Bootloader

IOC5226 Operating System Capstone

Tsung Tai Yeh
Department of Computer Science
National Yang Ming Chiao Tung University

Acknowledgements and Disclaimer

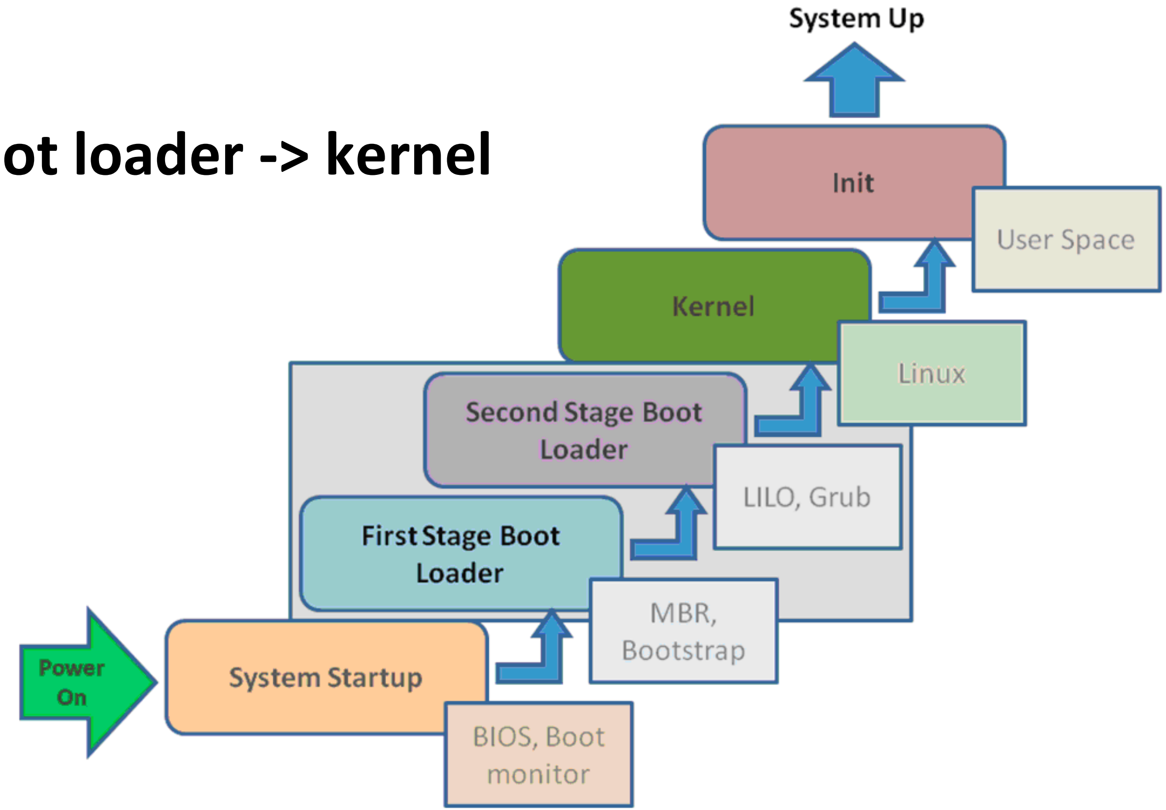
- Slides were developed in the reference with
  - MIT 6.828 Operating system engineering class, 2018
  - MIT 6.004 Operating system, 2018
  - Remzi H. Arpaci-Dusseau etl. , Operating systems: Three easy pieces. WISC

Outline

- Booting on the X86 Processor
  - BIOS
  - MBR
  - Bootloader
- Booting on rpi
- Linux Bootstraping

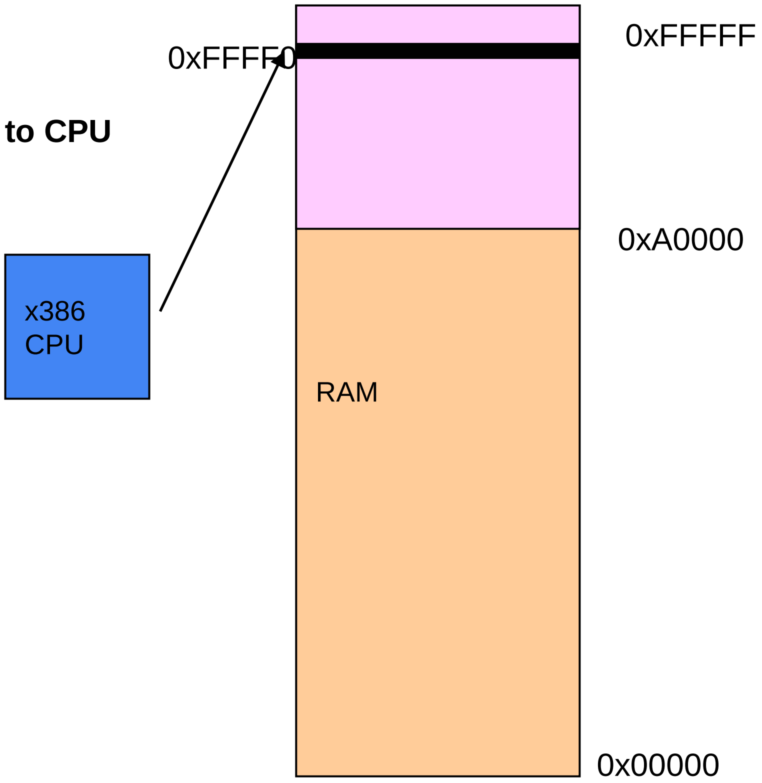
Booting on x86 processor

BIOS -> MBR -> boot loader -> kernel



PC Booting

1. **Power supply sends POWER GOOD to CPU**
2. **CPU resets**
3. **Run FFFF:0000 @ BIOS ROM**
4. **Jump to a real BIOS start address**
5. Power On Self Test (**POST)**
6. **Beep if there is an error**
7. **Read CMOS data/settings**
8. **Run 2nd-stage boot**

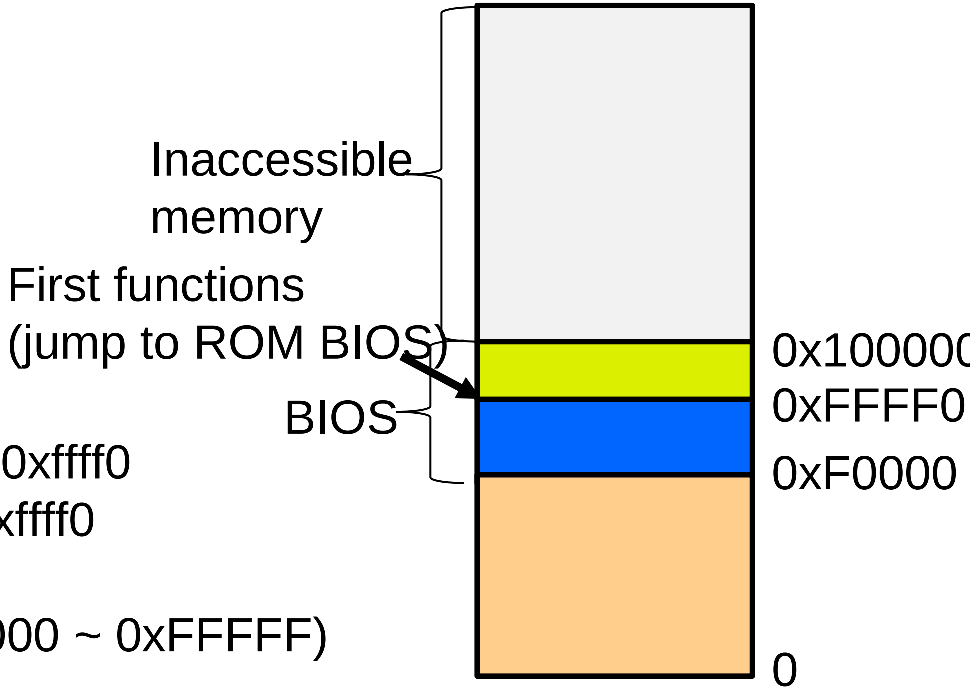


Powering up: Reset

Power on Reset

8086 CPU

Every register initialized to 0 except
CS = 0xf000, IP = 0xfff0

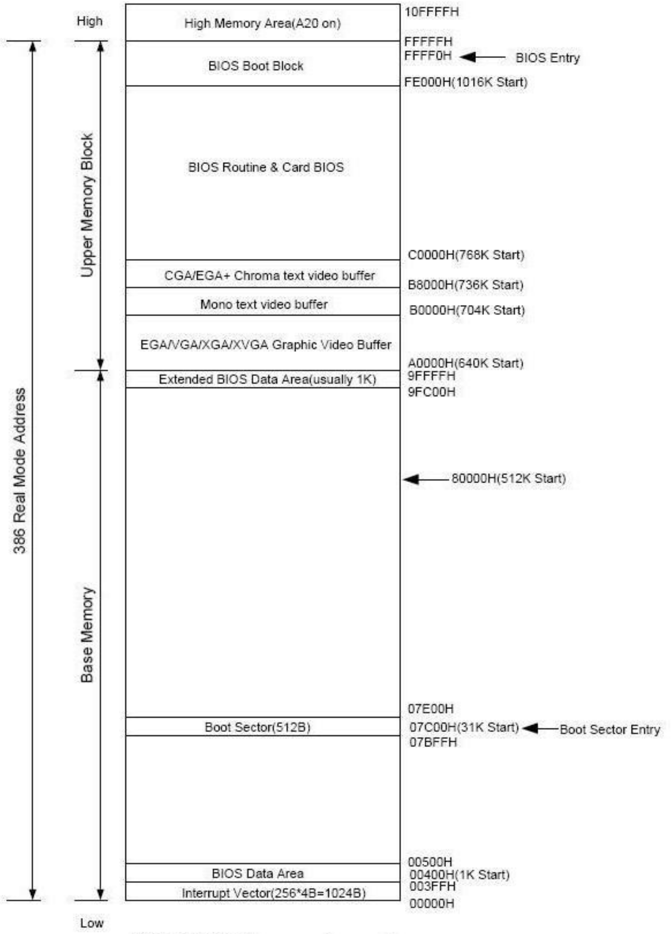


1. Physical address = (CS << 4) + IP = 0xffff0
2. First instruction fetch from location 0xffff0
3. Processor in real mode (20-bits)
   a. Limited to 1MB addresses (0x00000 ~ 0xFFFFF)
   b. No protection; no privilege levels
   c. Direct access to all memory
   d. No multi-tasking
4. First instruction is on the top of accessible memory

http://www.cse.iitm.ac.in/~chester/courses/16o_os/slides/3_Hardware.pdf

Powering up: BIOS

- BIOS presents in a small chip connected to processor
  - Flash/EPROM/EEPROM
- **BIOS work**
  - Power on self test
  - Initialize video card and other devices
  - Display BIOS screen
  - Perform brief memory test
  - Set DRAM memory parameters
  - Configure plug & play devices
  - Assign DMA channels and IRQs
  - The reset vector (0x7c00) contains a jump (jmp) instruction that usually points to the BIOS entry point



PC BIOS Memory Layout

https://www.twblogs.net/a/5b7f367e2b717767c6ae576c

Powering up: BIOS

- 0x00000 ~ 0x9FFFF (Base Memory) 640 KB

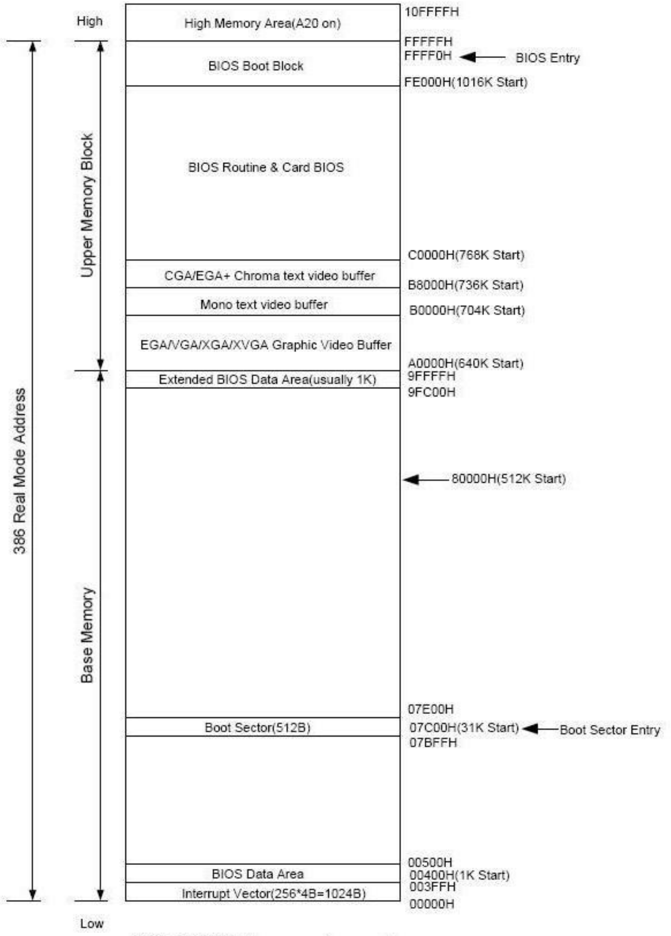
```
0x00000 - 0x003FF: 中斷向量表 (1024B)
0x00400 - 0x004FF: bios數據區 (256B)
0x00500 - 0x07BFF: 自由內存區
0x07C00 - 0x07DFF: 引導程序加載區 (512B)
0x07E00 - 0x9FFFF: 自由內存區
```

- 0xA0000 ~ 0xBFFFF (for VGA) 128 KB

```
0xA0000 - 0xAFFFF: EGA/VGA/XGA/XVGA圖形視頻緩衝區 (64KB)
0xB0000 - 0xB7FFF: Mono text video buffer (32KB)
0xB8000 - 0xBFFFF: CGA/EGA+ chroma text video buffer (32KB)
```

- 0xC0000 ~ 0xFFFFF (BIOS)

```
0xC0000 - 0xC7FFFF: 顯卡bios使用 (32KB)
0xC8000 - 0xCBFFFF: ide控制器bios使用 (16KB)
0xCC000 - 0xEFFFFF:
0xF0000 - 0xFFFFFF: 系統bios使用 (64KB)
```

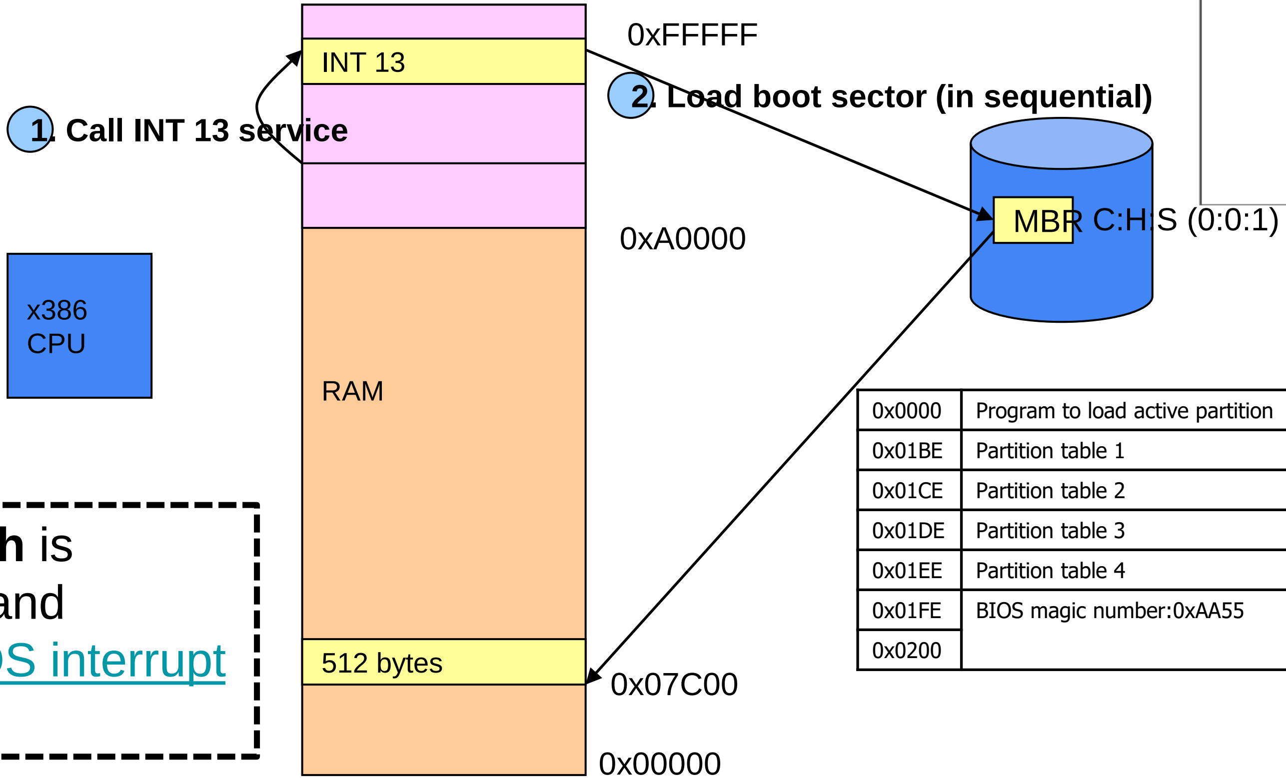


PC BIOS Memory Layout

https://www.twblogs.net/a/5b7f367e2b717767c6ae576c

PC Booting (Cont)

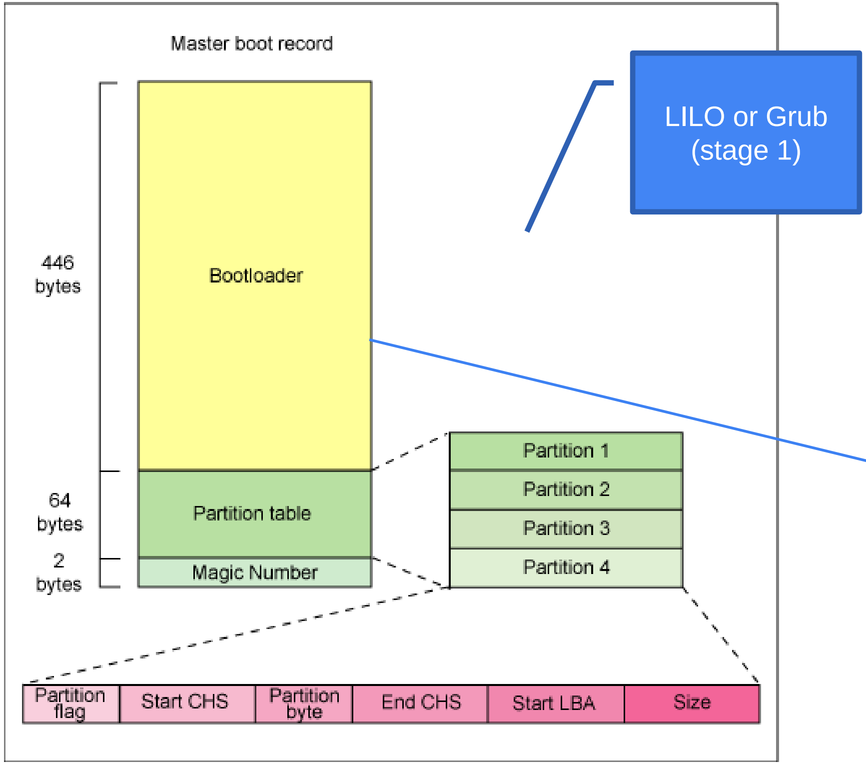
INT 13h	低階磁碟服務。 AH=00h 復位磁碟機。 AH=01h 檢查磁碟機狀態。 AH=02h 讀磁區。 AH=03h 寫磁區。 AH=04h 校驗磁區。 AH=05h 格式化磁軌。 AH=08h 取得驅動器參數。 AH=09h 初始化硬碟機參數。 AH=0Ch 尋道。 AH=0Dh 復位硬碟控制器。 AH=15h 取得驅動器類型。 AH=16h 取得軟碟機中碟片的狀態。



0x0000	Program to load active partition
0x01BE	Partition table 1
0x01CE	Partition table 2
0x01DE	Partition table 3
0x01EE	Partition table 4
0x01FE	BIOS magic number:0xAA55
0x0200	

INT 13h is shorthand for BIOS interrupt call

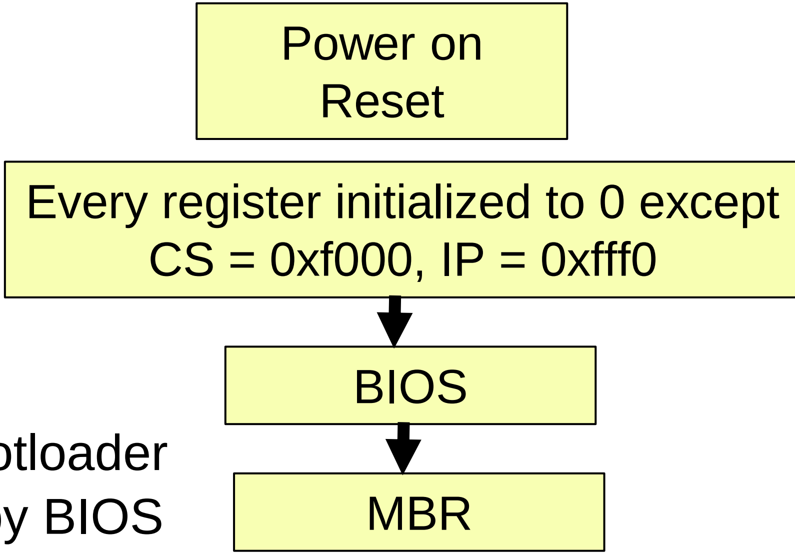
MBR (Master Boot Record)



1. **Partition table**: describes the partitions of a storage device
2. **Bootstrap code**: instructions to identify the configured bootable partition

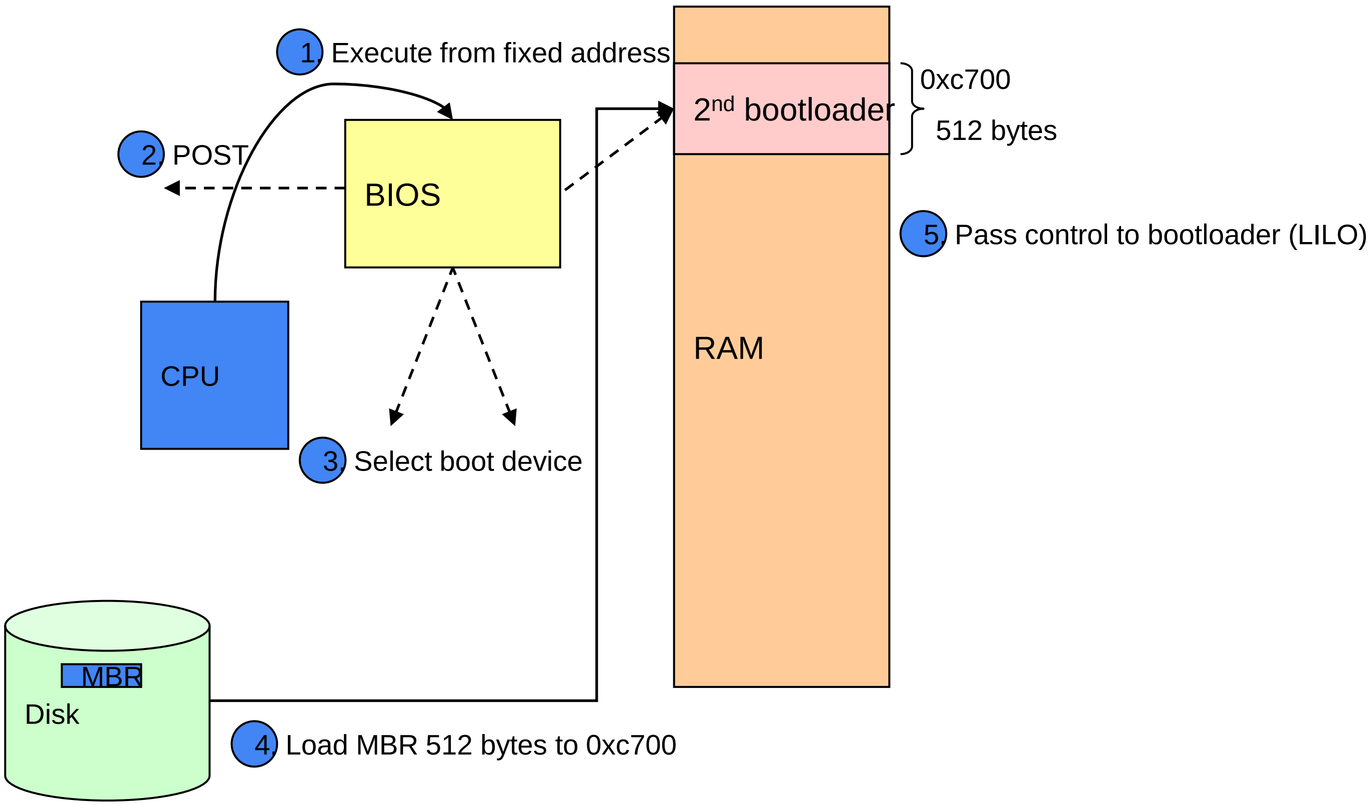
1st stage bootloader

Powering up: MBR

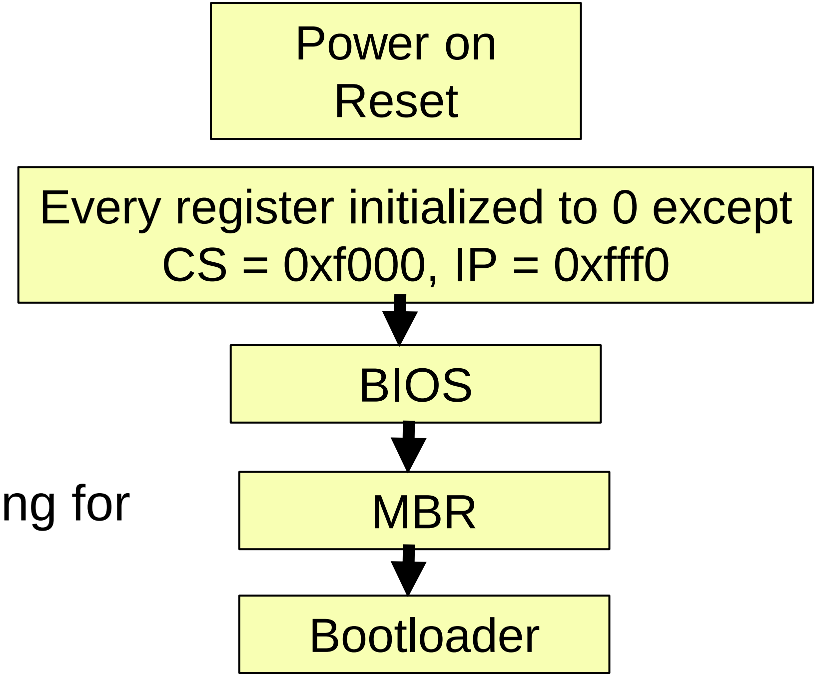


- **Sector 0 in the disk called Master Boot Record (MBR)**
    - Includes code that boots the OS or bootloader
    - Copied from disk to RAM (@0x7c00) by BIOS
    - Size: 512 bytes
    - 446 bytes bootable code
    - 64 bytes disk partition information (16 bytes per partition)
    - MBR looks through partition table and loads the bootloader such as Linux or Windows
    - Or MBR may directly load the OS

Linux Boot Example



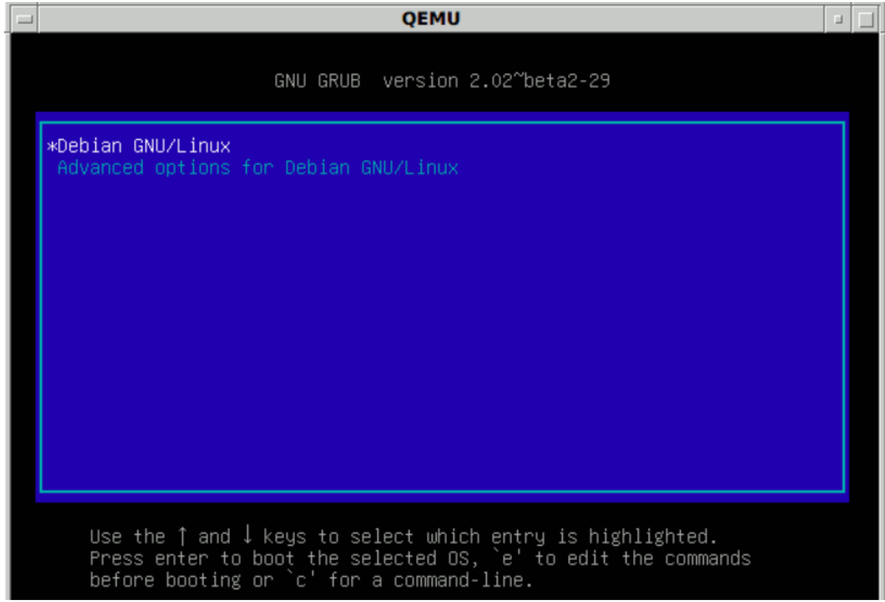
Powering up: bootloader



- Objective of the bootloader
  - After BIOS
  - INIT hardware devices
  - Build sound hardware/software setting for the OS kernel
- Other jobs done
  - Setup GDT (global descriptor table)
  - Switch from real mode to protected mode
  - Read operating system from disk
  - The 1st bootloader may be presented in the MBR (sector 0)
  - The 2nd bootloader -> GRUB/LILO

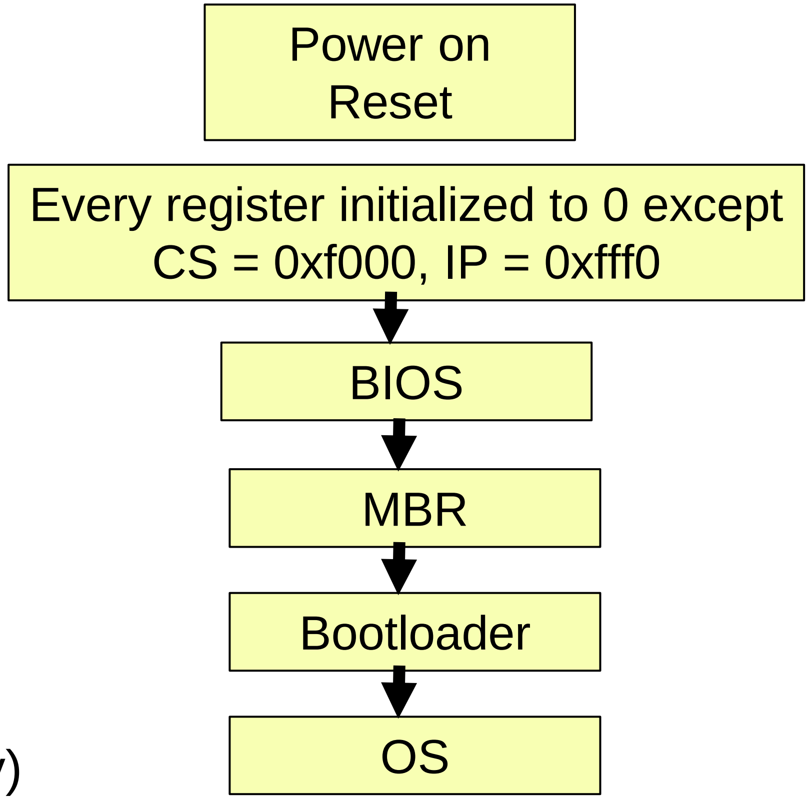
Grand Unified Bootloader (GRUB)

- 2^{nd} stage bootloader
- Allow the user to select which OS to load
- Can read many filesystem formats
- Load kernel image and the configuration
- Can load kernel images over the network



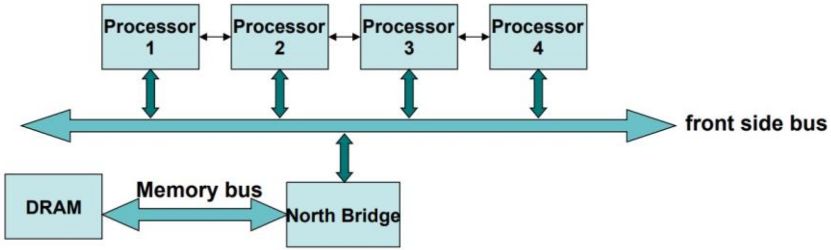
Powering up: OS

- **The operating system**
  - Set up virtual memory
  - Initialize interrupt vectors
  - Initialize
    - Timers
    - Monitors
    - Hard disks
    - Consoles
    - File systems
  - Initialized other processor (if any)
  - Startup user process



Multiprocessor booting

- One processor designated as **“Boot Processor” (BSP)**
  - Designation done either by hardware or BIOS
  - All other processors are designated **AP (Application Processors)**
- BIOS boots the BSP
- BSP learns system configuration
- BSP triggers boot of other AP
  - Done by sending an startup IPI (inter processor interrupt) signal to the AP



http://www.cse.iitm.ac.in/~chester/courses/16o_os/slides/3_Hardware.pdf

Takeaway Questions

- Where could we find BIOS?
  - (A) Hard drive
  - (B) CPU
  - (C) ROM
- How does BIOS find its entry point?
  - (A) Interrupt
  - (B) Reset vector
  - (C) Using system call

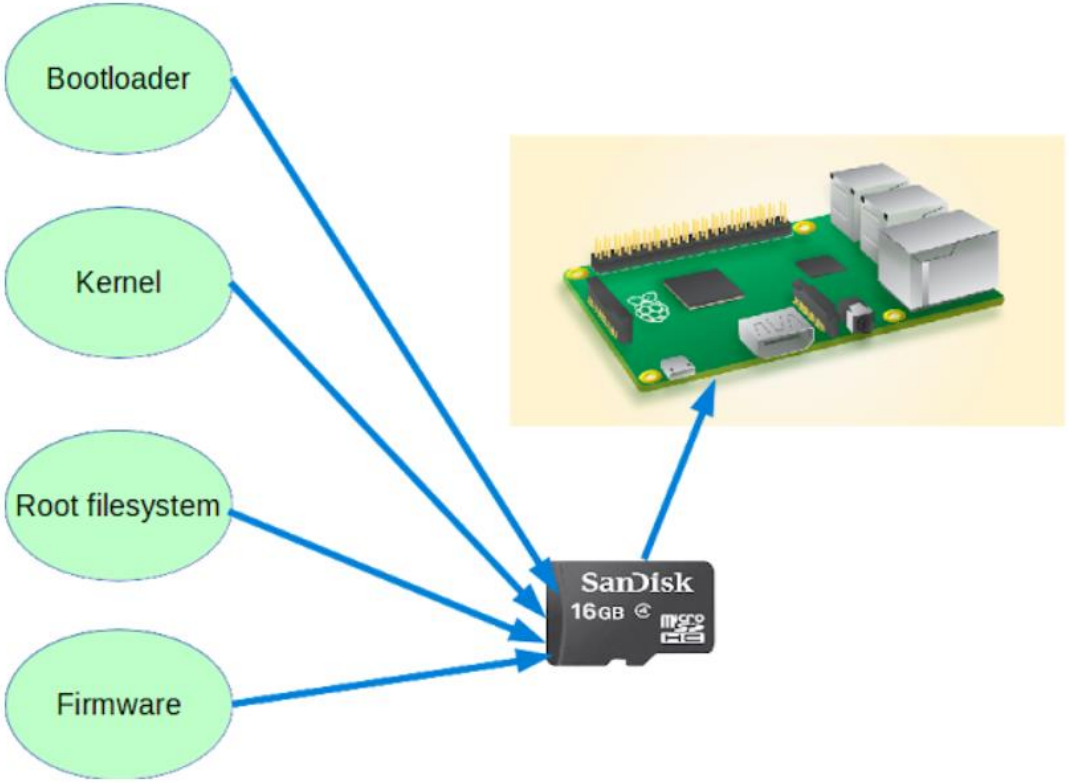
Takeaway Questions

- How does bootloader load the OS kernel?
  - (A) System call
  - (B) BIOS interrupt
  - (C) Reset vector
- Who take charge of the transition of real mode to protect mode?
  - (A) Bootloader
  - (B) BIOS
  - (C) OS kernel

Outline

- Booting on the X86 Processor
  - BIOS
  - MBR
  - Bootloader
- Booting on rpi
- Linux Bootstraping

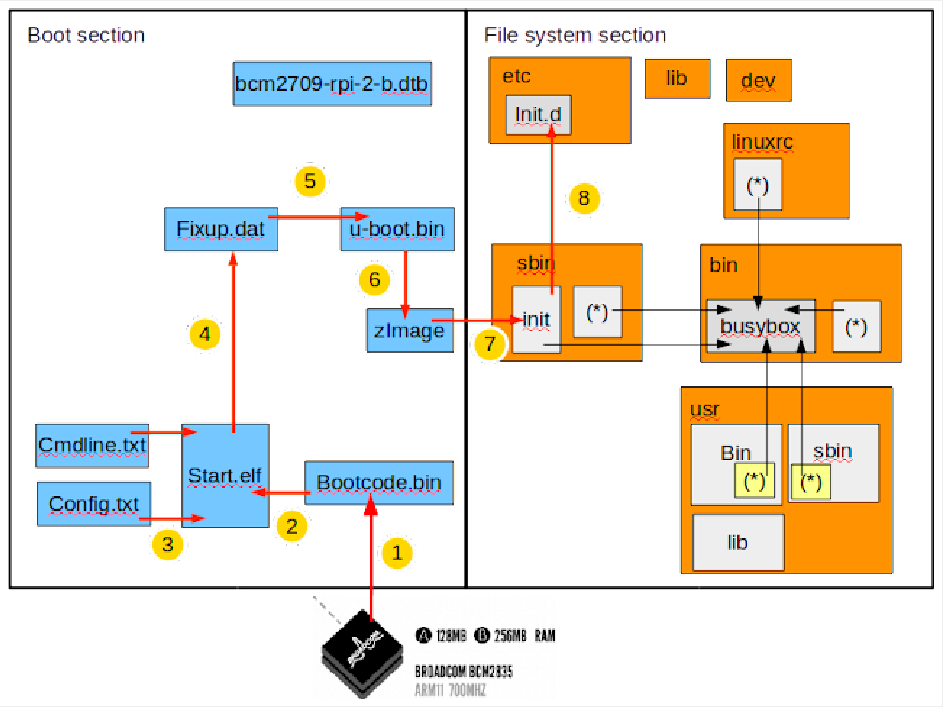
A Entire Linux System Includes



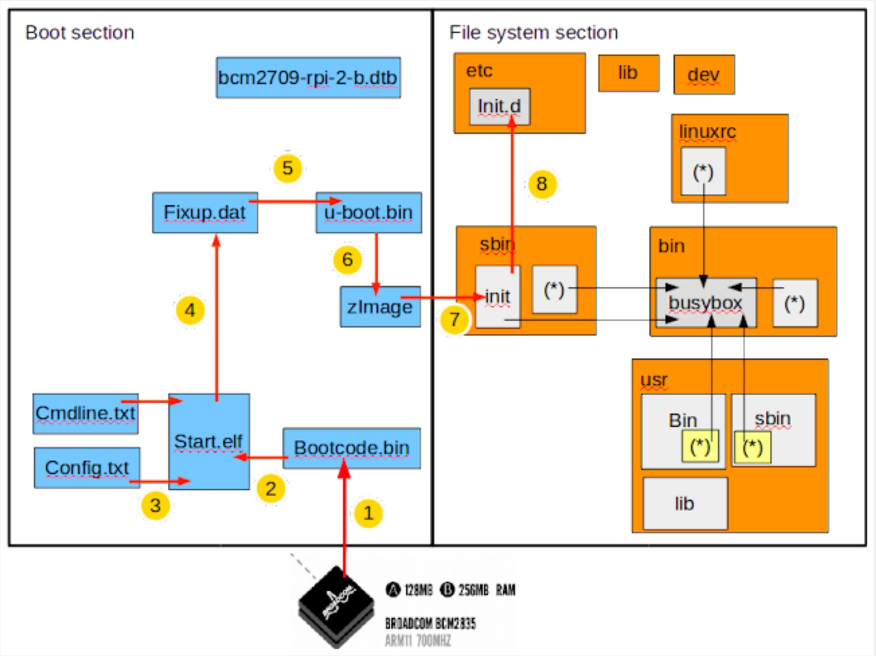
National Yang Ming Chiao Tung University
Computer Architecture & System Lab

Booting on rpi board

- When power on
  - First-stage bootloader on ROM
    - CPU/RAM are not initialized
    - GPU handles this first stage bootloader
  - Second-stage bootloader
    - Mount bootcode.bin on FAT32 of SD card
    - GPU places bootcode.bin on its L2 cache, activates RAM and read start.elf



Booting on rpi board



- GPU firmware
  - Start.elf (third-stage bootloader)
    - VideoCore OS
    - Read config.txt that represents BIOS setting
    - GPU and CPU RAM use different memory region
  - Run fixup.dat
    - Organize SDRAM partition between GPU/CPU
    - Reset CPU
  - Read zImage to RAM and kernel takes over -> /sbin/init -> login shell

Boot sequence of Raspberry Pi

- Boot from the GPU
- **Stage 1:**
  - GPU activates bootstrap code in the ROM to check filesystem on SD card
- **Stage 2:**
  - GPU loads bootcode.bin in /boot from the SD card to L2 cache (first-stage bootloader)
- **Stage 3:**
  - Bootcode.bin activates SDRAM and loads loader.bin to RAM and executes loader.bin
- **Stage 4:**
  - Loader.bin (second-stage bootloader) loads start.elf that is the firmware of the GPU

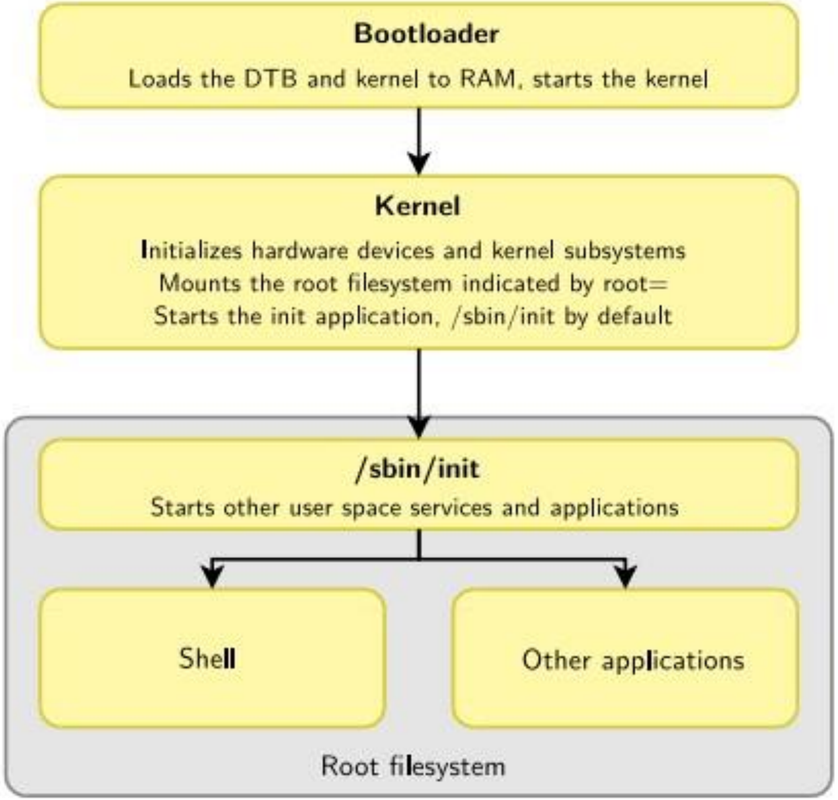
Boot sequence of Raspberry Pi

- **Stage 5:**
  - Start.elf reads config.txt and cmdline.txt and loads kernel.img that is Linux kernel
- **Stage 6:**
  - Activating the CPU after the start.elf loads kernel.img

Outline

- Booting on the X86 Processor
  - BIOS
  - MBR
  - Bootloader
- Booting on rpi
- Linux Bootstraping

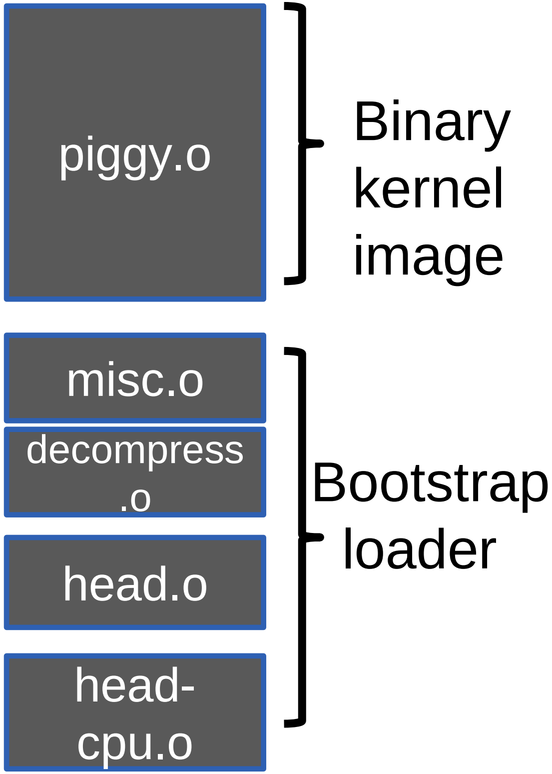
Overall Linux boot sequence



https://bootlin.com/doc/training/buildroot/buildroot-slides.pdf

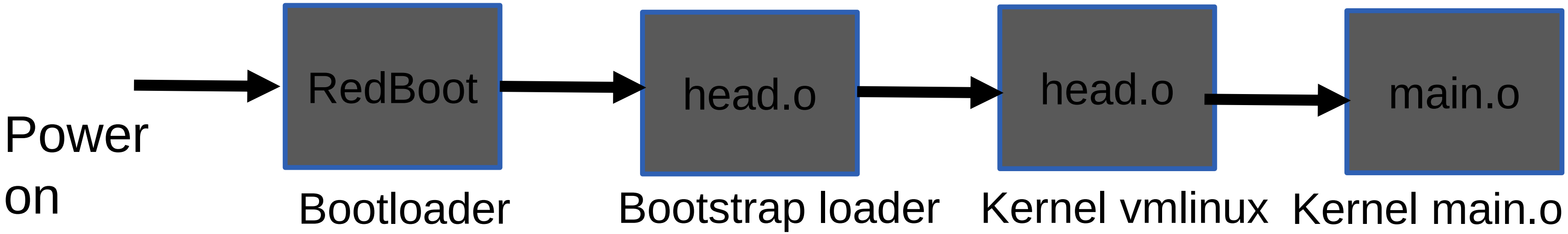
Bootstrap Loader

- **The second-stage loader (bootstrap loader)**
  - Load the Linux kernel image into memory
  - Act as the glue between a board-level bootloader and the Linux kernel
  - Low-level assembly processor initialization
  - Decompression and relocation of the kernel image
- **The first-stage loader**
  - Controls the board upon power-up
  - Does not reply on the Linux kernel in any way

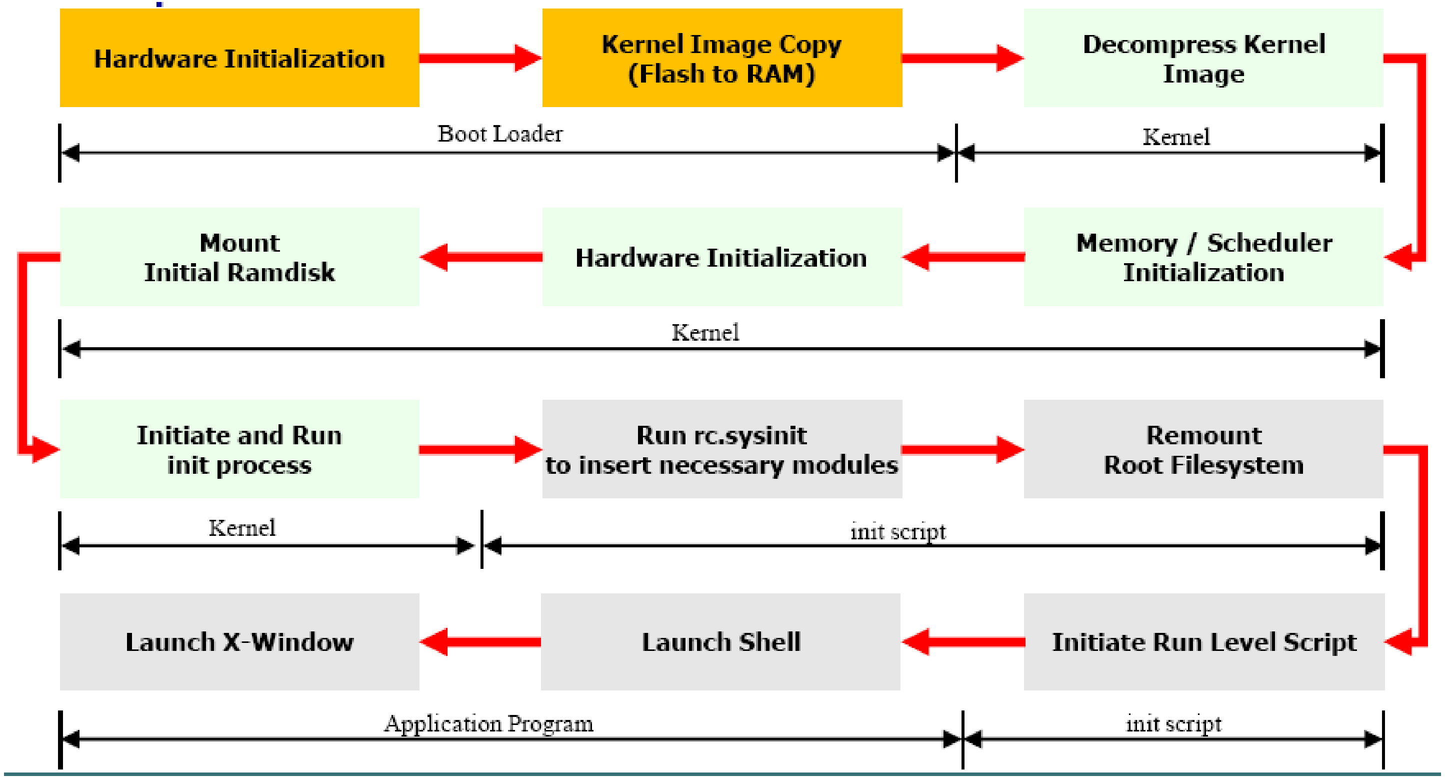


Kernel entry point: head.o

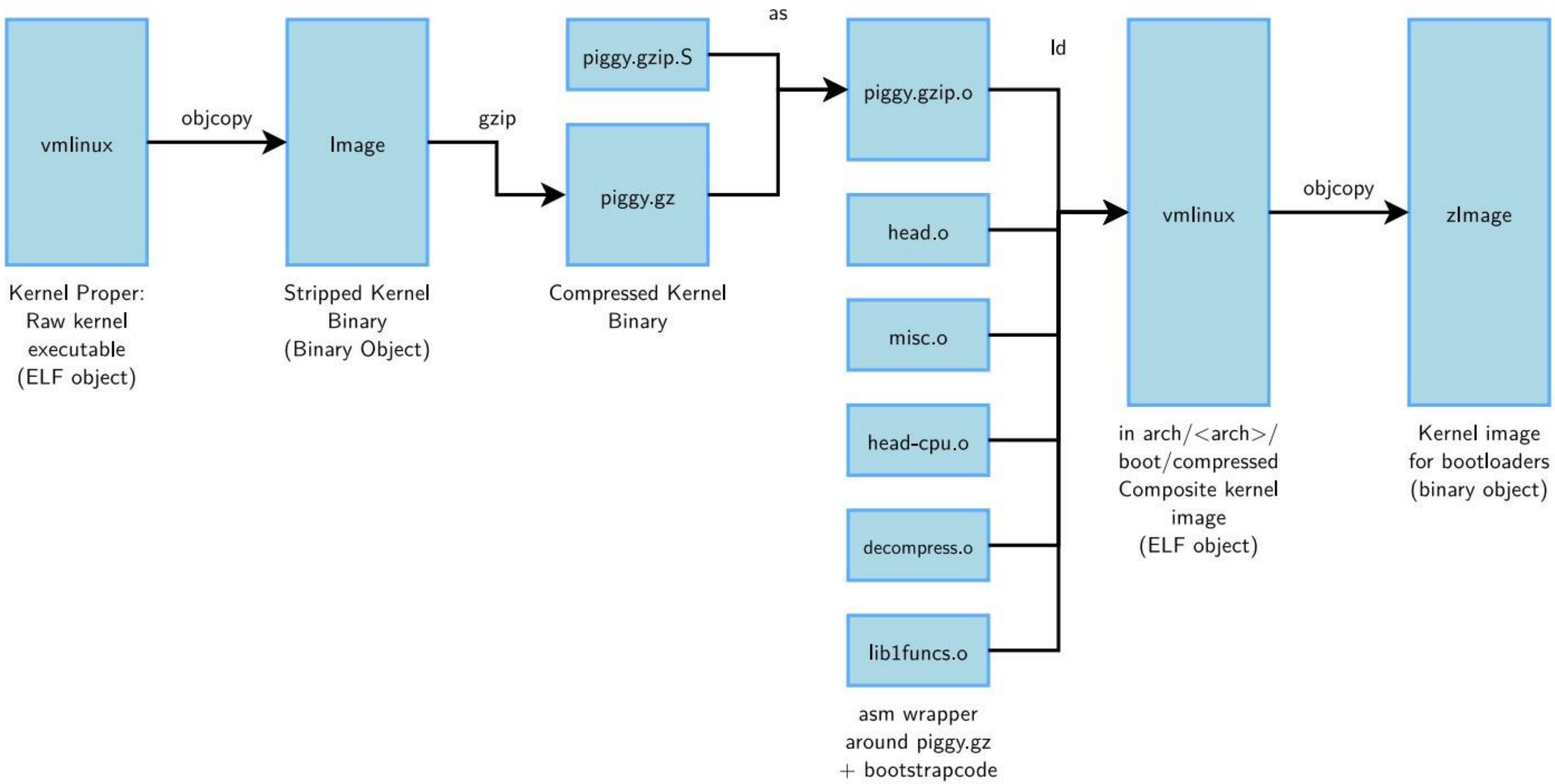
- The un-compression code jumps into the main kernel entry point
  - Located in arch/<arch>/kernel/head.S
  - Check the architecture, processor and machine type
  - Configure the MMU, create page table entries and enable virtual memory
  - Same code for all architectures
  - Calls the start_kernel function in init/main.c



Loading kernel



Kernel bootstrap



https://bootlin.com/doc/legacy/kernel-init/kernel-init.pdf

Bootstrap code for compressed kernels

- **vmlinux.lds**
  - Kernel proper, in ELF format, including symbols, comments, debug info
- **System.map**
  - Text-based kernel symbol table for vmlinux module
- **Image**
  - Binary kernel module, stripped of symbols, notes and comments
  - objcopy –O binary –R .note –R .comment –S vmlinux.lds arch/arm/boot/Image
- **head.o**
  - Architecture-specific startup code
  - Passed control by the bootloader

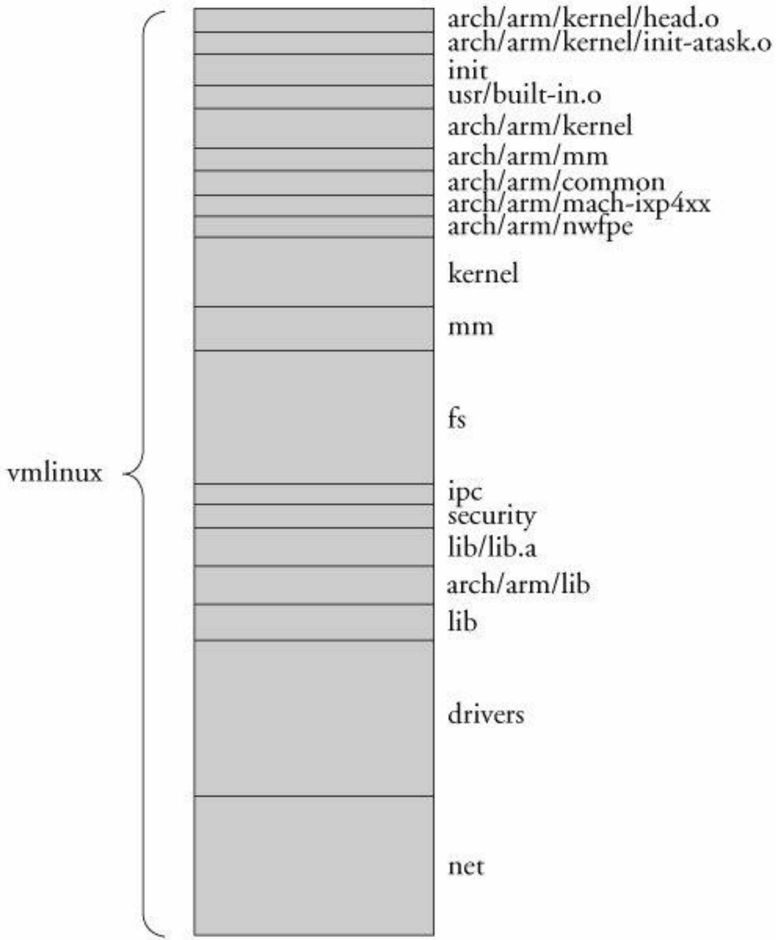
Located in arch/<arch>/boot/compressed

Bootstrap code for compressed kernels

- **piggy.gz**
  - The file image compressed with gzip (gzip –f -9 < Image > piggy.gz)
- **piggy.o**
  - The file piggy.gz in assembly language format from piggy.S
  - It can be linked with a subsequent object, misc.o
- **misc.o, decompress.o**
  - Routines used for decompressing the kernel image (piggy.gz)
- **vmlinux**
  - Composite kernel image and is the result when the kernel proper is linked with the objects
- **zImage**
  - Final composite kernel image loaded by bootloader

vmlinux

- head.o
  - Kernel architecture-specific startup code
- arch/arm/kernel/init-task.o
  - Initial thread and task structs required by kernel
- init
  - Main kernel-initialization code
- usr/built-in.o
  - Built-in initramfs image
- arch/arm/nwfpe
  - Architecture-specific floating point – emulation code



Using the initial RAM disk (initrd)

- After Linux kernel finds the ”init” and aims:
  - Used to prepare the work before mounting the root file system
  - The init process is in the root file system
  - However, the root filesystem can be mounted in SATA/SCSI storage devices
  - Don’t want to load so many drivers to the kernels
  - The boot loader informs the kernel that an initrd exists and where it is located in memory

rest_init: Starting the init process

```
static noinline void __init_refok rest_init(void)
        __releases(kernel_lock)
{
        int pid;

        rcu_scheduler_starting();
        /*
         * We need to spawn init first so that it obtains pid 1, however
         * the init task will end up wanting to create kthreads, which, if
         * we schedule it before we create kthreadd, will OOPS.
         */
        kernel_thread(kernel_init, NULL, CLONE_FS | CLONE_SIGHAND);
        numa_default_policy();
        pid = kernel_thread(kthreadd, NULL, CLONE_FS | CLONE_FILES);
        rcu_read_lock();
        kthreadd_task = find_task_by_pid_ns(pid, &init_pid_ns);
        rcu_read_unlock();
        complete(&kthreadd_done);

        /*
         * The boot idle thread must execute schedule()
         * at least once to get things moving:
         */
        init_idle_bootup_task(current);
        preempt_enable_no_resched();
        schedule();
        preempt_disable();

        /* Call into cpu_idle with preempt disabled */
        cpu_idle();
```

Root file system

- **Initial RAM Disk (initrd)**
  - A small self-contained root file system
  - Contains directives to load specific device drivers before the completion of the boot cycle
  - When the kernel boots, it copies the compressed binary file from the specified physical location in RAM into a proper kernel ramdisk and mount it as the root file system
  - Use **linuxrc** file to execute commands
- **The root file system**
  - Refer to the file system mounted at the base of the file system hierarchy, designated simply as /
  - Contains programs and utilities to boot a system and initialize services

```
/
/bin
/dev
/etc
/lib
/sbin
/usr
/var
/tmp
```

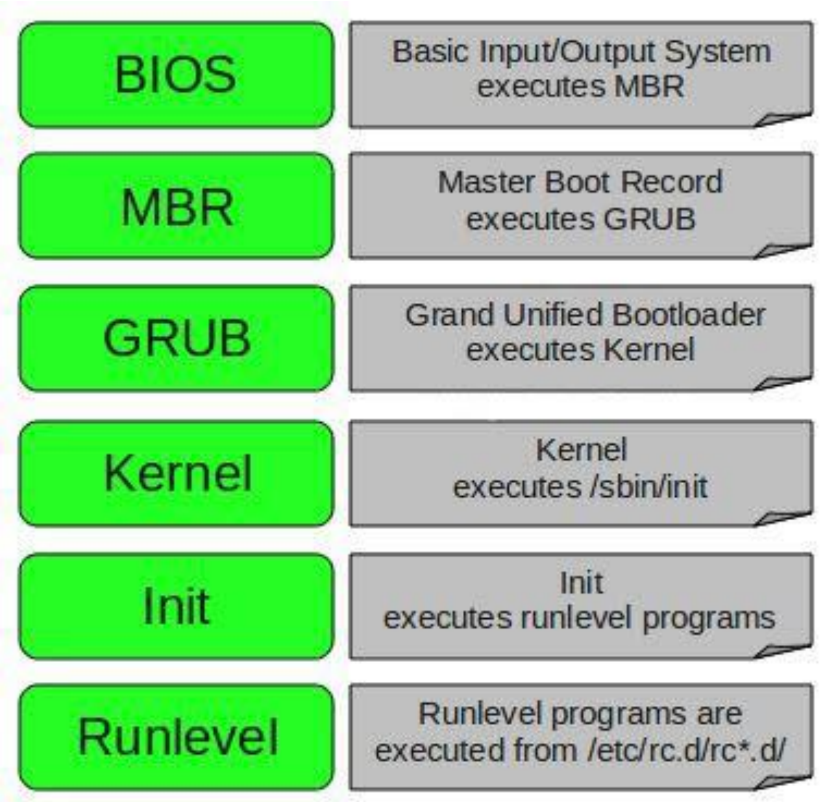
Final stage of the boot

- After kernel thread calls init during the final stages of boot
  - run_init_process()
  - /sbin/init is spawned by the kernel on boot
    - Mount the root file system
    - Spawn the first user space program, init
- **inittab**
  - When init is started, it reads the system configuration file /etc/inittab
  - Contains directive for each runlevel
  - e.g. runlevel 0 instructs init to halt the system
  - Runlevel directories are typically rooted at /etc/rc.d

Summary

- OS booting

BIOS -> MBR -> boot loader -> kernel -> init process -> login



https://www.thegeekstuff.com/2011/02/linux-boot-process

Takeaway Questions

- Where is the entry point of the Linux Kernel?
  - (A) BIOS
  - (B) head.S
  - (C) MBR
- What is the name of the first process after booting?
  - (A) init
  - (B) root
  - (C) initrd