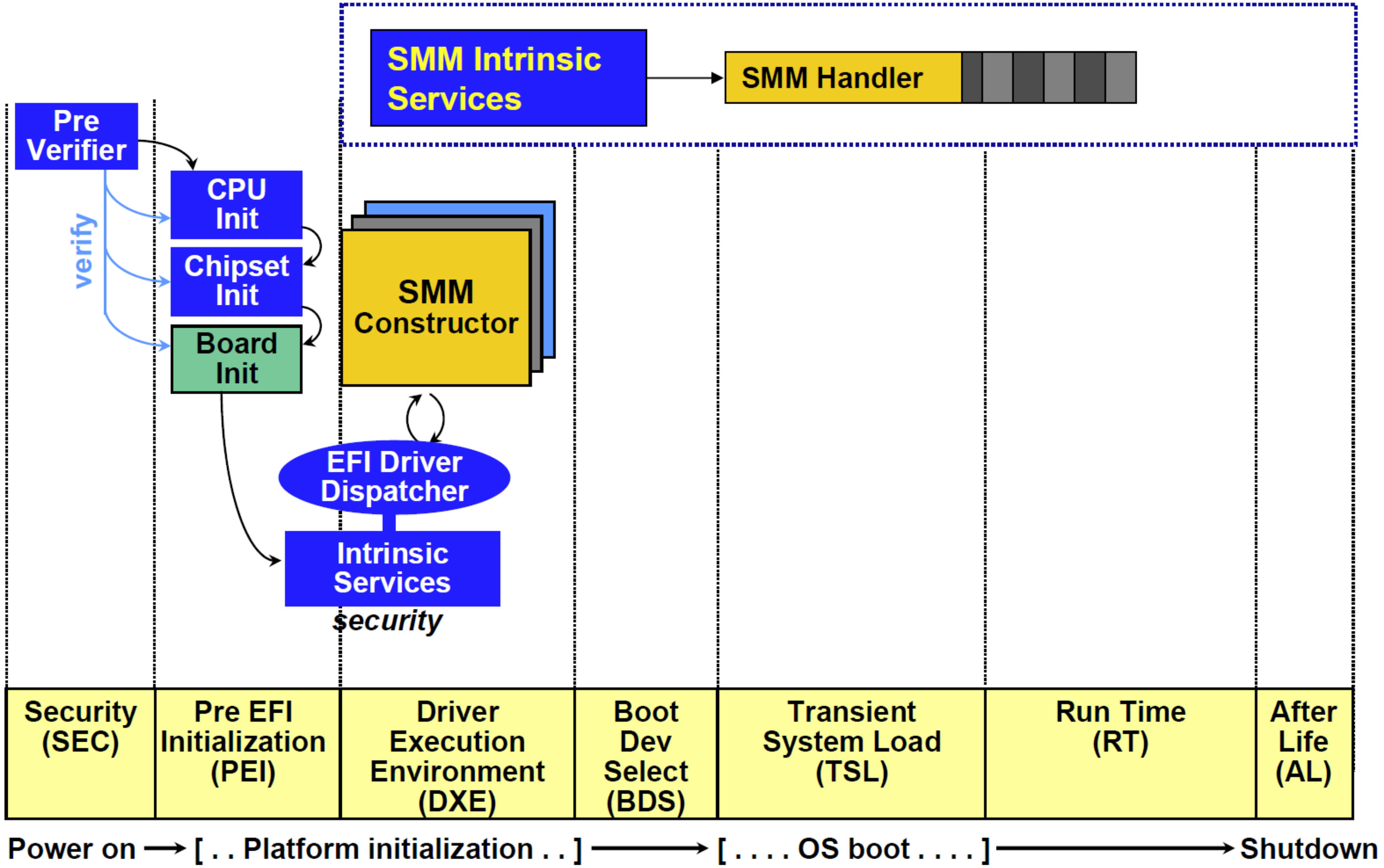
UEFI Boot Process

- SEC
  - Target of reset vector
  - Verify integrity of the PEI code
  - Set up Cache-as-RAM
  - Runs the PEI
- PEI
  - Initializes hardware
  - Loads and execute DXE

DXE (Driver Execution Environment)

- Almost a mini-OS (complex)
  - When it is up, there is a file system with various bootloader executables
- Goal: setup all the necessary drivers so that the actual OS's bootloader has most of what it needs to work with
  - Architectural abstraction
- Finds the EFI_SYSTEM_PARTITION, sets up and mount FAT32 file system
- Depending on user preferences, execute the boot loader
  - For Linux, usually GRUB 2

UEFI and SMM


SMM Intrinsic Services
SMM Handler
Pre Verifier
CPU Init
Chipset Init
Board Init
verify
SMM Constructor
EFI Driver Dispatcher
Intrinsic Services
security
Security (SEC)
Pre EFI Initialization (PEI)
Driver Execution Environment (DXE)
Boot Dev Select (BDS)
Transient System Load (TSL)
Run Time (RT)
After Life (AL)
Power on → [. . Platform initialization . .] → [. . . . OS boot . . . .] → Shutdown

BIOS vs UEFI boot

- UEFI does not run the MBR bootstrap loader
  - No 446 byte limit

- UEFI loads a FAT32 file system driver and sets up a FAT32 root file system

End