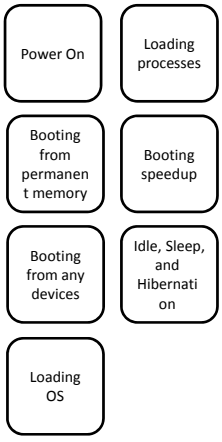
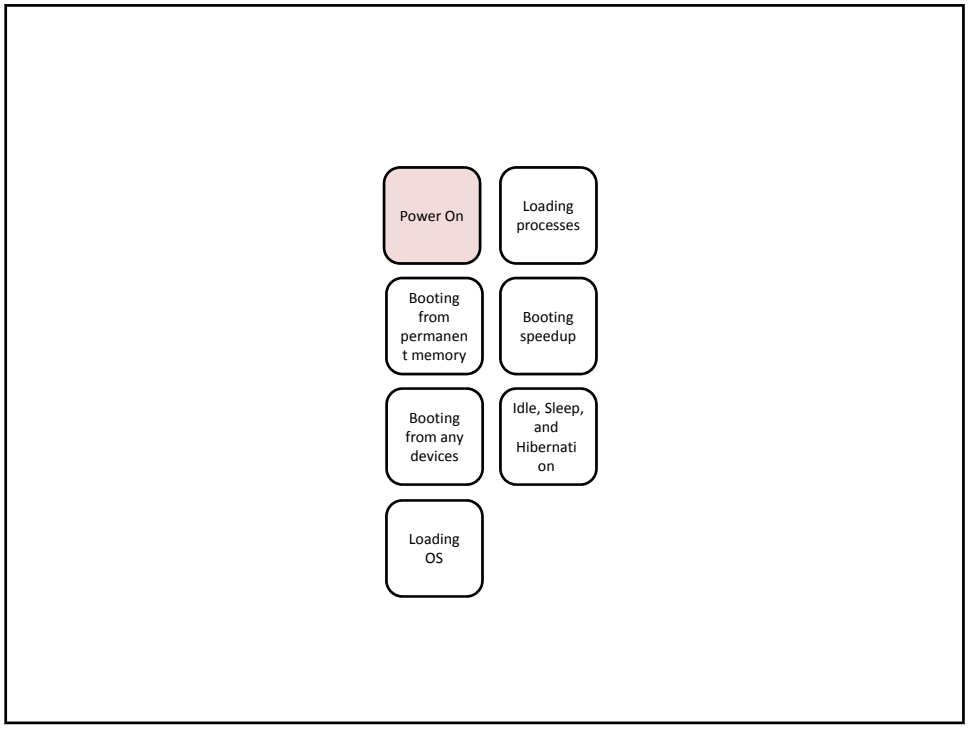
Operating System Design and Implementation

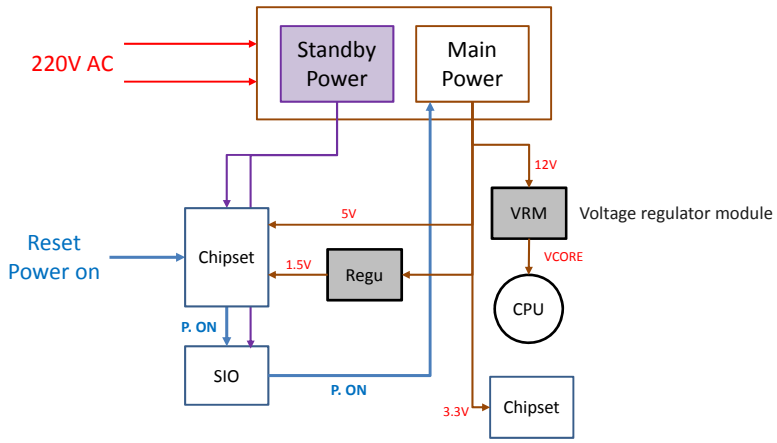
Booting process

Shiao-Li Tsao





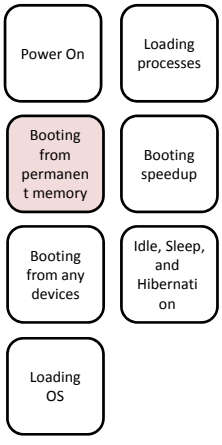
High Level View of Monthboard Power Delivery Circuits



Source: http://laptoprepairegypt.com/blog/tag/analysis-of-resource-management-on-the-motherboard-circuit-diagram/

References

- http://www.kitguru.net/components/power-supplies/ironlaw/how-computer-power-supplies-work-kitguru-guide/
- http://www.hardwaresecrets.com/everything-you-need-to-know-about-the-motherboard-voltage-regulator-circuit/



Why four loaders are necessary

- First loader (bootloader in permanent memory such as ROM/NOR flash/randomly access and byte addressable)
  - Processor can run but knows nothing about the whole system
  - Prepared by by SoC vendors or motherboard vendors to make sure there is no problem with the whole system
  - Check/initialize the hardware
  - May provide basic services to other programs
  - May provide shells to end-users for basic operations
  - Load/jump to the second loader (based on predefined procedures/configurations)

Why four loaders are necessary (Cont.)

- Second loader (bootloader in any devices such as DISK/CD-ROM/NAND flash/remote server/…)
  - As soon as first bootloader can access the second loader (on any device), load it into memory, and can jump to the starting instruction
  - Prepare by OS vendors or other third party vendors
    - Stored in the correct location/in correct format)
    - Must know how to loader OSs
  - May provide shell to end-users to select OSs (multi-booting) or set configurations
  - Loading OS into memory
  - Optional for embedded processors

Why four loaders are necessary (Cont.)

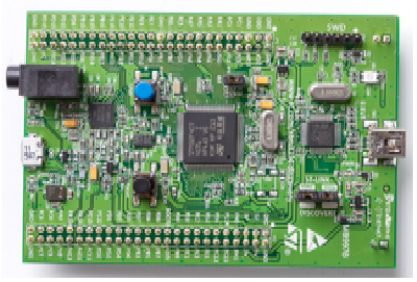
- Third loader (OS loader that loads OS)
  - A program executes without OS services
  - Initial and prepare OS services
- Fourth loader (processes replies on OS to fork/exec other processes)
  - OS is now ready and can provide services
  - Any process calls OS services to fork/execute other processes

1st boot-loader functions

- Initialize the hardware setting
  - Power on self test (POST) in x86/BIOS
- Basic monitor and debugger
- Pass the control to the 2nd bootloader

First Bootlader - embedded processor

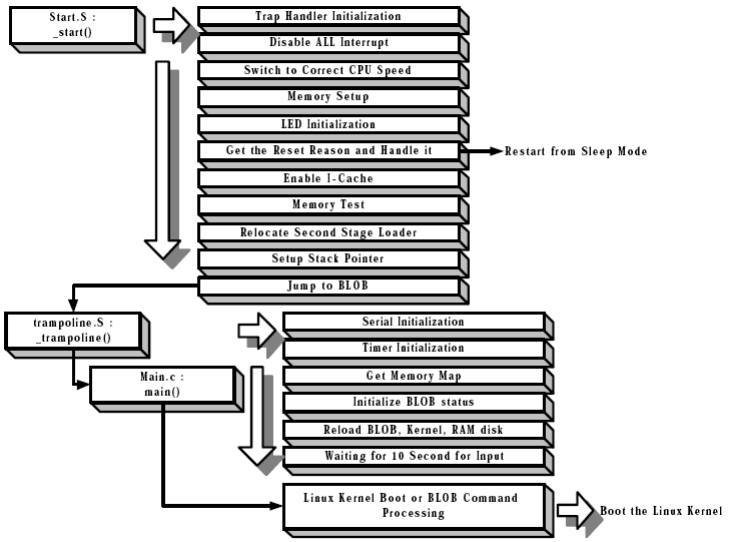
- SoC: e.g. STM32F4
- Reference:
  - http://www.st.com/web/en/resource/technical/document/datasheet/DM00037051.pdf
  - p.18



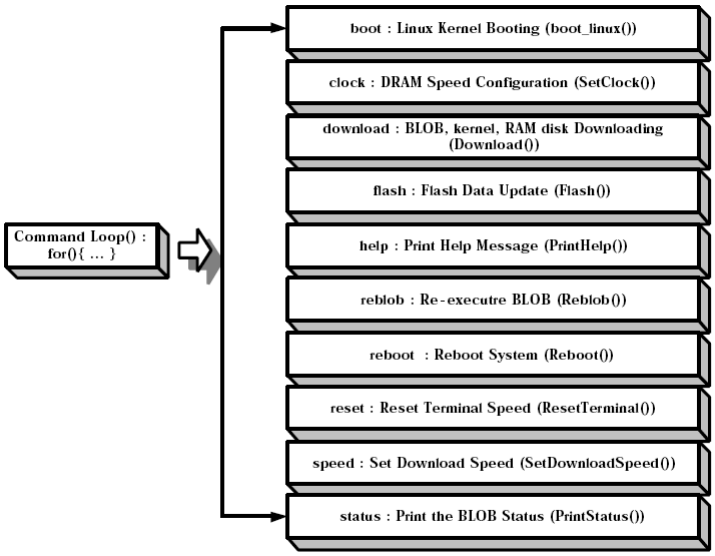
First Bootlader - embedded processor

- SoC: e.g. STM32F4
  - Memory space
- Reference:
  - http://www.st.com/web/en/resource/technical/document/datasheet/DM00037051.pdf
  - p.70

Boot-loader Example



Boot-loader Example



ARM Examples

- Vector table

```
.text

/* Jump vector table as in table 3.1 in [1] */
.globl _start
_start:       b      reset
       b      undefined_instruction
       b      software_interrupt
       b      prefetch_abort
       b      data_abort
       b      not_used
       b      irq
       b      fiq
```

Reset_Handler

```
/* the actual reset code */
reset:
       /* First, mask **ALL** interrupts */
       ldr    r0, IC_BASE
       mov    r1, #0x00
       str    r1, [r0, #ICMR]


       /* switch CPU to correct speed */
       ldr    r0, PWR_BASE
       LDR    r1, cpuspeed
       str    r1, [r0, #PPCR]


       /* setup memory */
       bl     memsetup

       /* init LED */
       bl     ledinit
```

Reset_Handler (Cont.)

```
       /* check if this is a wake-up from sleep */
       ldr    r0, RST_BASE
       ldr    r1, [r0, #RCSR]      Reset status register
       and    r1, r1, #0x0f
       teq    r1, #0x08
       bne    normal_boot   /* no, continue booting */

       /* yes, a wake-up. clear RCSR by writing a 1 (see
9.6.2.1 from [1]) */
       mov    r1, #0x08
       str    r1, [r0, #RCSR]     ;

       /* get the value from the PSPR and jump to it */
       ldr    r0, PWR_BASE
       ldr    r1, [r0, #PSPR]      Power manager scratch pad register
       mov    pc, r1
```

Reset_Handler (Cont.)

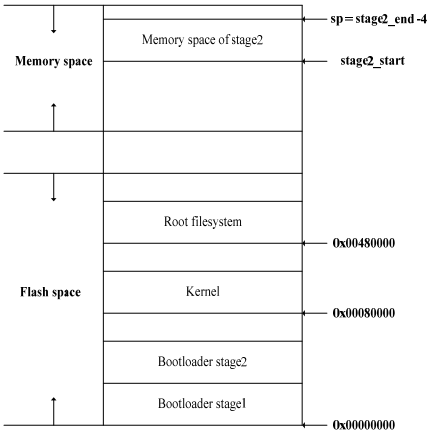
```
normal_boot:
       /* enable I-cache */
       mrc    p15, 0, r1, c1, c0, 0      @ read control reg
       orr    r1, r1, #0x1000            @ set Icache
       mcr    p15, 0, r1, c1, c0, 0      @ write it back

       /* check the first 1MB  in increments of 4k */
       mov    r7, #0x1000
       mov    r6, r7, lsl #8      /* 4k << 2^8 = 1MB */
       ldr    r5, MEM_START

mem_test_loop:
       mov    r0, r5
       bl     testram
       teq    r0, #1
       beq    badram

       add    r5, r5, r7
       subs   r6, r6, r7
       bne    mem_test_loop
```

```
       /* the first megabyte is OK, so let's clear it */
       mov    r0, #((1024 * 1024) / (8 * 4))    /* 1MB in
steps of 32 bytes */
       ldr    r1, MEM_START
…
clear_loop:
       stmia  r1!, {r2-r9}
       subs   r0, r0, #(8 * 4)
       bne    clear_loop

       /* relocate the second stage loader */
       add    r2, r0, #(128 * 1024)      /* blob is 128kB
*/
       add    r0, r0, #0x400             /* skip first 1024
bytes */
       ldr    r1, MEM_START
       add    r1, r1, #0x400             /* skip over here
as well */
..
copy_loop:
       ldmia  r0!, {r3-r10}
       stmia  r1!, {r3-r10}
       cmp    r0, r2
       ble    copy_loop
```

Reset_Handler (Cont.)

```
       /* set up the stack pointer */
       ldr    r0, MEM_START
       add    r1, r0, #(1024 * 1024)
       sub    sp, r1, #0x04


       /* blob is copied to ram, so jump to it */
       add    r0, r0, #0x400
       mov    pc, r0
```

Bootloader memory map



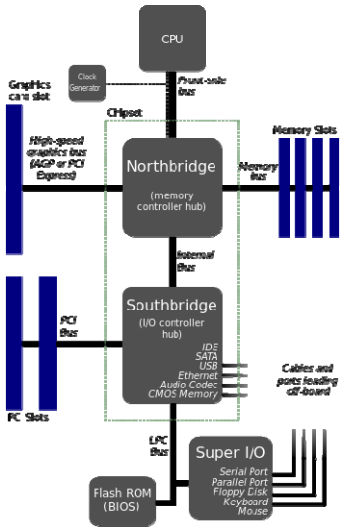
Boot-loader C program

```
int main(void)
{
~
   led_on();
~
   SerialInit(baud9k6);
   TimerInit();
~
   SerialOutputString(PACKAGE " version " VERSION  "\n"
                "Copyright (C) 1999 2000 2001 "
~
   get_memory_map();
~
   SerialOutputString("Running from ");
   if(RunningFromInternal())
       SerialOutputString("internal");
   else
       SerialOutputString("external");
…
```

Boot-loader C program

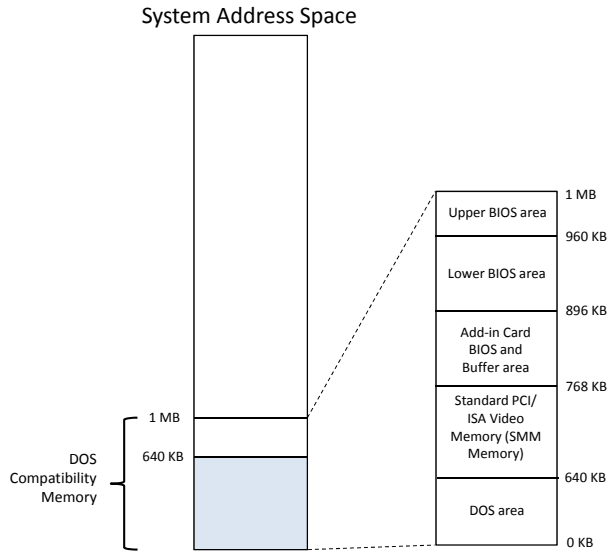
```
   /* wait 10 seconds before starting autoboot */
SerialOutputString("Autoboot in progress, press any key…");
   for(i = 0; i < 10; i++) {
        SerialOutputByte('.');
        retval = SerialInputBlock(commandline, 1, 1);
~
   if(retval == 0) {
~
        boot_linux(commandline);   }
~
   for(;;) {
~
        if(numRead > 0) {
        if(MyStrNCmp(commandline, "boot", 4) == 0) {
               boot_linux(commandline + 4);
        } else if(MyStrNCmp(commandline, "clock", 5) == 0) {
               SetClock(commandline + 5);
   } else if(MyStrNCmp(commandline, "download ", 9) == 0) {
~  return 0;
} /* main */
```

First Bootloader – x86

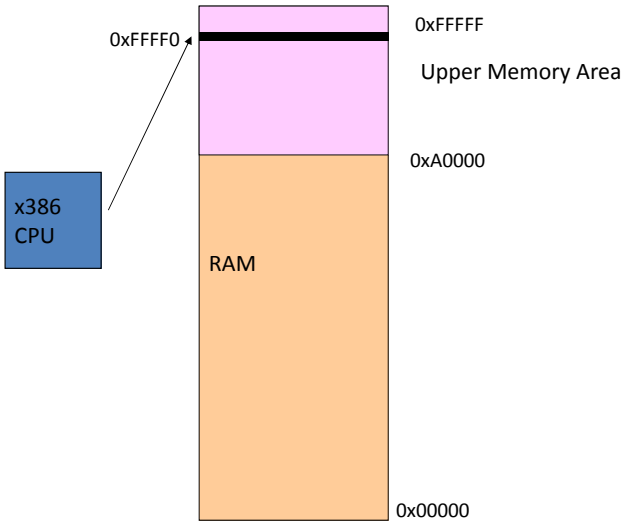


Source: Wikipedia (https://en.wikipedia.org/wiki/Northbridge_(computing))

First Bootloader – x86



PC Booting (Cont)

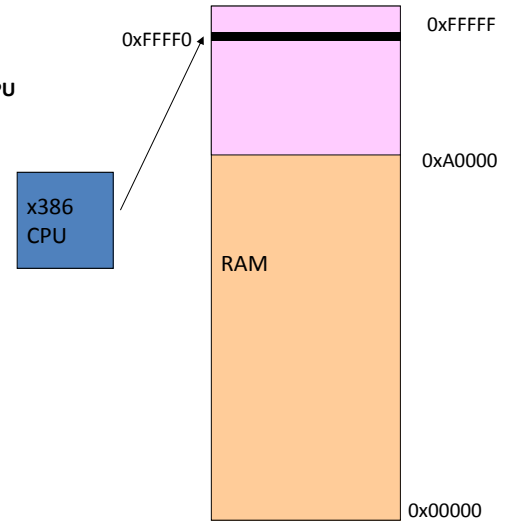


UMA

Address	First 16K (x0000h-x3FFFh)	Second 16K (x4000h-x7FFFh)	Third 16K (x8000h-xBFFFh)	Fourth 16K (xC000h-xFFFFh)
A0000-AFFFFh	VGA Graphics Mode Video			
B0000- BFFFFh	VGA Monochrome Text Mode Video RAM		VGA Color Text Mode Video RAM	
C0000- CFFFFh	VGA Video BIOS ROM		IDE Hard Disk BIOS ROM	Optional Adapter ROM BIOS or RAM UMBs
D0000- DFFFFh	Optional Adapter ROM BIOS or RAM UMBs			
F0000- FFFFFh	System BIOS ROM			

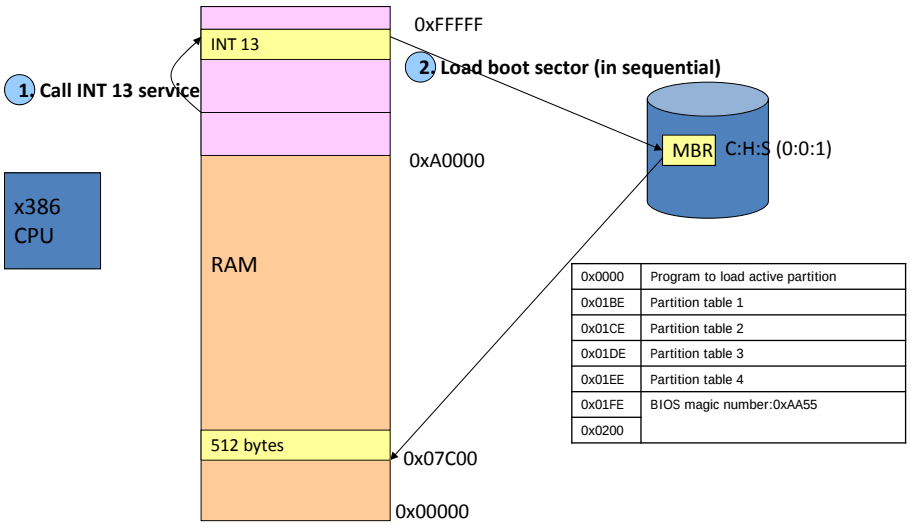
PC Booting (Cont)

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. **POST**
6. **Beep if there is an error**
7. **Read CMOS data/settings**
8. **Run 2nd-stage boot**

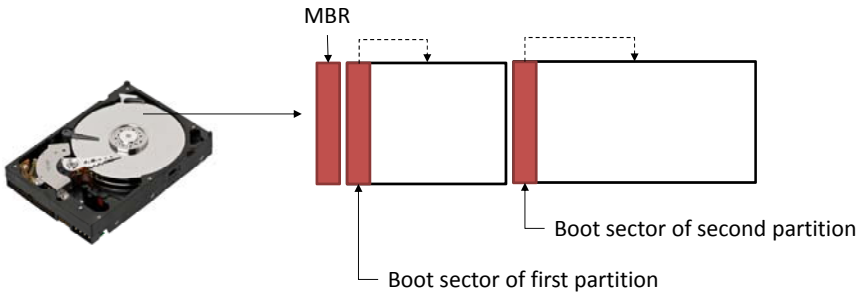


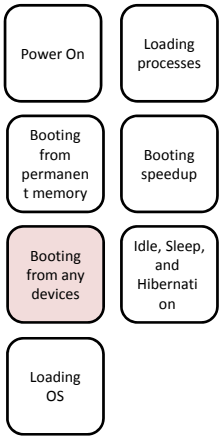
INT	Address	Type	Description
00h	0000:0000h	Processor	Divide by zero
01h	0000:0004h	Processor	Single step
02h	0000:0008h	Processor	Non maskable interrupt (NMI)
03h	0000:000Ch	Processor	Breakpoint
04h	0000:0010h	Processor	Arithmetic overflow
05h	0000:0014h	Software	Print screen
06h	0000:0018h	Processor	Invalid op code
07h	0000:001Ch	Processor	Coprocessor not available
08h	0000:0020h	Hardware	System timer service routine
09h	0000:0024h	Hardware	Keyboard device service routine
0Ah	0000:0028h	Hardware	Cascade from 2nd programmable interrupt controller
0Bh	0000:002Ch	Hardware	Serial port service - COM post 2
0Ch	0000:0030h	Hardware	Serial port service - COM port 1
0Dh	0000:0034h	Hardware	Parallel printer service - LPT 2
0Eh	0000:0038h	Hardware	Floppy disk service
0Fh	0000:003Ch	Hardware	Parallel printer service - LPT 1
10h	0000:0040h	Software	Video service routine
11h	0000:0044h	Software	Equipment list service routine
12h	0000:0048H	Software	Memory size service routine
13h	0000:004Ch	Software	Hard disk drive service
14h	0000:0050h	Software	Serial communications service routines
15h	0000:0054h	Software	System services support routines
16h	0000:0058h	Software	Keyboard support service routines
17h	0000:005Ch	Software	Parallel printer support services

PC Booting (Cont)

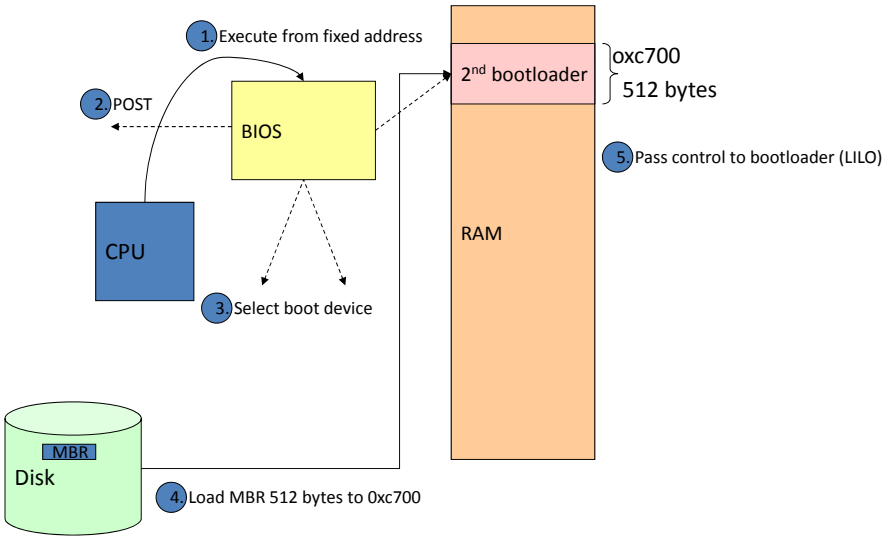


MBR

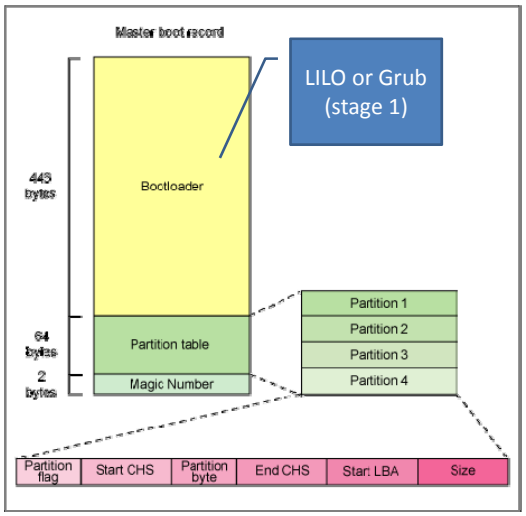

MBR
Boot sector of second partition
Boot sector of first partition

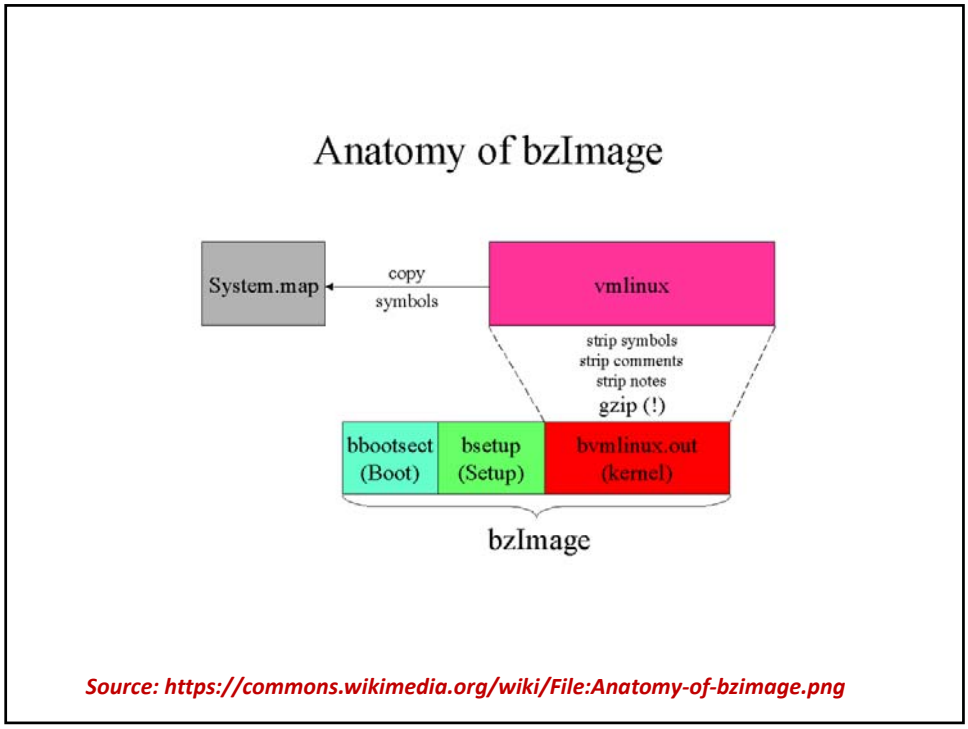

Power On
Loading processes
Booting from permanent memory
Booting speedup
Booting from any devices
Idle, Sleep, and Hibernation
Loading OS

Linux Boot Example

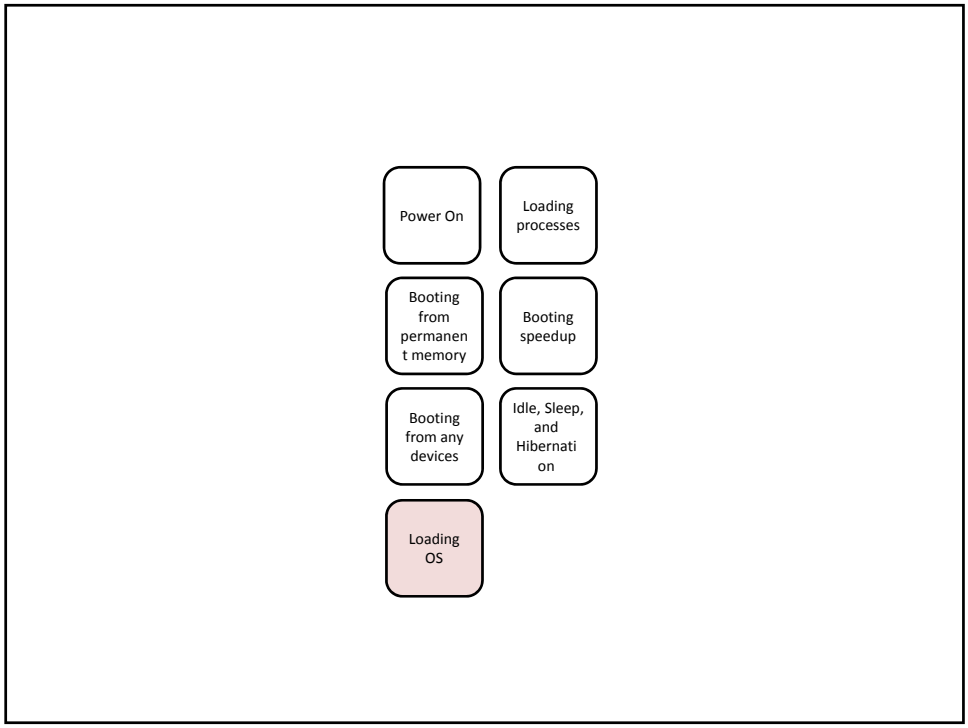


MBR (Master Boot Record)

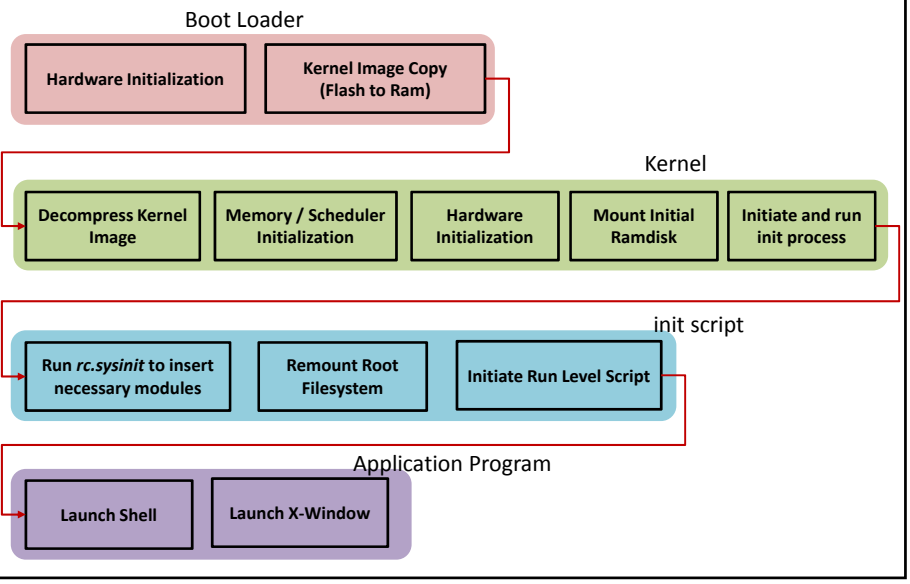




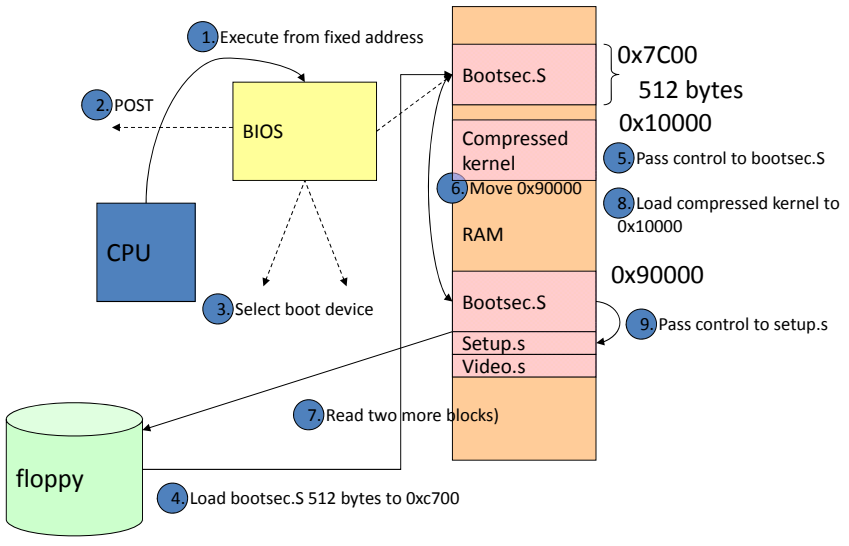
Source: https://commons.wikimedia.org/wiki/File:Anatomy-of-bzimage.png



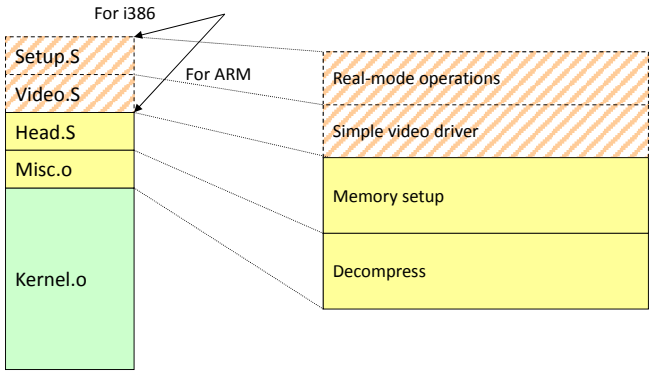
Loading Kernel



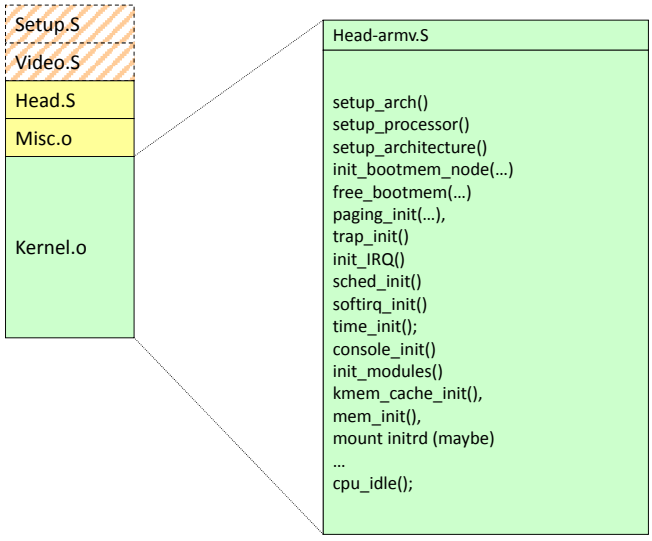
Linux Boot Example



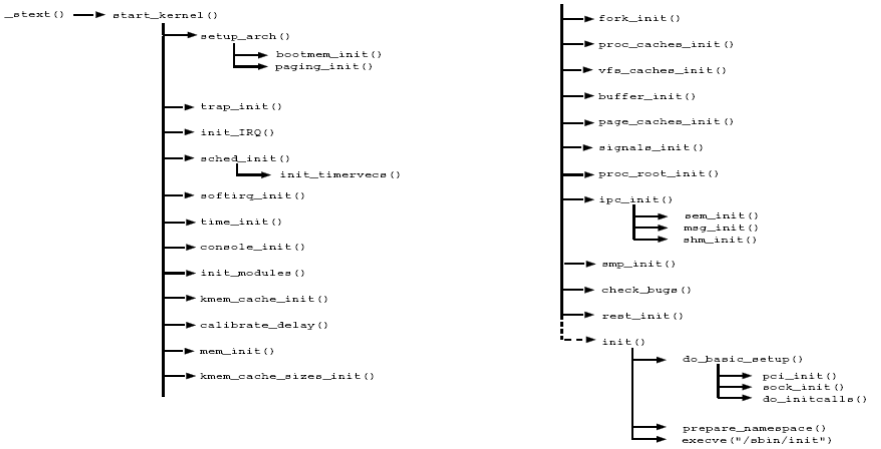
Kernel Image Structure



Kernel Image Structure



Linux booting



Power On

Loading processes

Booting from permanent memory

Booting speedup

Booting from any devices

Idle, Sleep, and Hibernation

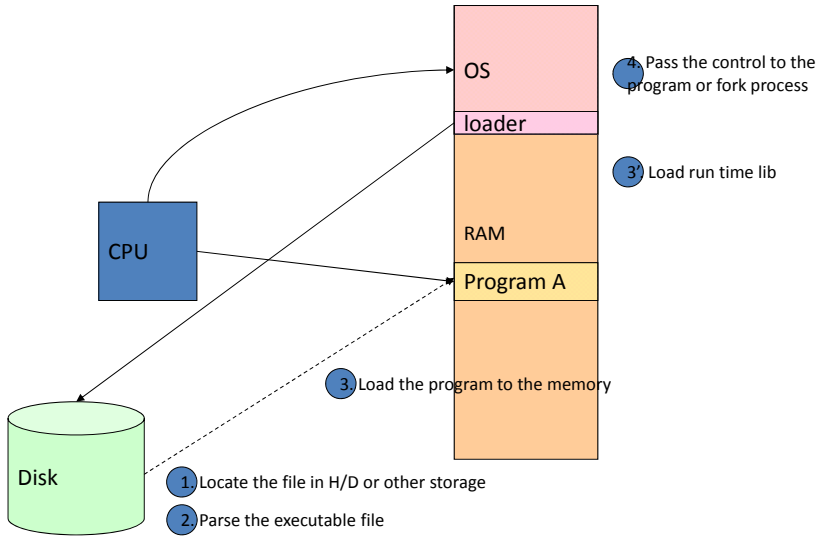
Loading OS

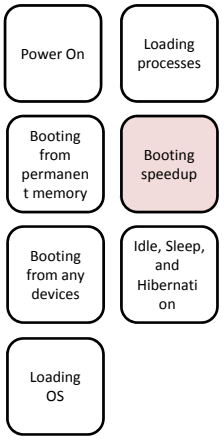
Mount root disk

Source: http://linux-development-for-fresher.blogspot.tw/2012/07/linux-boot-process-in-nutshell.html

initrd (initramfs): boot loader initialized RAM disk

Loader





Bootloader Speedups

- Remove waiting time
- Removing unnecessary initialization routines
- Uncompressed kernel
- DMA Copy Of Kernel On Startup
- Fast Kernel Decompression
- Kernel XIP

Uncompressed kernel

- Fast boot, but imagine size has been larger
  - 2MB – 2.5MB non-compressed (ARM)
  - 1MB – 1.5MB compressed (ARM)
- It should be different results, performance of CPUs, speed of flash memory
  - Profiling is required
- Make Image vs. make zImage

Fast Kernel Decompression

- Use other compression/decompression algorithm
  - Slow compression, good compression ratio, fast decompression
- GZIP vs. Sony UCL
- Small kernel size, fast kernel loading time

Fast Kernel Decompression

Image file:	initrd-2.6.5-1.358		Power PC
method	UCL	GZIP	improved %
parameter	-b4194304	-8	.
source file size	819200	819200	.
compressed size	187853	189447	.
compression rate	77.1%	76.9%	0.3%
compression time: user (sec)	5.13	2.03	-152.5%
sys (sec)	0.09	0.06	-36.5%
total (sec)	5.22	2.09	-149.0%
decompression time: user (sec)	0.12	0.3	59.7%
sys (sec)	0.1	0.08	-16.9%
total (sec)	0.22	0.39	43.0%

Fast Kernel Decompression

Image file:	vmlinux-2.4.20 for ibm-440gp		PowerPC
method	UCL	GZIP	improved %
parameter	-b4194304	-8	.
source file size	1810351	1810351	.
compressed size	790250	776807	.
compression rate	56.3%	57.1%	-1.3%
compression time: user (sec)	17.29	6.07	-185.0%
sys (sec)	0.04	0.02	-92.4%
total (sec)	17.33	6.09	-184.6%
decompression time: user (sec)	0.12	0.16	26.1%
sys (sec)	0.03	0.04	35.8%
total (sec)	0.15	0.2	28.2%

Kernel XIP

- Direct addressing on NOR flash memory
- Cannot compress kernel
- PowerPC 405LP/266 MHZ

Boot Stage	Non-XIP Time	XIP Time
Copy kernel to RAM	85 ms	12 ms *
Decompress kernel	453 ms	0 ms
Kernel time to initialize (time to first user space program)	819 ms	882 ms
Total kernel boot time	1357 ms	894 ms
Reduction:	*	**463 ms**

Kernel XIP (Cont.)

- TI OMAP 5912/196 MHZ

Boot Stage	Non-XIP Time Kernel compressed	Non-XIP Time Kernel not compressed	XIP Time
Copy kernel to RAM	56 ms	120 ms	0 ms
Decompress kernel	545 ms	0 ms	0 ms
Kernel time to initialize (time to first user space program)	88 ms	208 ms	110 ms
Total kernel boot time	689 ms	208 ms	110 ms
Reduction:	*	**481 ms**	**579 ms**

Embedded Linux Speedups

- Removing unnecessary message printout
- Remove unnecessary functions and device drivers
- Modularization of device driver
- Asynchronous function calls
- Avoid performance measurement routine
- RTC no sync
- Using read-only file system
- Using lazy mount technique on R/W file systems
- Deferred Initcalls

Application Speedups

- Using binary script, not shell script
- Using init process with simplified and optimized
- Parallel RC scripts
- Application XIP
- Using pre-link shared lib
- Optimize of application programs
- Move from glibc to uClibc