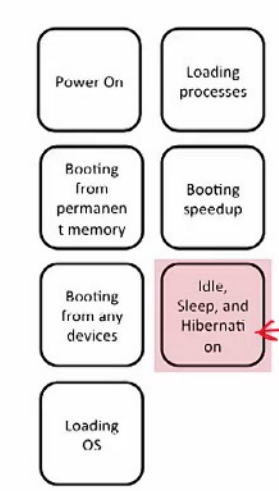
L2 Booting Process

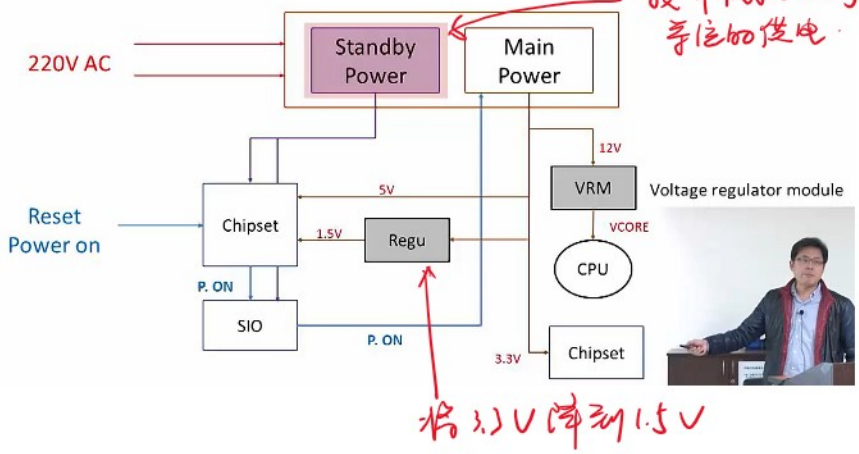
2019年12月1日 19:32



长时间 sleep 就是 Hibernation. (休眠)

Power On

High Level View of Monthboard Power Delivery Circuits



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/

Booting From Permanent Memory

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

e.g: BIOS for x86

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
- 1. Check/initialize the hardware
- 2. May provide basic services to other programs
- 3. May provide shells to end-users for basic operations
- 4. Load/jump to the second loader (based on predefined procedures/configurations)

BIOS for x86

固件

Permant mem

(Byte addressable)

固件里面的驱动(services)

Why four loaders are necessary (Cont.)

我在不附带 CPU 直接寻址的地方.

- Second loader (bootloaer 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

e.g. Grub

必须附带第一个 bootloader 有能力附载 load 它呆

手机上很少.

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

load the user application

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

eg.

uboot for embedded sys.

BIOS for x86.

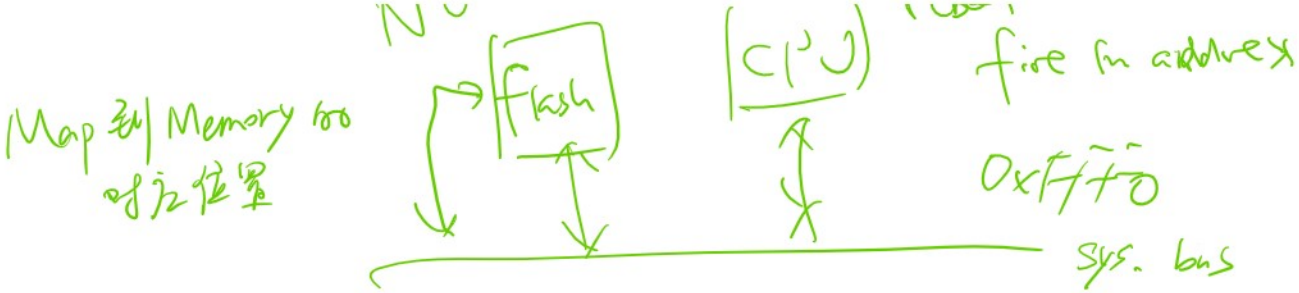
Hardware 层

NOR

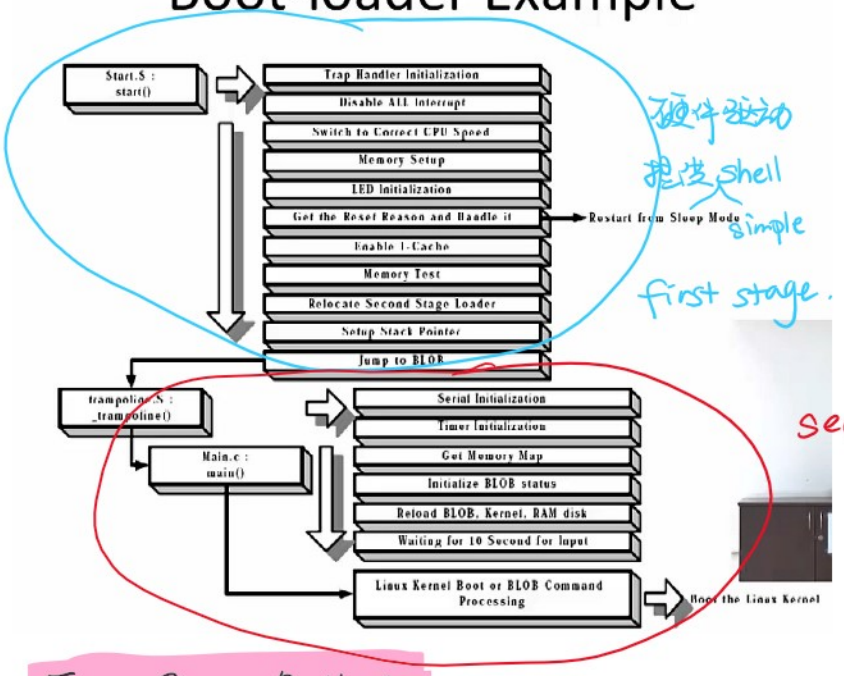
CPU

reboot

fixe in address



Boot-loader Example



硬件激活
提供shell
simple
first stage.
second stage

First Stage Bootloader

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
```

CPU一开机，第一件事情就是jump到Vector Table的reset entry

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
```

不希望"还没开张营业"前就有人可以"进来买东西"了

把系统CPU的speed

检查Mem是不是对的(写再读，判断Mem好不好)

LED灯亮了，说明first stage bootloader已经结束了

Reset_Handler (Cont.)

```
        /* check if this is a wake-up from sleep */
        ldr     r0, RST_BASE
```

Reset_Handler (Cont.)

```
        /* check if this is a wake-up from sleep */
        ldr     r0, RST_BASE
        ldr     r1, [r0, #RCSR]
        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]
        mov     pc, r1
```

Reset status register

Power manager scratch pad register

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
```

(判断Mem有没有问题)

清扫 Mem

```
        /* 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
```

把 second stage bootloader 搬到内存里去.

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

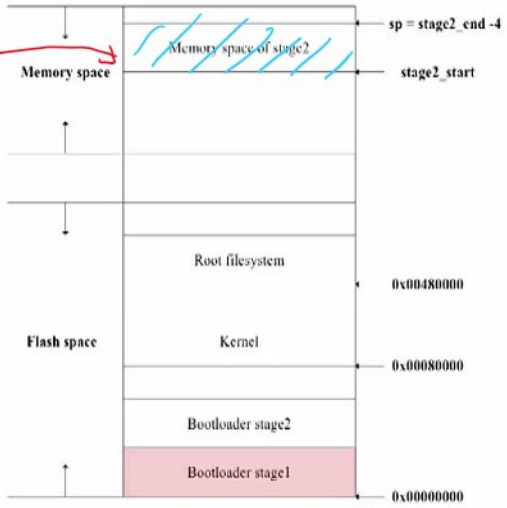
icache → 通常有重复性的才能提高

不重复的开 instruction cache 反而更慢 (要多读到 cache 一次)

在内存中清出一块地方, 用以存放 second stage loo bootloader.

这个段落未完. 会有 function call stack, mem read/write 等操作. 在 ROM 里没办法让 CPU 展开等脚

Bootloader memory map



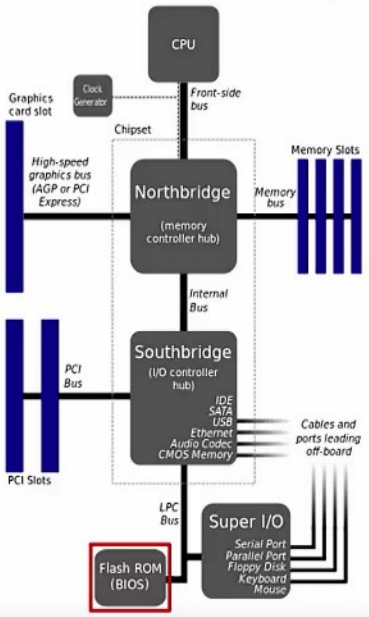
Second Stage Bootloader

Boot-loader C program

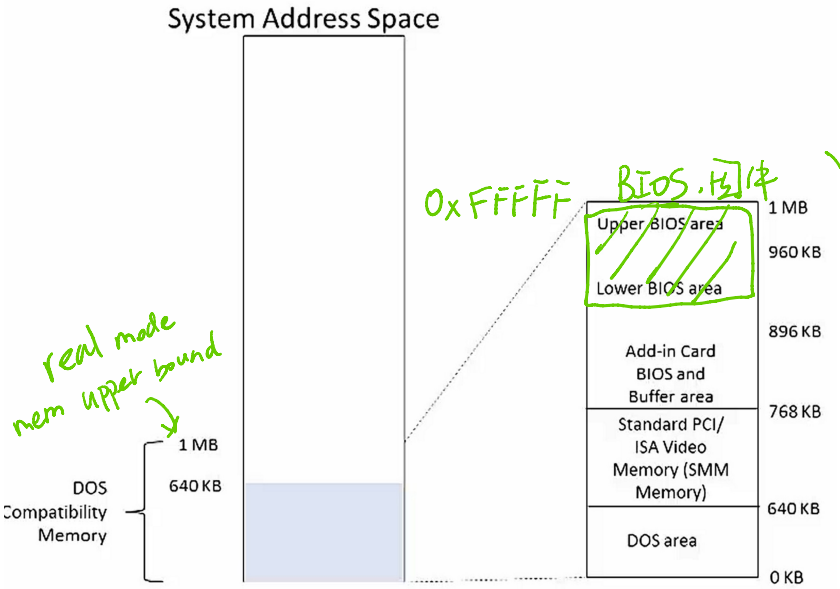
```
int main(void)
{
~
   led_on();
~
   SerialInit(baud9k6);
   TimerInit();
~
   SerialOutputString(PACKAGE " version " VERSION  "\n"
                "Copyright (C) 1999 2000 2001 "
~
   get_memory_map();
~
   SerialOutputString("Running from ");
   if(RunningFromInternal())
       SerialOutputString("internal");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

在Terminal上看到信息

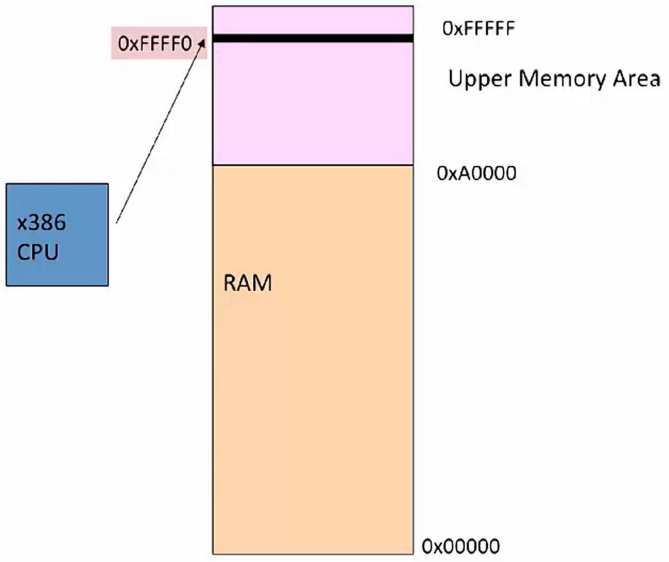
First Bootloader – x86



First Bootloader – x86



PC Booting (Cont)



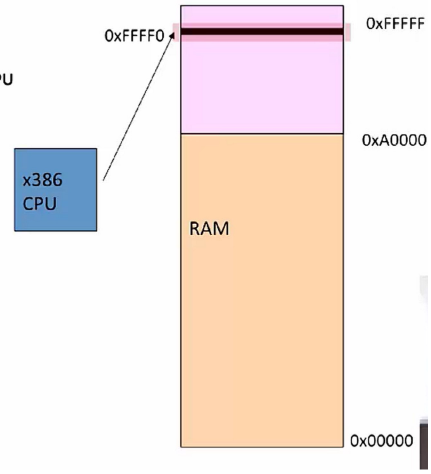
UMA

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)

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 Power on self test
6. Beep if there is an error
7. Read CMOS data/settings
8. Run 2nd-stage boot



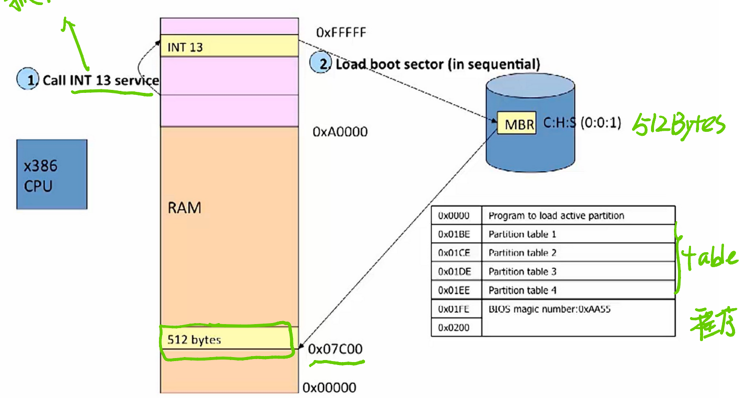
x86 define spec → BIOS 提供基本的服务.
→ 根据这些服务存取硬件资料
eg. 让 mouse, keyboard 动等.

BIOS 提供的基本中断服务:

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)

抓 MBR中的资料.

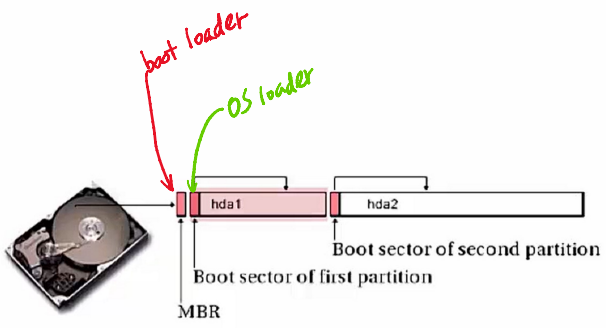


0x0000	Program to load active partition
0x01BE	Partition table 1
0x01CE	Partition table 2
0x01DE	Partition table 3
0x01EE	Partition table 4
0x01FE	BIOS magic number:0xAA55
0x0200	

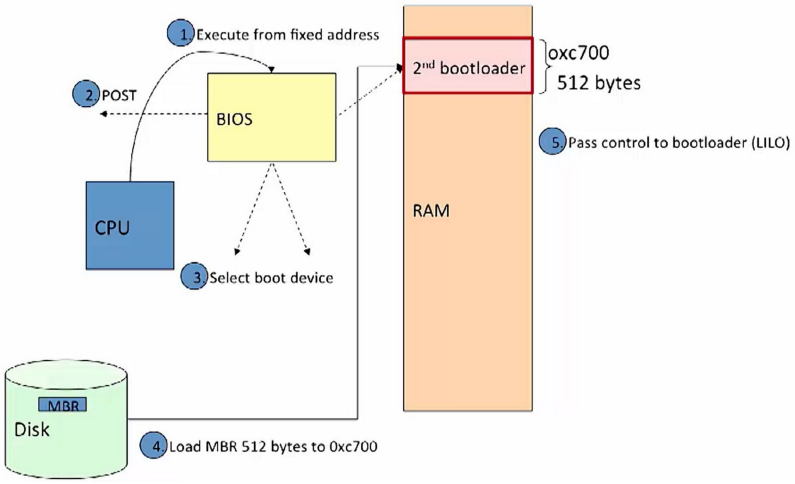
table: 说明这个HD被切割成几个partition

程序

MBR

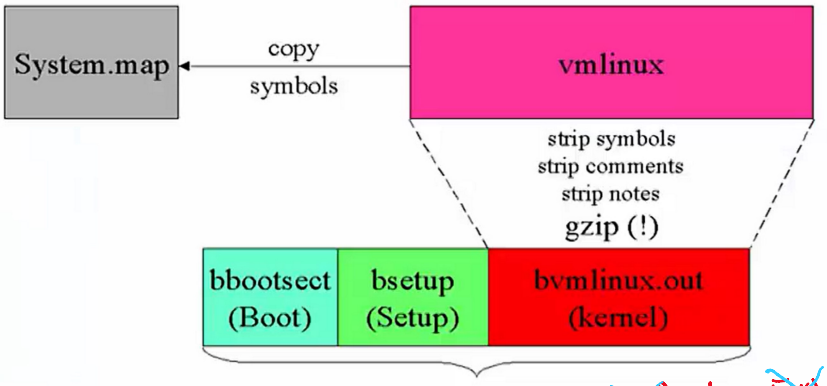


Linux Boot Example



到目前为止，CPU还认不到 file system.
(CPU是从绝对位置，(如MBR)去拿到那的，而不是相对位置)

Anatomy of bzImage



bzImage (~~压缩的~~，~~压缩后就在 partition 里~~)
不准确，应是自解压的。
最前面一段是没有被压缩过的，运算在被 Bootloader load 起来之后才能正常执行。会解压后面的压缩的。

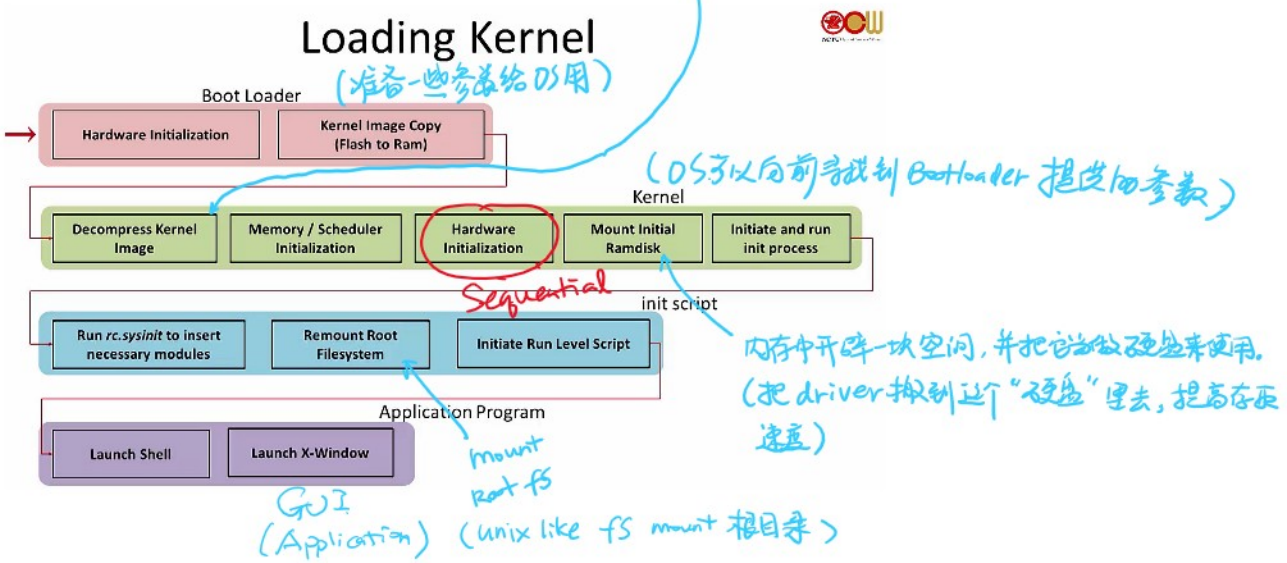
为什么要压缩 kernel？

- 因为硬盘的 I/O 速度很慢，所以压缩会快快。

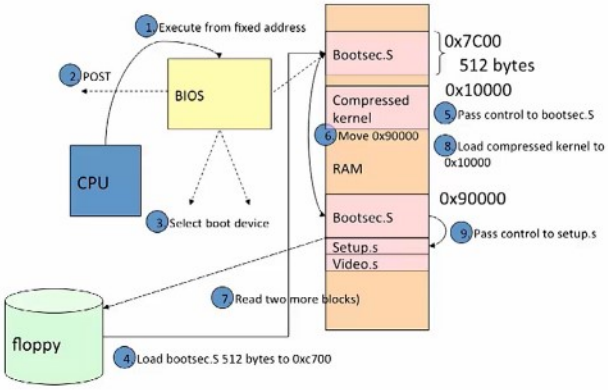
- 因为硬盘的 I/O 速度较慢，而解压速度较快.
- 在 Embedded 里面是 Not true 的，因为 CPU 能力弱，解压缩慢，而 Image 被存在 Flash mem 里，I/O 较快.

Load OS

如鸿蒙系统如果一款程序就是在这里.



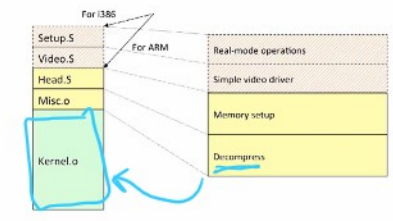
Linux Boot Example



Backward compatiable.

导致了很复杂.

Kernel Image Structure

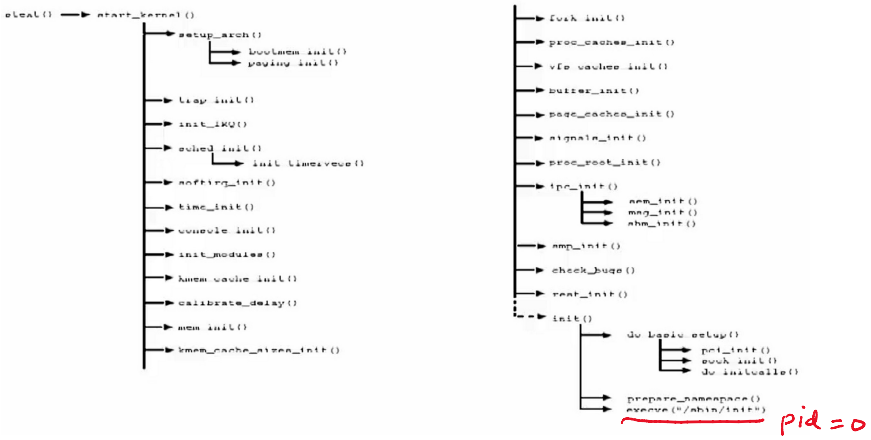


1MB 空间 → 4GB 空间

Real Mode → Protected Mode

Press Esc to exit full screen

Linux booting

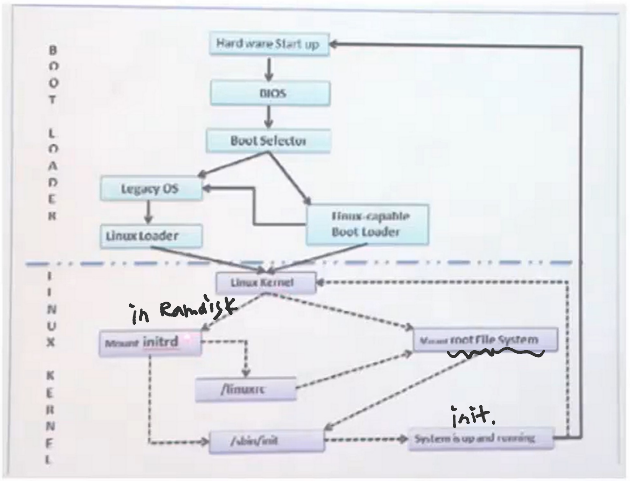


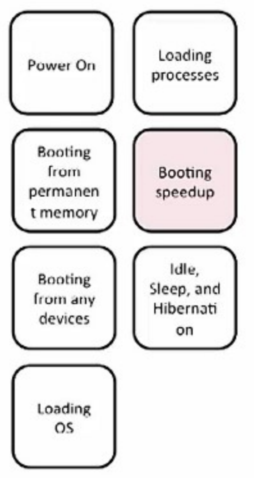
Call fork / execve
(OS service)

user program 或
1行与1行之间是不连续的 (对于CPU来看)
之间隔了用户 call library, library call
OS service 或 code.

a. 我们第3个 bootloader 执行完后，是由 Scheduler 指 PC 指到 init 程序去的.
会执行到 fork, 从而执行了一段 Kernel 的程序，再执行 exec. 从而执行到 Kernel
的程序，最后回到 init.

Mount root disk





加速

Bootloader Speedups

- Remove waiting time
- Removing unnecessary initialization routines
- Uncompressed kernel ← 对于一些 embedded system
- DMA Copy Of Kernel On Startup
- Fast Kernel Decompression ← 选更好的算法 (Async 的压缩办法. 压慢点没事. 解压要快)
- Kernel XIP ← eXecute In Place

直接把启动好的 memory 做成 image ? ← 放在 ROM 里

不行. 很多 kernel structure 是动态的. 完全静态的环境确实可能可行

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