Operating System Design and Implementation

Lecture 4: BIOS & Bootloader

Tsung Tai Yeh

Tuesday: 3:30 – 5:20 pm
Classroom: ED-302

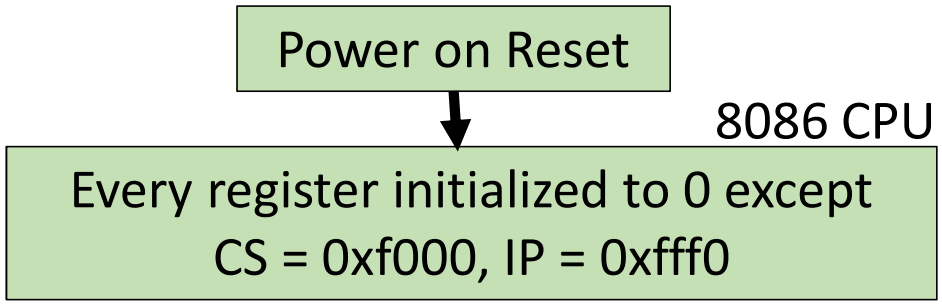
Acknowledgements and Disclaimer

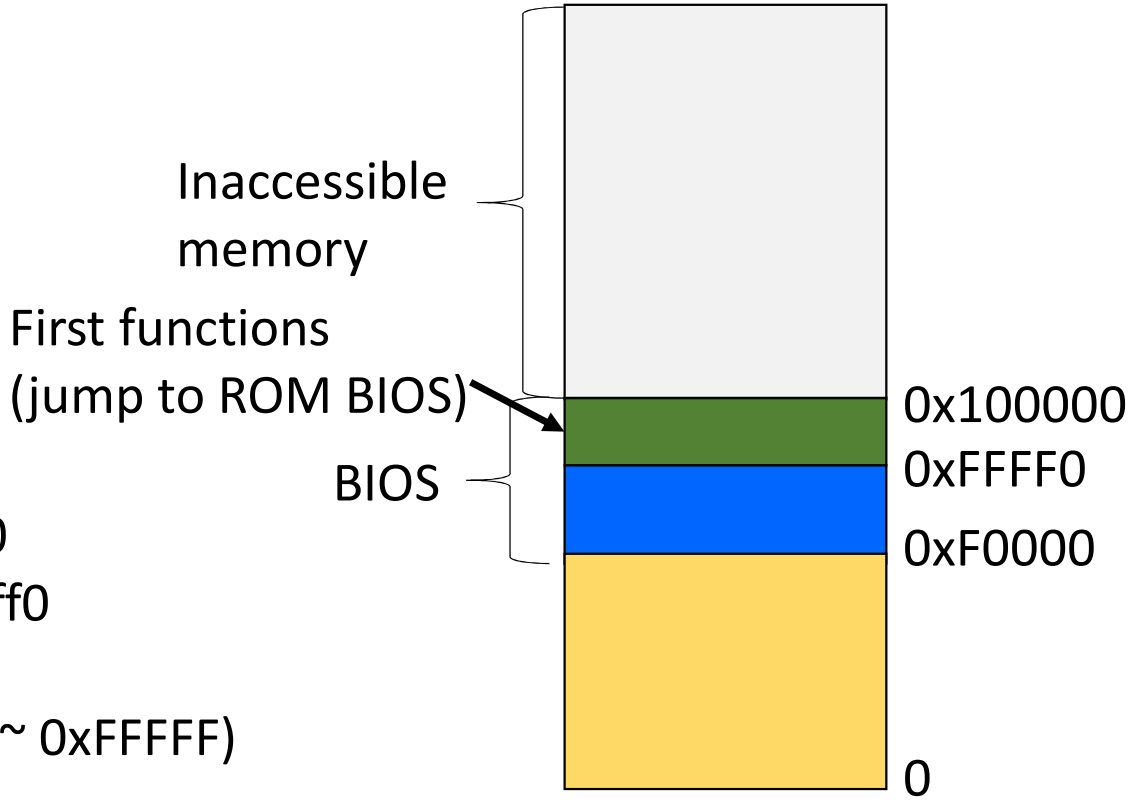
- Slides was developed in the reference with
  MIT 6.828 Operating system engineering class, 2018
  MIT 6.004 Operating system, 2018
  Remzi H. Arpaci-Dusseau etl. , Operating systems: Three easy pieces. WISC

Outline

- Boot sequence
  - Bootloader on x86
  - Bootloader on embedded system
- Linux kernel initialization
  - Kernel bootstrap
  - Compressed kernels
  - The root file system

Powering up: Reset



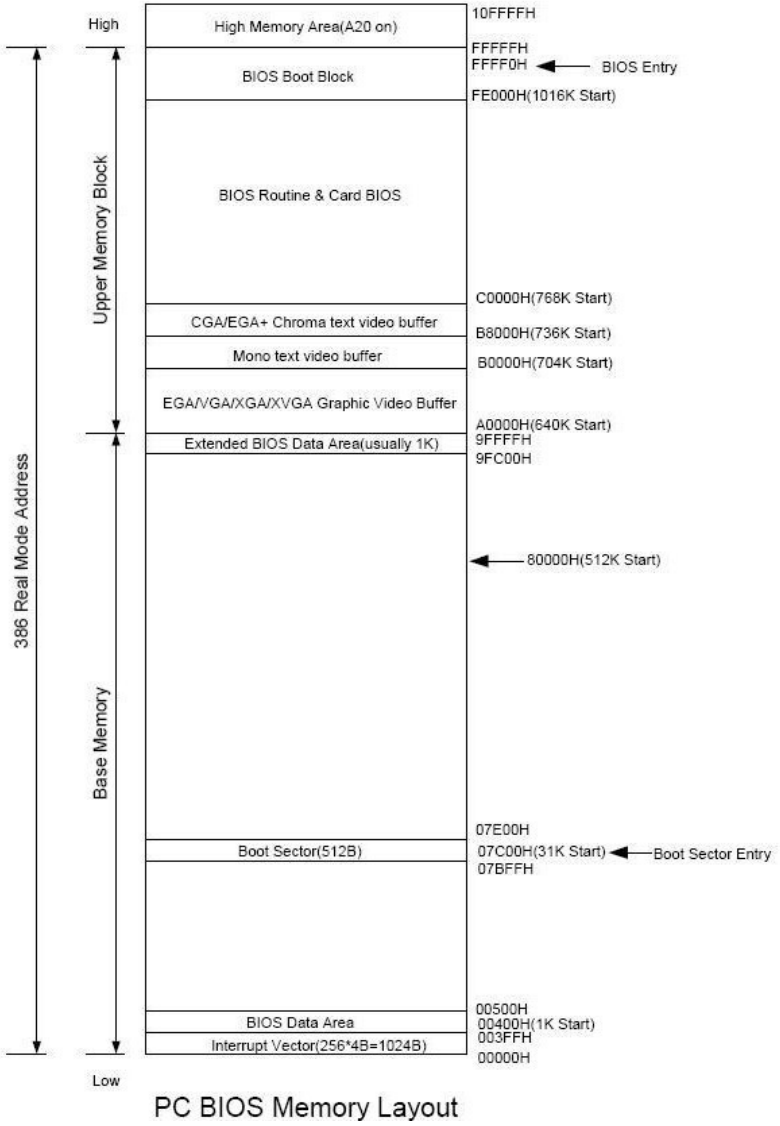


1. Physical address = (CS << 4) + IP = 0xffff0
2. First instruction fetch from location 0xffff0
3. Processor in real mode
   a. Limited to 1MB addresses (0x00000 ~ 0xFFFFF)
   b. No protection; no privilege levels
   c. Direct access to all memory
   d. No multi-tasking
4. First instruction is on the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

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

Powering up: BIOS

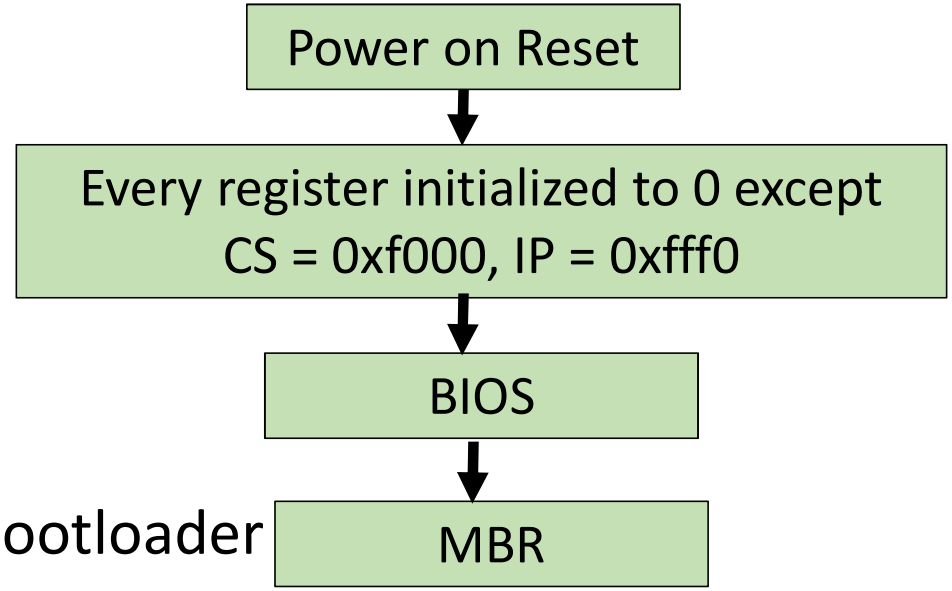
- BIOS presents in a small chip connected to processor
  - Flash/EPROM/EEPROM
- **BIOS work**
  - Power on self test
  - Initialize video card and other devices
  - Display BIOS screen
  - Perform brief memory test
  - Set DRAM memory parameters
  - Configure plug & play devices
  - Assign DMA channels and IRQs
  - Identify the boot device
    - Read sector 0 from boot device into memory location 0x7c00
    - Jumps to 0x7c00



PC BIOS Memory Layout

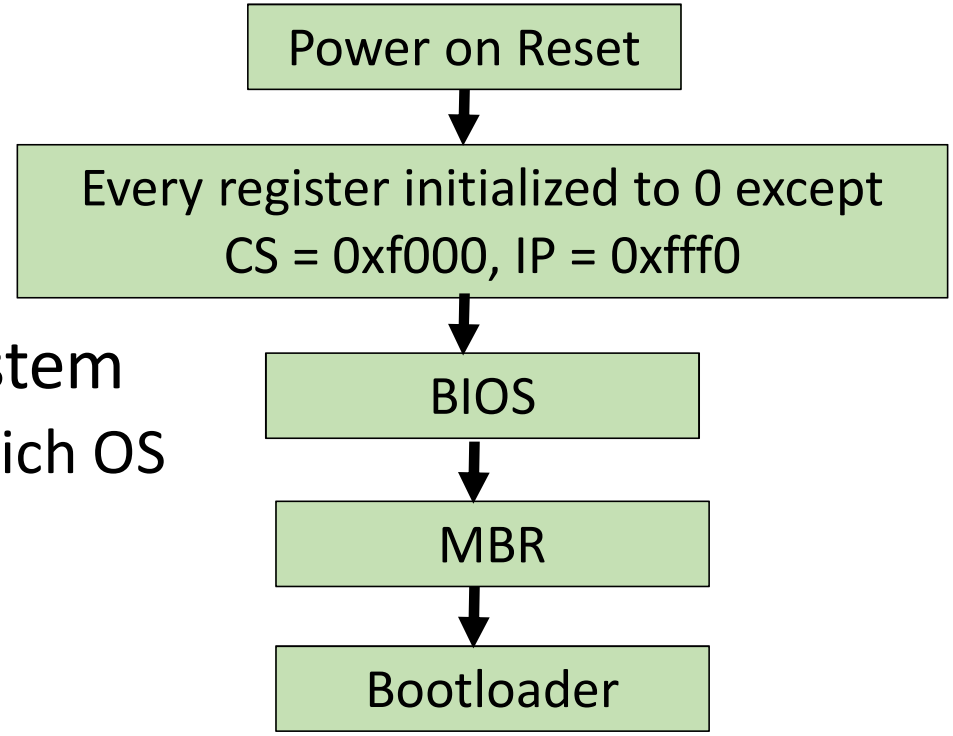
https://www.twblogs.net/a/5b7f367e2b717767c6ae576c

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

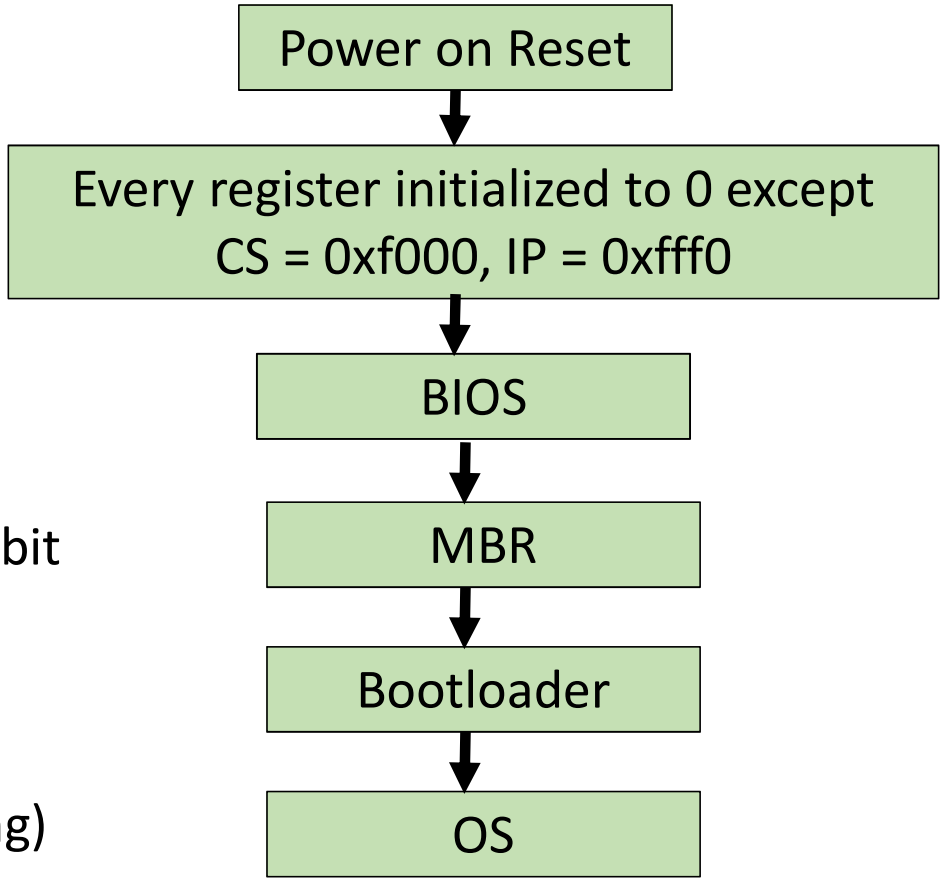
Powering up: bootloader



- Bootloader loads the operating system
  - May also allow the user to select which OS to load
- Other jobs done
  - Disable interrupts
  - Setup GDT (global descriptor table)
  - Switch from real mode to protected mode
  - Read operating system from disk
  - The bootloader may be presented in the MBR (sector 0)

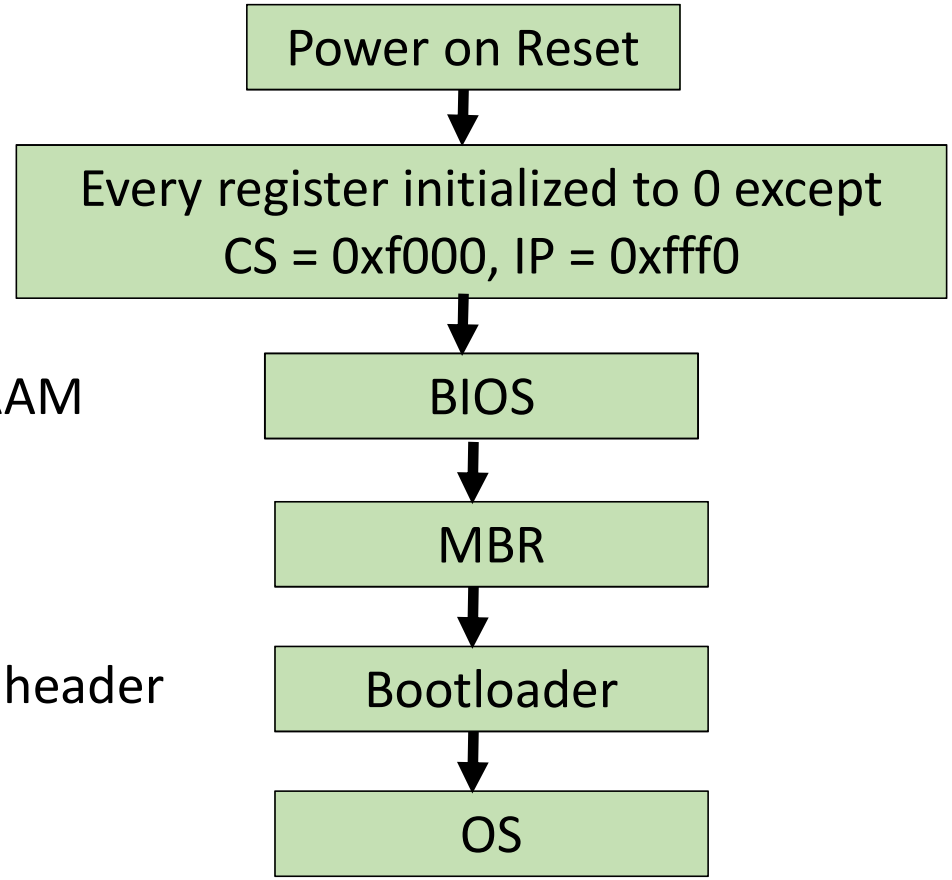
Powering up: xv6

- **Bootloader**
  - Present in sector 0 of disk
  - 512 bytes
  - bootasm.S
    - Enters in 16 bit real mode, leaves in 32 bit protected mode
    - Disable interrupts
    - Enable A20 physical address line
    - Load GDT (only segmentation, no paging)
    - Set stack to 0x7c00
    - Never returns



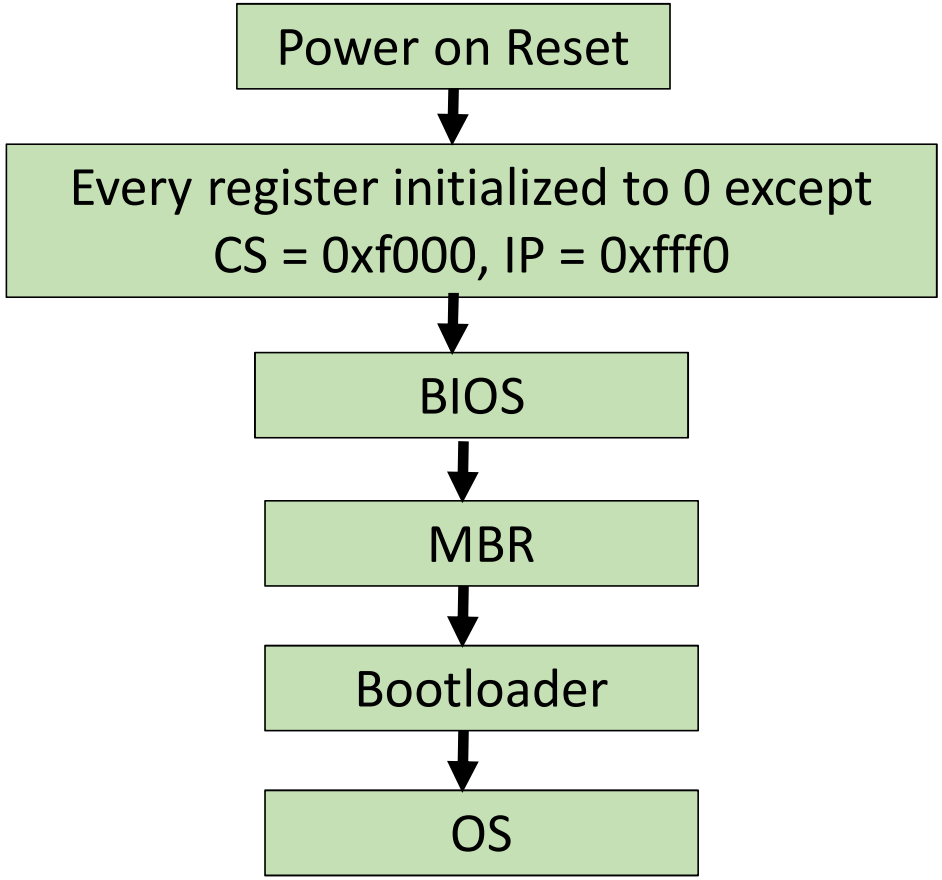
Powering up: xv6 (cont.)

- **Bootloader**
  - bootmain.c
    - Loads the xv6 kernel from sector 1 to RAM
    - Starting at 0x10000 (1MB)
    - Invoke the xv6 kernel entry
    - _start present in entry.S
    - This entry point is known from the ELF header



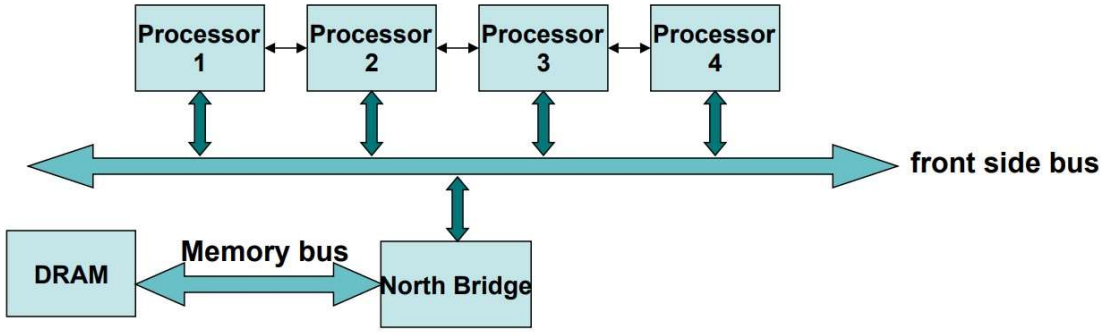
Powering up: OS

- **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

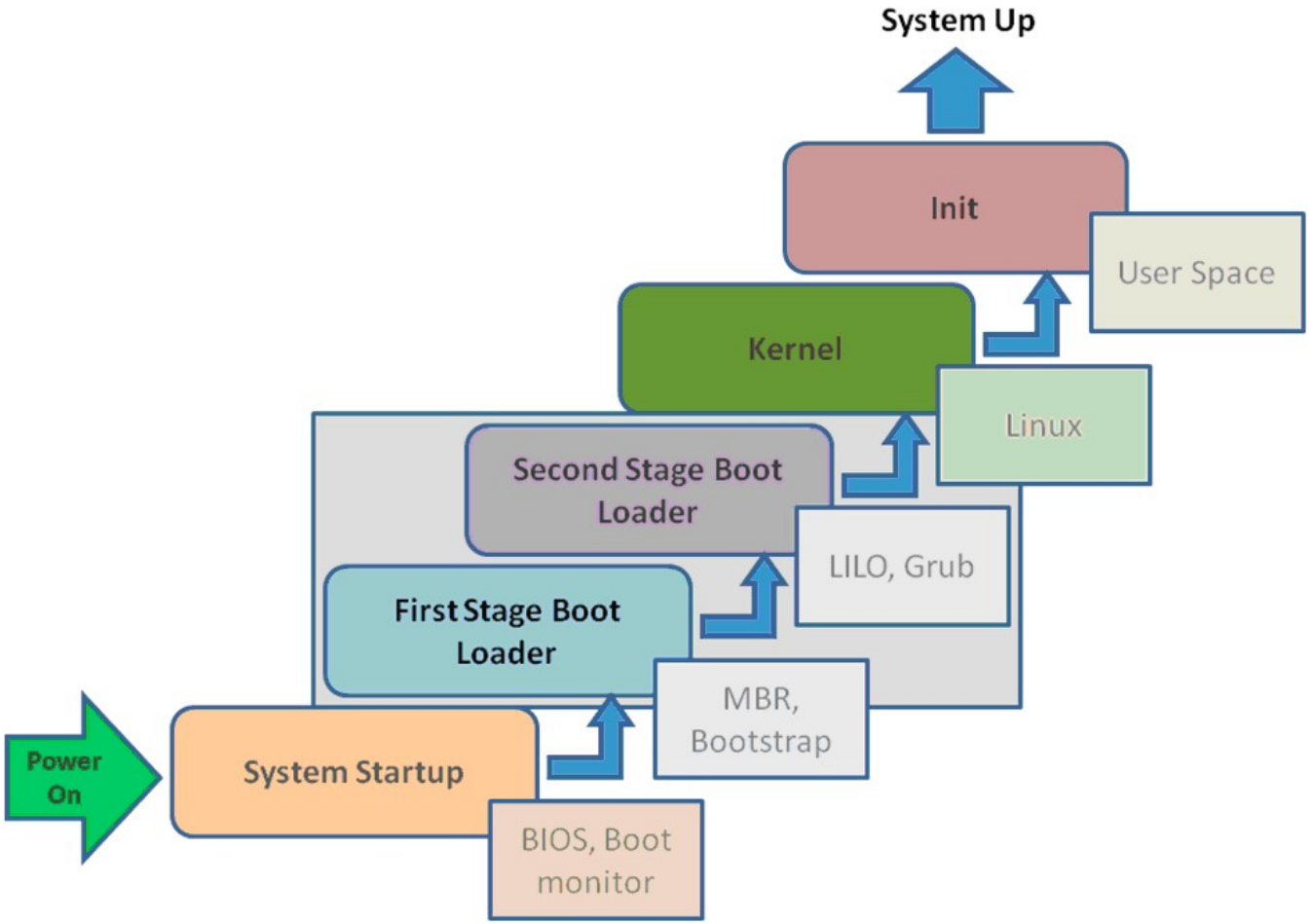
Boot sequence of Raspberry Pi

- 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
- **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

Bootloaders

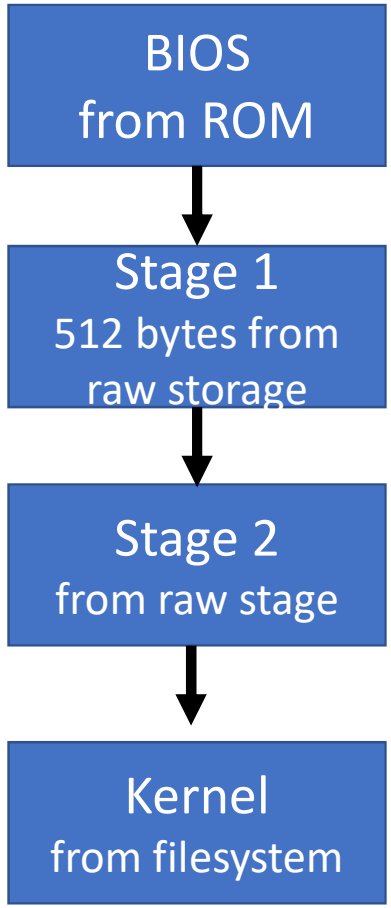
- The bootloader is a piece of code that is responsible for
  - Basic hardware initialization
  - Loading an operating system kernel from non-volatile storage
  - Possibly decompression of the application binary
  - Execution of the application
- Most bootloaders provides a shell with various commands
  - Loading of data from storage or network
  - Memory inspection
  - Hardware diagnostics and testing

Bootloader on x86 processor



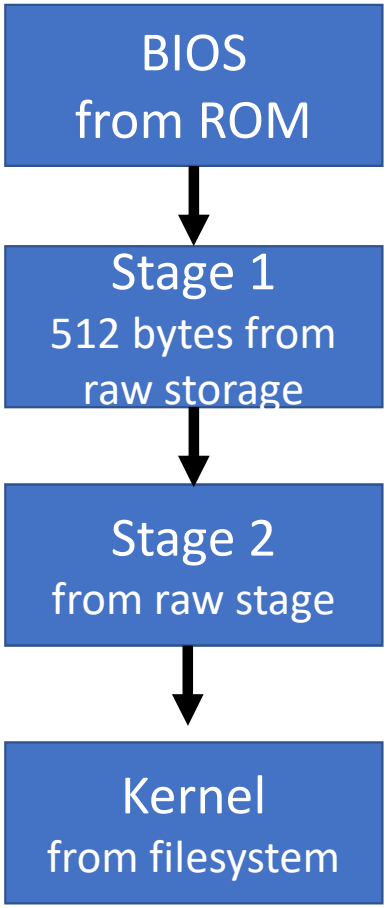
Bootloaders on BIOS-based x86

- **Basic Input Output System (BIOS)**
  - a program
  - bundled on a board with non-volatile memory on x86 processor
- On old BIOS-based x86 platform
  - Responsible for **basic hardware initialization**
  - **Loading small piece code** from non-volatile storage
  - This piece of code is typically a 1st stage bootloader which will load the full bootloader itself
  - It typically understands filesystem format so that kernel file can be loaded directly from a normal filesystem



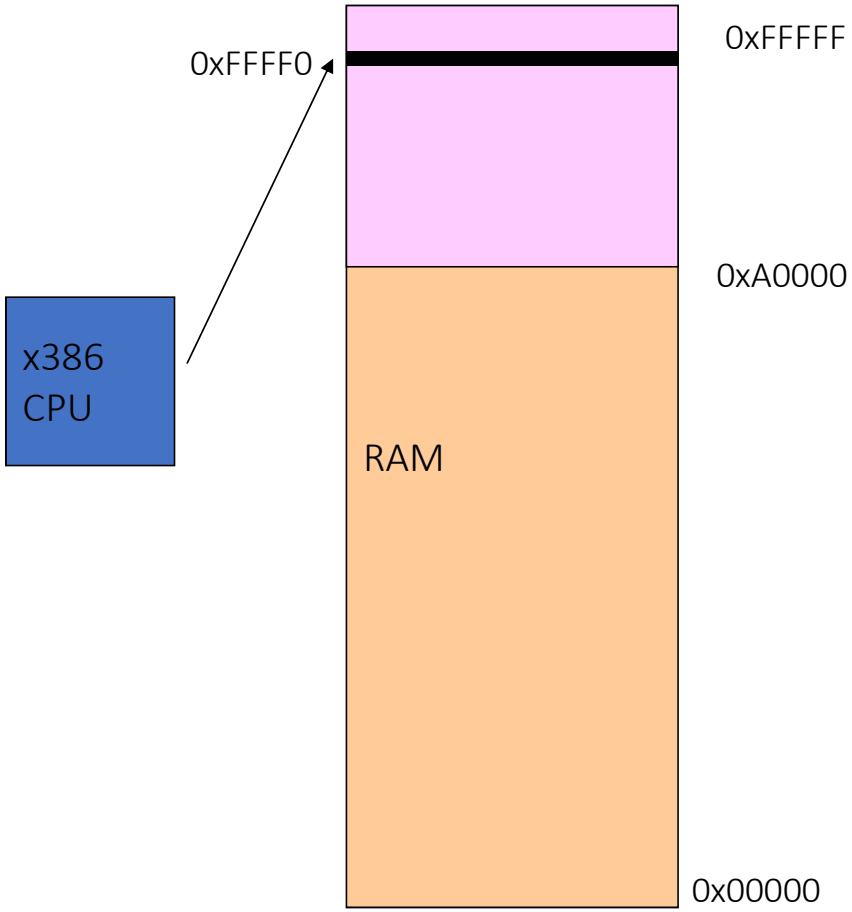
Bootloaders on x86

- **Grand Unified Bootloader (GRUB)**
  - 2nd stage bootloader
  - Can read many filesystem formats
  - Load kernel image and the configuration
  - Can load kernel images over the network
- Syslinux
  - for network and removable media booting (USB key, CD-ROM)

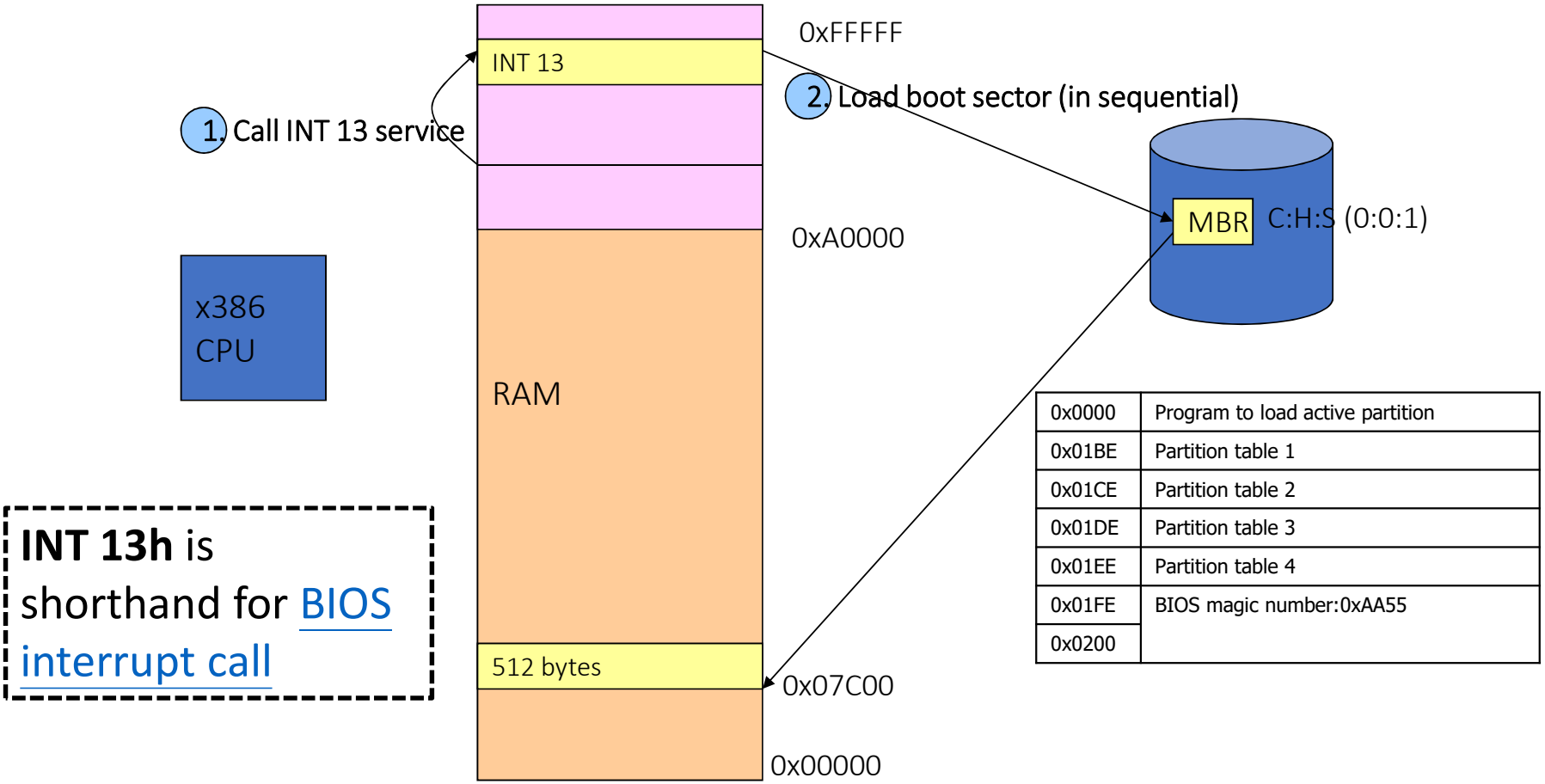


PC Booting (Cont)

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



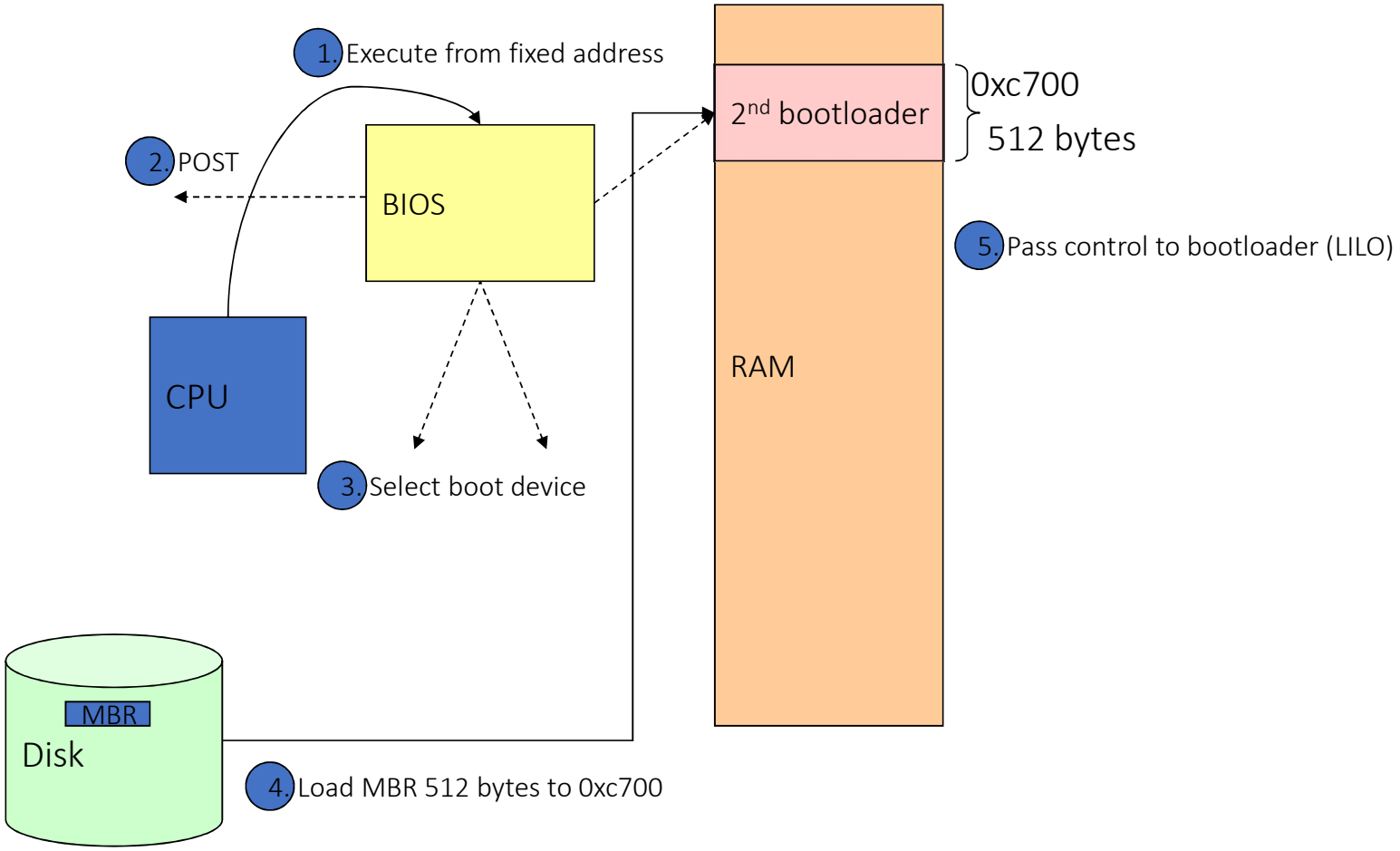
PC Booting (Cont)



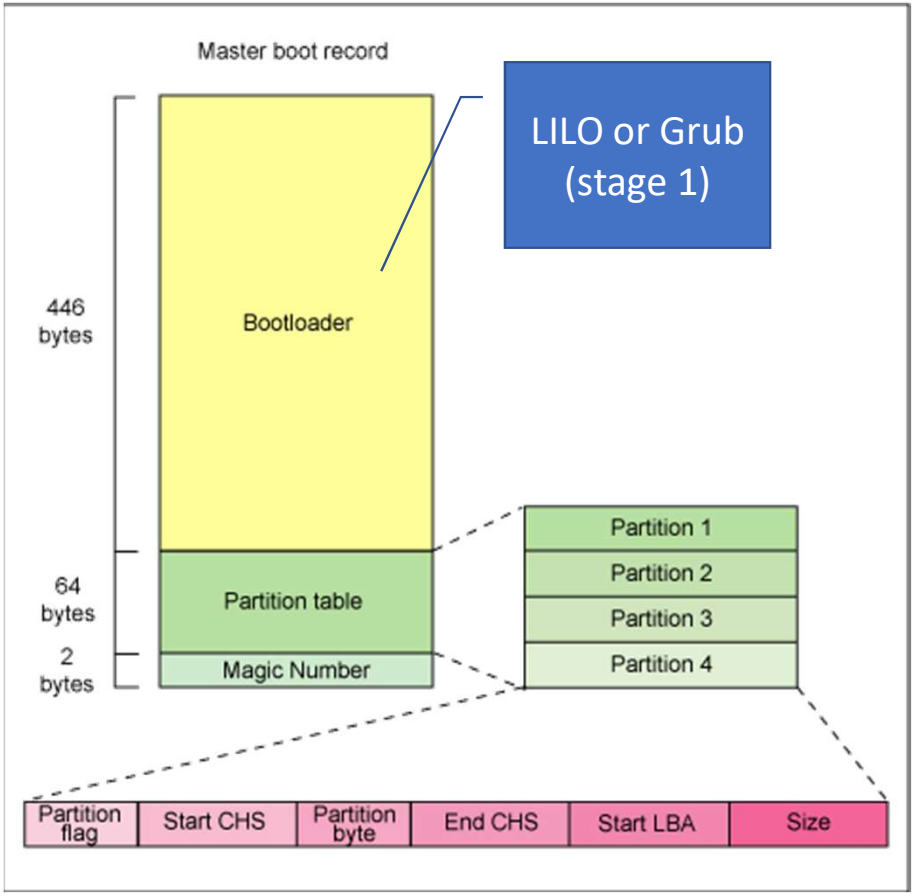
0x0000	Program to load active partition
0x01BE	Partition table 1
0x01CE	Partition table 2
0x01DE	Partition table 3
0x01EE	Partition table 4
0x01FE	BIOS magic number:0xAA55
0x0200	

INT 13h is shorthand for BIOS interrupt call

Linux Boot Example



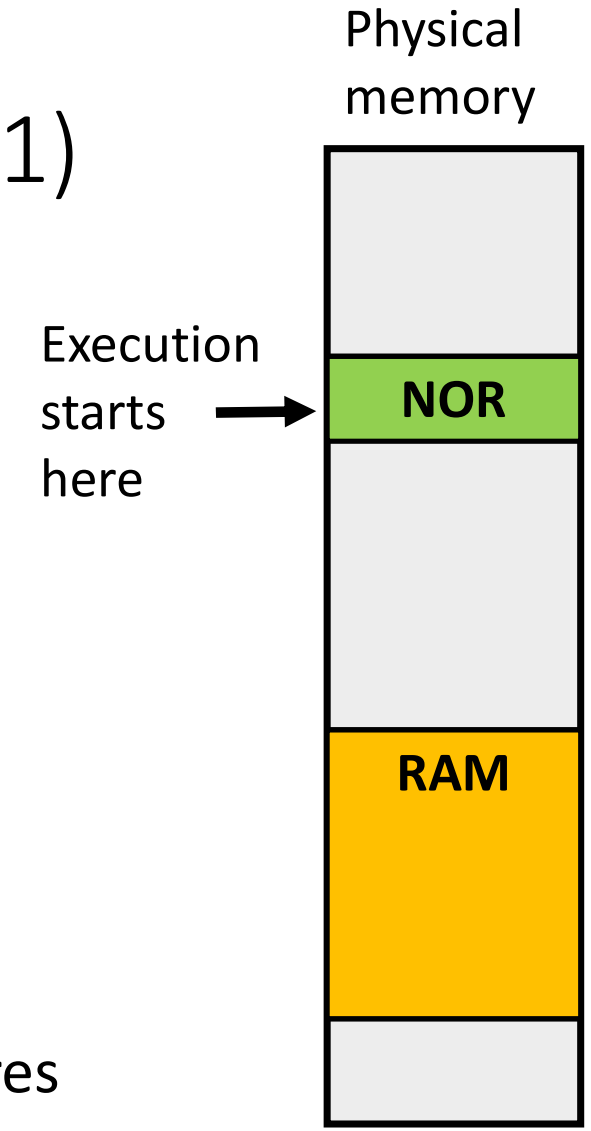
MBR (Master Boot Record)



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

Booting on embedded CPUs (case 1)

- When powered, that CPU starts executing code at a fixed address
- There is no other booting mechanism provided by the CPU
- The hardware design must ensure that a NOR flash chip is wired so that it is accessible at the address at which the CPU start executing instructions
- The first stage bootloader must be programmed at this address in the NOR
- **Not very common anymore** (unpractical, and requires NOR flash)

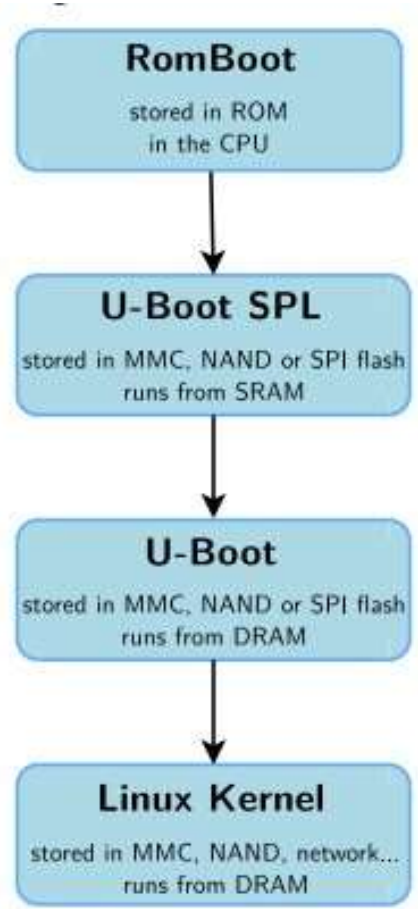


Booting on embedded CPUs (case 2)

- The CPU has an integrated boot code in ROM
  - BootROM on AT91 CPUs, “ROM code” on OMAP, etc.
- This boot code is able to load a first stage bootloader from a storage device into an internal SRAM (DRAM not initialized yet)
  - Storage device can typically be: MMC, NAND, SPI flash
- The first bootloader is
  - Limited in size due to hardware constraints (SRAM size)
  - Provided either by U-Boot or by the CPU vender
- This first bootloader must
  - Initialize DRAM and other hardware devices
  - Load a second stage bootloader into DRAM

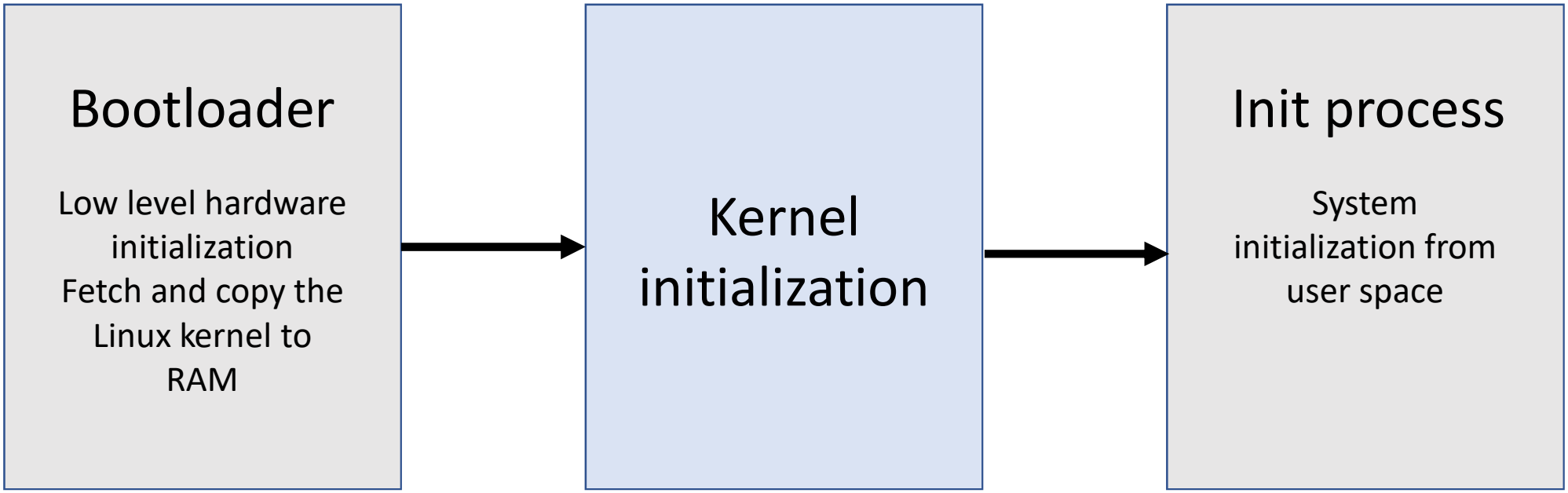
Booting on Microchip ARM SAMA5D3

- **RomBoot**
  - Tries to find a valid bootstrap image from various storage sources, and load it into SRAM
  - Size limited to 64KB. No user interaction possible in standard boot mode
- **U-Boot SPL**
  - **Run from SRAM**, initialize the DRAM, and NAND or SPI controller, and load the 2nd bootloader into DRAM and start it
  - No user interaction possible
- **U-Boot**
  - **Runs from DRAM**, initializes other hardware devices (network, USB, etc.), loads kernel image from storage or network to DRAM
- **Linux kernel**
  - **Runs from DRAM**, takes over the system completely, the boot loader no longer exists

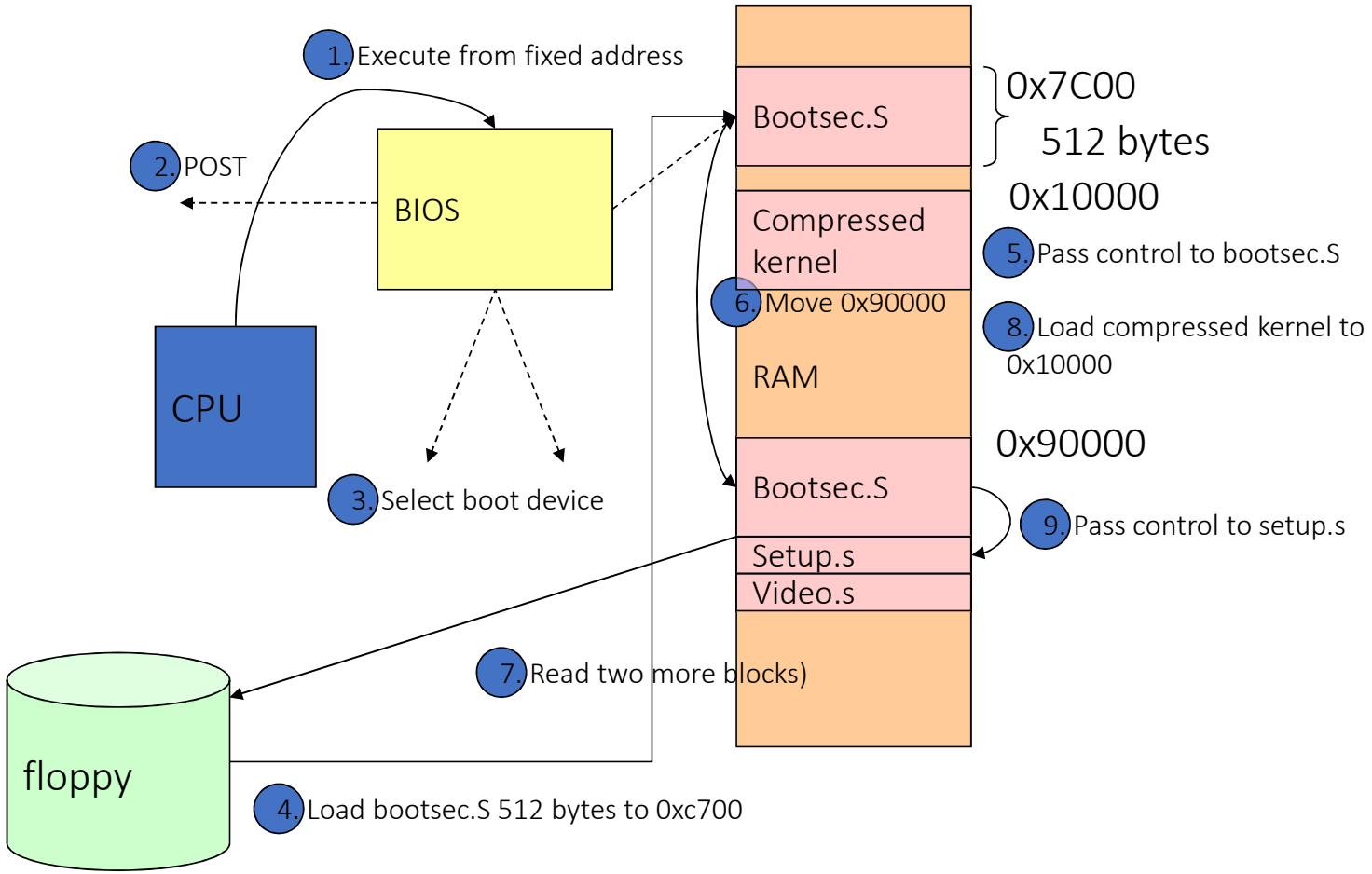


https://bootlin.com/doc/training/buildroot/buildroot-slides.pdf

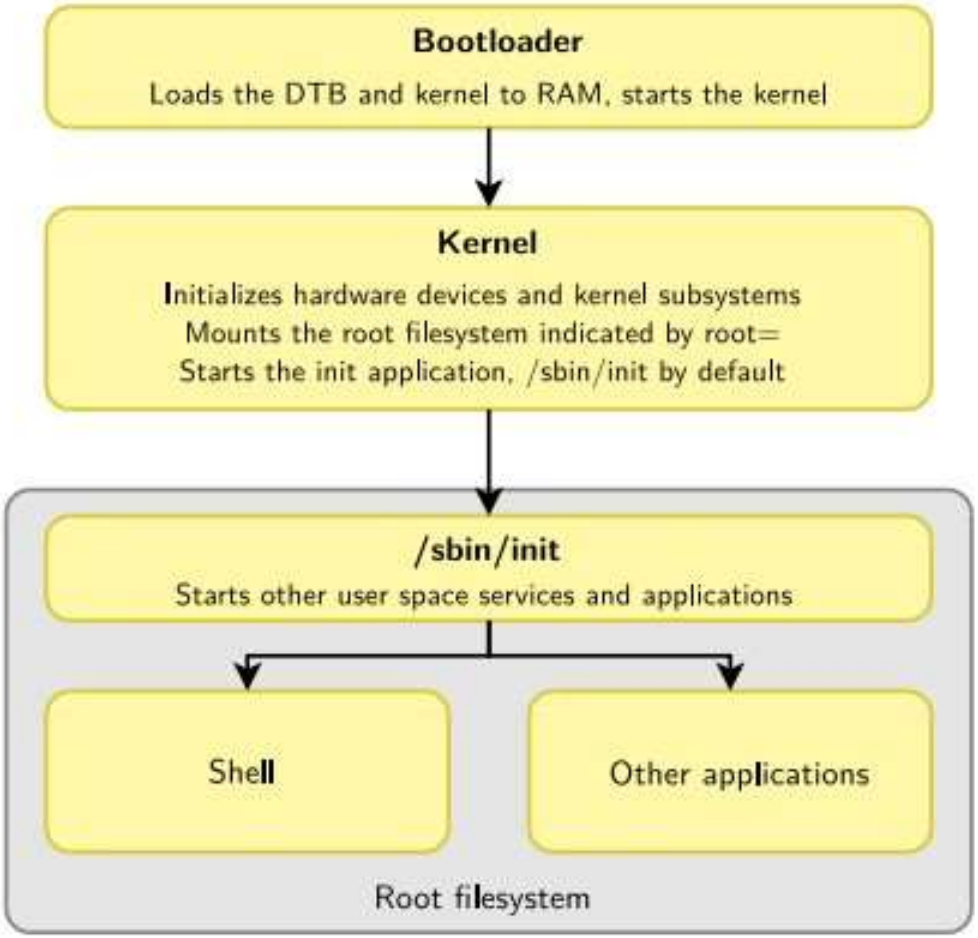
From Bootloader to user space



Linux Boot Example



Overall Linux boot sequence



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

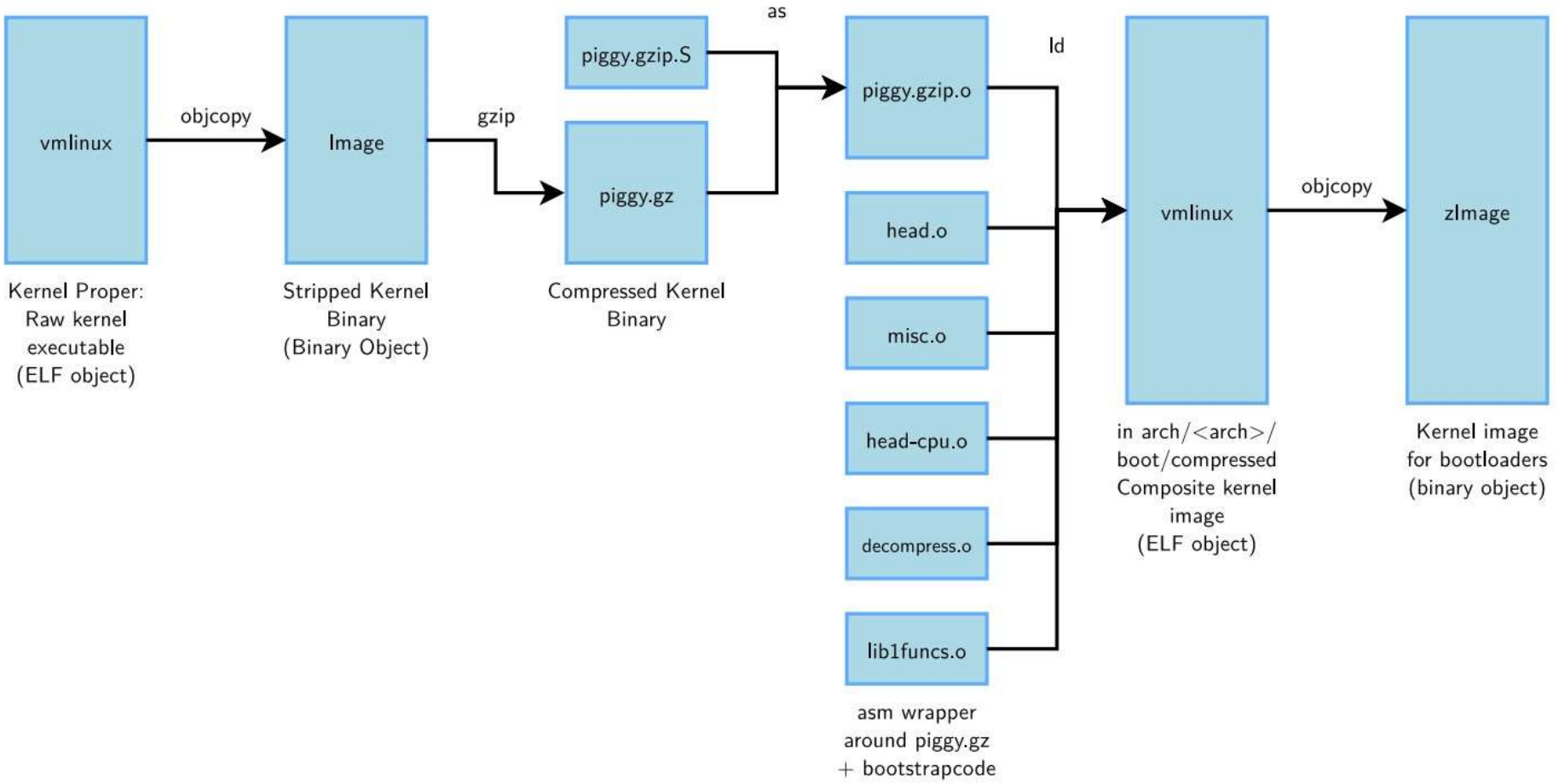
Kernel bootstrap (1)

- How the kernel bootstraps itself appears in kernel building

Raspberry pi Linux kernel
https://github.com/raspberrypi/linux

```
make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- zImage -j4
(....)
  LD      vmlinux
  SORTEX  vmlinux
  SYSMAP  System.map
  OBJCOPY arch/arm/boot/Image
  Kernel: arch/arm/boot/Image is ready
  Kernel: arch/arm/boot/Image is ready
  LDS     arch/arm/boot/compressed/vmlinux.lds
  AS      arch/arm/boot/compressed/head.o
  GZIP    arch/arm/boot/compressed/piggy.gzip
  CC      arch/arm/boot/compressed/misc.o
  CC      arch/arm/boot/compressed/decompress.o
  CC      arch/arm/boot/compressed/string.o
  AS      arch/arm/boot/compressed/lib1funcs.o
  AS      arch/arm/boot/compressed/ashldi3.o
  AS      arch/arm/boot/compressed/bswapsdi2.o
  AS      arch/arm/boot/compressed/piggy.gzip.o
  LD      arch/arm/boot/compressed/vmlinux
  OBJCOPY arch/arm/boot/zImage
  Kernel: arch/arm/boot/zImage is ready
```

Kernel bootstrap (2)



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

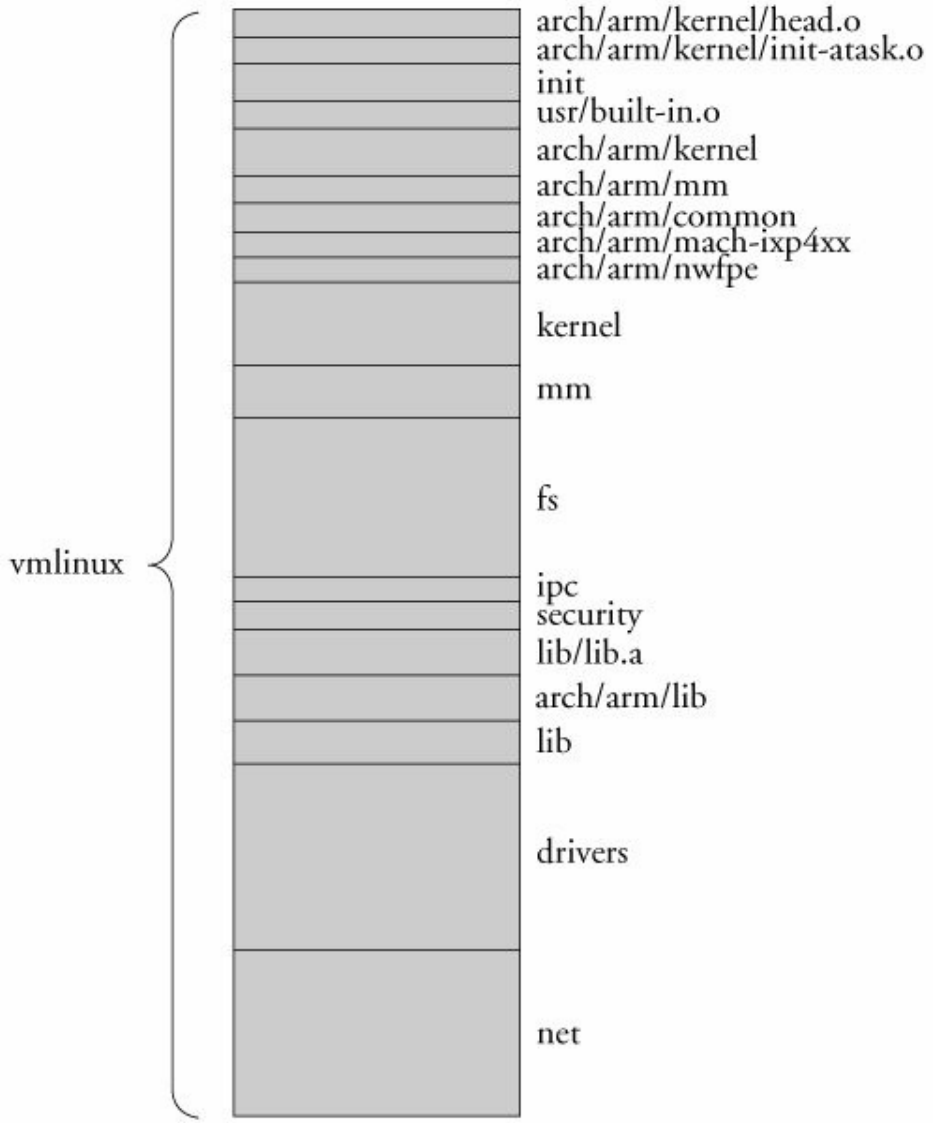
Located in arch/<arch>/boot/compressed

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

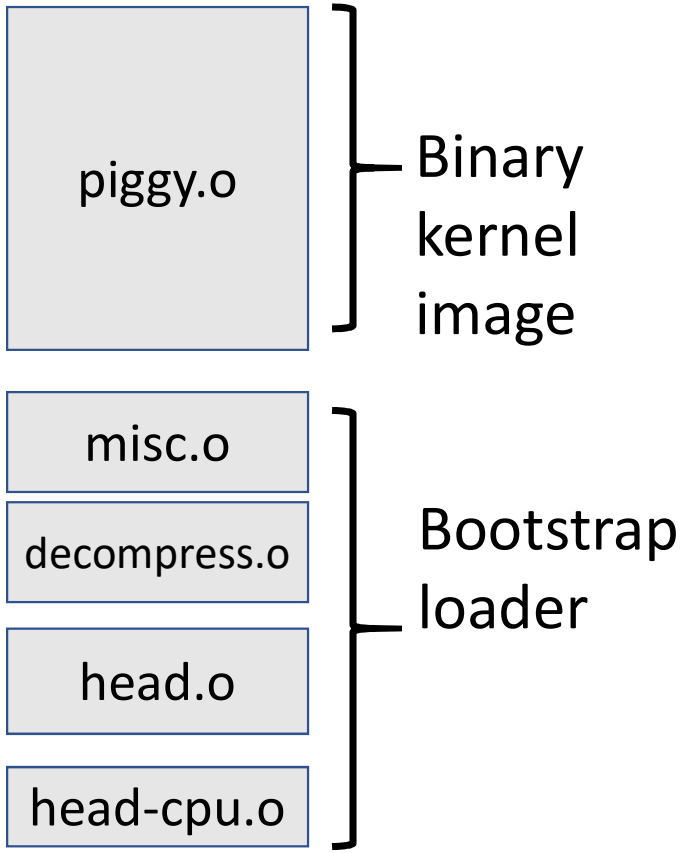
vmlinux

- 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



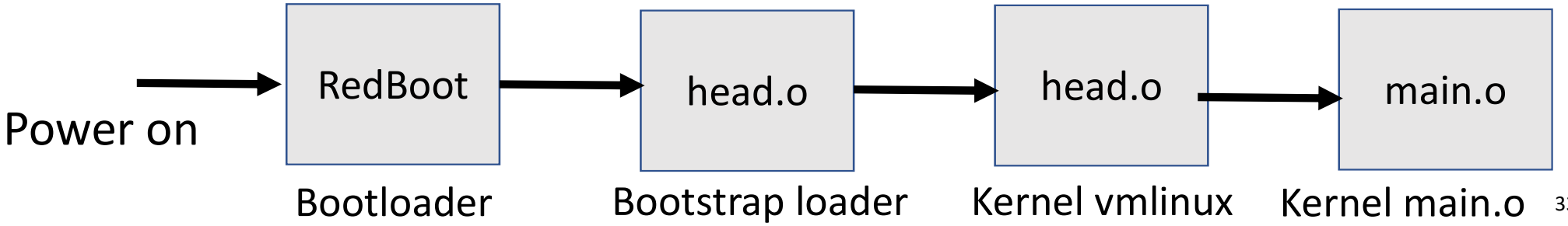
Bootstrap Loader

- **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



Kernel startup: main.c

- The final task performed by the kernel's own head.o
  - Control is passed from head.o to the start_kernel() in …/init/main.c
  - Most of the Linux kernel initialization takes place in this routine
- The function setup_arch() in start_kernel()
  - Identify the specific CPU
  - Provides a mechanism for calling high-level CPU-specific initialization routines

https://elixir.bootlin.com/linux/latest/source/init/main.c

Start_kernel main actions

- Call setup_arch (& command_line)
  - Function defined in arch/<arch>/kernel/setup.c
  - Copying the command line from where the bootloader left it
  - On ARM, this function calls
    - setup_processor: CPU information is display
    - setup_machine: locating the machine in the list of supported machines
  - Initializes the console (to get error messages)
  - Initializes many subsystems
  - Eventually calls rest_init

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();
}
```

Kernel_init

- Kernel_init does two main things
  - Call do_basic_setup in ../init/main.c
  - Once kernel services are ready, start device initialization (Linux 2.6.36 excerpt):

```
static void __init do_basic_setup(void)
{
    cpuset_init_smp();
    usermodehelper_init();
    init_tmpfs();
    driver_init();
    init_irq_proc();
    do_ctors();
    do_initcalls();
}
```

do_initcalls

- The initcall mechanism is to determine correct order of the built-in modules and subsystems initializatior
- Defined in
  - include/linux/init.h

```
/*
 * A "pure" initcall has no dependencies on anything else, and purely
 * initializes variables that couldn't be statically initialized.
 *
 * This only exists for built-in code, not for modules.
 */
#define pure_initcall(fn)               __define_initcall("0",fn,1)

#define core_initcall(fn)               __define_initcall("1",fn,1)
#define core_initcall_sync(fn)          __define_initcall("1s",fn,1s)
#define postcore_initcall(fn)           __define_initcall("2",fn,2)
#define postcore_initcall_sync(fn)      __define_initcall("2s",fn,2s)
#define arch_initcall(fn)               __define_initcall("3",fn,3)
#define arch_initcall_sync(fn)          __define_initcall("3s",fn,3s)
#define subsys_initcall(fn)             __define_initcall("4",fn,4)
#define subsys_initcall_sync(fn)        __define_initcall("4s",fn,4s)
#define fs_initcall(fn)                 __define_initcall("5",fn,5)
#define fs_initcall_sync(fn)            __define_initcall("5s",fn,5s)
#define rootfs_initcall(fn)             __define_initcall("rootfs",fn,rootfs)
#define device_initcall(fn)             __define_initcall("6",fn,6)
#define device_initcall_sync(fn)        __define_initcall("6s",fn,6s)
#define late_initcall(fn)               __define_initcall("7",fn,7)
#define late_initcall_sync(fn)          __define_initcall("7s",fn,7s)
```

init_post

- The last step of Linux booting
  - First tries to open a console
  - Then tries to run the init process
  - Effectively turning the current kernel thread into the user space init process

init_post Code: init/main.c

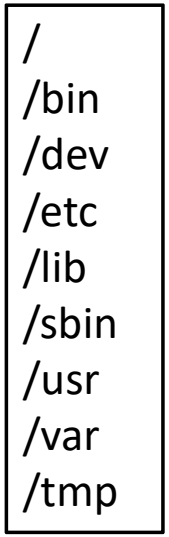
```
static noinline int init_post(void) __releases(kernel_lock) {
    /* need to finish all async __init code before freeing the memory */
    async_synchronize_full();
    free_initmem();
    mark_rodata_ro();
    system_state = SYSTEM_RUNNING;
    numa_default_policy();

    current->signal->flags |= SIGNAL_UNKILLABLE;
    if (ramdisk_execute_command) {
        run_init_process(ramdisk_execute_command);
        printk(KERN_WARNING "Failed to execute %s\n", ramdisk_execute_command);
    }

    /* We try each of these until one succeeds.
     * The Bourne shell can be used instead of init if we are
     * trying to recover a really broken machine. */
    if (execute_command) {
        run_init_process(execute_command);
        printk(KERN_WARNING "Failed to execute %s.  Attempting defaults...\n", execute_command);
    }
    run_init_process("/sbin/init");
    run_init_process("/etc/init");
    run_init_process("/bin/init");
    run_init_process("/bin/sh");

    panic("No init found. Try passing init= option to kernel. See Linux Documentation/init.txt");
}
```

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

Root file system

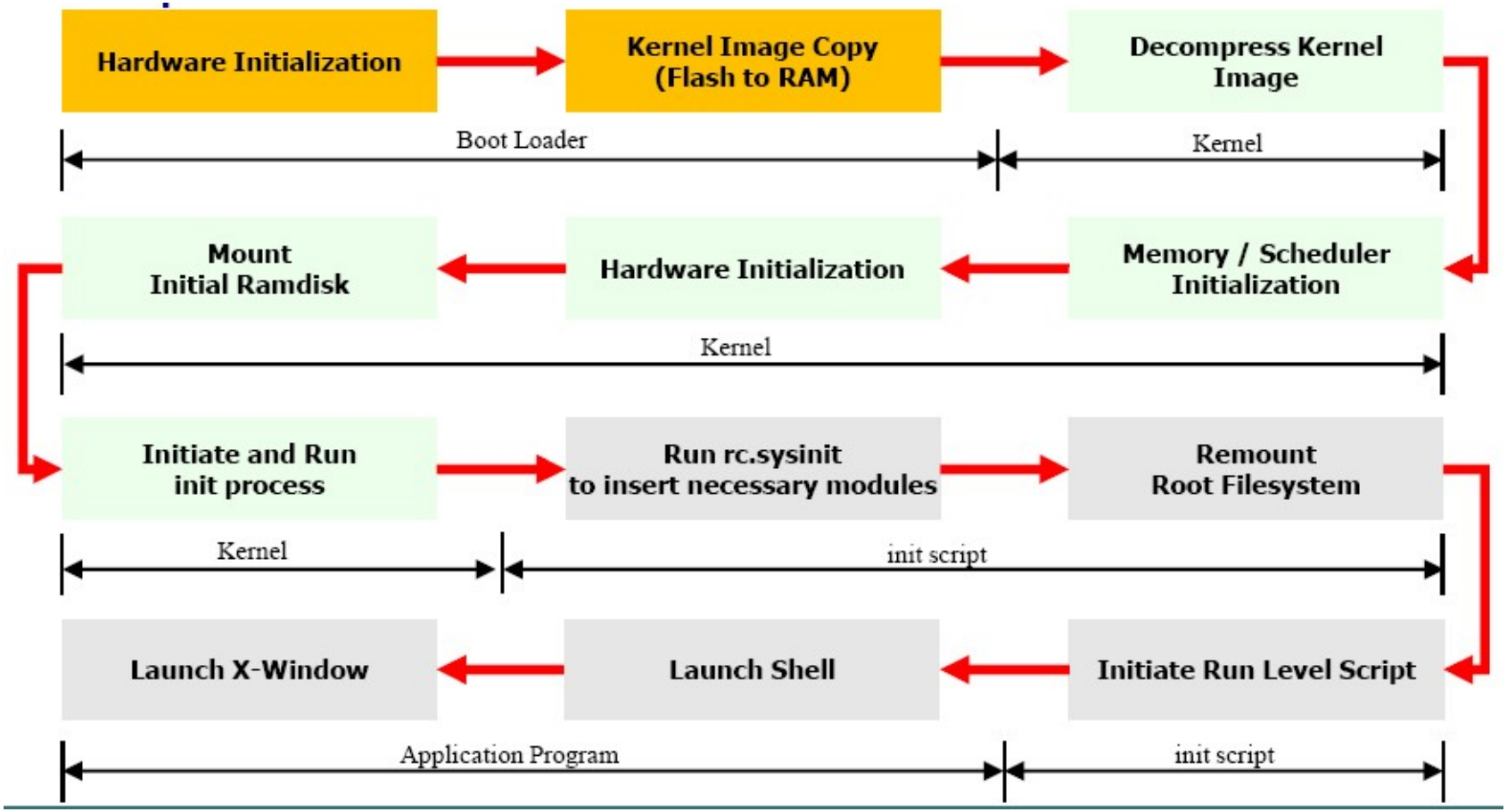
- **The root file system**
  - Refer to the file system mounted at the base of the file system hierarchy, designated simply as /
  - Contains programs and utilities to boot a system and initialize services
- **Initial RAM Disk (initrd)**
  - A small self-contained root file system
  - Contains directives 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



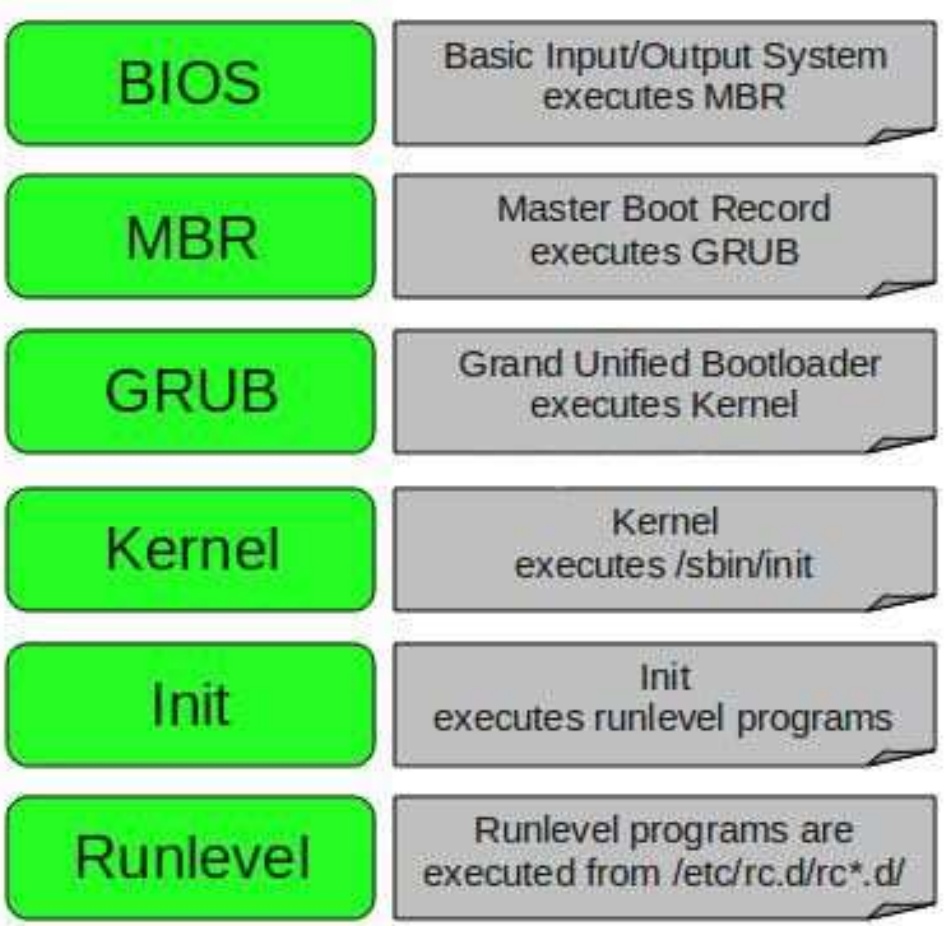
initramfs

- **initramfs**
  - Executing early user space programs
  - initramfs is loaded before the call to do_basic_setup(), which loads firmware for devices before its driver has been loaded
  - initramfs is a cpio archive, whereas initrd is a gizpped file system image -> much easier to use
  - initramfs is integrated into Linux kernel source tree and is built automatically when building the kernel image

Loading kernel



Booting kernel



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

Summary

- The bootloader executes bootstrap code
- Bootstrap code initializes the processor and board, and un-compresses the kernel code to RAM, and calls the kernel's start_kernel function
- Copies the command line from the bootloader
- Identifies the processor and machine
- Initializes the console
- Initializes kernel services (memory allocation, scheduling, file cache …)
- Creates a new kernel thread (init process) and continues in the idle loop
- Initializes devices and execute initcalls