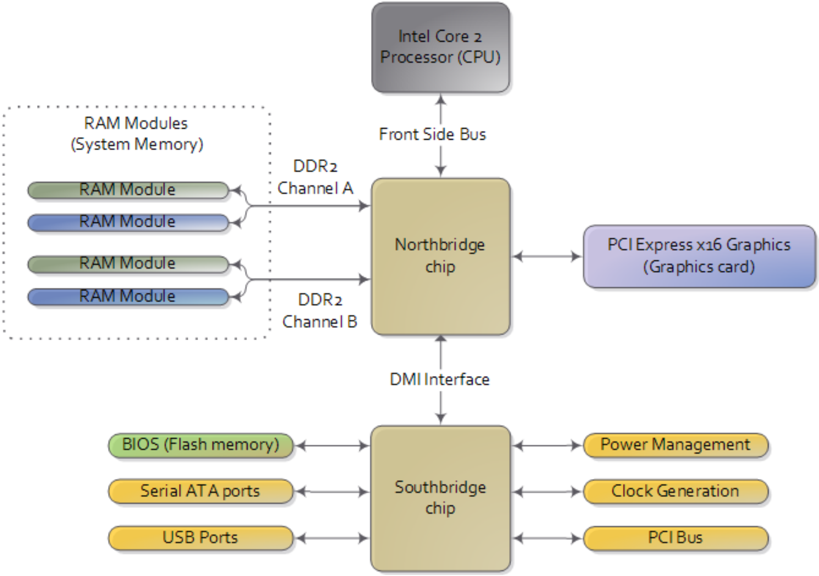
From Power Up To Bash Prompt

Wang Xiaolin

June 18, 2013

✉ wx672ster+os@gmail.com

Motherboard Chipsets And The Memory Map



Facts

- The CPU doesn't know what it's connected to
  - CPU test bench? network router? toaster? brain implant?
- The CPU talks to the outside world through its pins
  - some pins to transmit the physical memory address
  - other pins to transmit the values
- The CPU's gateway to the world is the front-side bus

Intel Core 2 QX6600

- 33 pins to transmit the physical memory address
  - so there are 2^{33} choices of memory locations
- 64 pins to send or receive data
  - so data path is 64-bit wide, or 8-byte chunks

This allows the CPU to physically address 64GB of memory ($2^{33} \times 8B$)

Some physical memory addresses are mapped away!

- only the addresses, not the spaces
- Memory holes
  - $640KB \sim 1MB$
  - /proc/iomem
- Memory-mapped I/O
  - BIOS ROM
  - video cards
  - PCI cards
  - ...

This is why 32-bit OSes have problems using 4G of RAM.

```
0xFFFFFFFF +--------------------+ 4GB
Reset vector |  JUMP to 0xF0000   |
0xFFFFFFF0 +--------------------+ 4GB - 16B
           |   Unaddressable    |
           | memory, real mode  |
           | is limited to 1MB. |
  0x100000 +--------------------+ 1MB
           |    System BIOS     |
   0xF0000 +--------------------+ 960KB
           |  Ext. System BIOS  |
   0xE0000 +--------------------+ 896KB
           |   Expansion Area   |
           | (maps ROMs for old |
           |  peripheral cards) |
   0xC0000 +--------------------+ 768KB
           | Legacy Video Card  |
           |   Memory Access    |
   0xA0000 +--------------------+ 640KB
           |   Accessible RAM   |
           |  (640KB is enough  |
           |  for anyone - old  |
           |     DOS area)      |
         0 +--------------------+ 0
```

What if you don't have 4G RAM?

the northbridge

1. receives a physical memory request
2. decides where to route it
   - to RAM? to video card? to ...?
   - decision made via the *memory address map*
     - /proc/iomem
     - it is built in setup()

The CPU modes

real mode: CPU can only address 1MB RAM

- 20-bit address, 1-byte data unit

32-bit protected mode: can address 4GB RAM

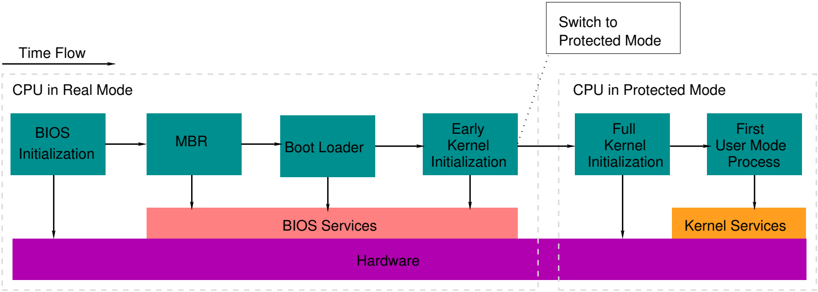
- 32-bit address, 1-byte data unit

64-bit protected mode: can address 64GB RAM (Intel Core 2 QX6600)

- 33 address pins, 8-byte data unit

```
$ grep 'address sizes' /proc/cpuinfo
```

Bootstrapping



1. bringing at least a portion of the OS into main memory, and
2. having the processor execute it
3. the initialization of kernel data structures
4. the creation of some user processes, and
5. the transfer of control to one of them

man 7 boot

Motherboard power up

1. initializes motherboard firmwares (chipset, etc.)
2. gets CPU running

Real mode

— CPU acts as a 1978 Intel 8086

- any code can write to any place in memory
- only 1MB of memory can be addressed
- registers are initialized
  - EIP has 0xFFFFFFF0, the reset vector
  - at the reset vector, there is a jump instruction, jumping to the *BIOS entry point* (0xF0000).

```
   0xFFFFFFFF +--------------------+ 4GB
Reset vector |  JUMP to 0xF0000   |
   0xFFFFFFF0 +--------------------+ 4GB - 16B
              |   Unaddressable    |
              | memory, real mode  |
              | is limited to 1MB. |
     0x100000 +--------------------+ 1MB
              |     System BIOS    |
      0xF0000 +--------------------+ 960KB
              |  Ext. System BIOS  |
      0xE0000 +--------------------+ 896KB
              |   Expansion Area   |
              | (maps ROMs for old |
              |  peripheral cards) |
      0xC0000 +--------------------+ 768KB
              | Legacy Video Card  |
              |   Memory Access    |
      0xA0000 +--------------------+ 640KB
              |   Accessible RAM   |
              |  (640KB is enough  |
              |  for anyone - old  |
              |     DOS area)      |
            0 +--------------------+ 0
```

BIOS

BIOS uses Real Mode addresses

- No GDT, LDT, or paging table is needed
  - the code that initializes the GDT, LDT, and paging tables must run in Real Mode
- Real mode address translation:

$$\text{segment number} \times 2^4 + \textit{offset}$$

e.g. to translate <FFFF:0001> into physical address:

$$FFFF \times 16 + 0001 = FFFF0 + 0001 = FFFF1$$

if: offset > 0xF (overflow)
then: address % 2^{20} (wrap around)

- only 80286 and later x86 CPUs can address up to:

$$FFFF0 + FFFF = 10FFEF$$

CPU starts executing BIOS code

1. POST
   - an ACPI-compliant BIOS builds several tables that describe the hardware devices present in the system
2. initializes hardwares
   - at the end of this phase, a table of installed PCI devices is displayed
3. find a boot device
4. load MBR into `0x7c00`
5. Jump to `0x7c00`
6. MBR moves itself away from `0x7c00` (fig 13)

```
|<-------------Master Boot Record (512 Bytes)------------>|
0          439        443    445               509       511
+----//-----+----------+------+------//---------+---------+
| code area | disk-sig | null | partition table | MBR-sig |
|    440    |     4    |  2   |     16x4=64     | 0xAA55  |
+----//-----+----------+------+------//---------+---------+
```

GRUB

1. GRUB stage 1 (in MBR) loads GRUB stage 2
2. stage 2 reads GRUB configuration file, and presents boot menu
3. loads the kernel image file into memory (fig 13)
   - can't be done in real mode, since it's bigger than 640KB
     - BIOS supports *unreal mode*
   - 1^{st} 512 bytes — INITSEG, 0x00090000
   - setup() — SETUPSEG, 0x00090200
   - load low — SYSSEG, 0x00010000
   - load high — 0x00100000
4. jumps to the kernel entry point
   - line 80 in 2.6.11/arch/i386/boot/setup.S

```
jmp trampoline
```

Memory At Bootup Time

The kernel image

- /boot/vmlinuz-x.x.x-x-x
- has been loaded into memory by the boot loader using the BIOS disk I/O services
- The image is split into two pieces:
  - a small part containing the real-mode kernel code is loaded below the 640K barrier
  - the bulk of the kernel, which runs in protected mode, is loaded after the first megabyte of memory

```
           ~                  ~
           |     load high    |
0x00100000 +------------------+ 1M
           ~     reserved     ~
0x000A0000 +------------------+ 640K
           ~                  ~
0x00098000 +------------------+
           | real mode stack  |
           ~                  ~
           |   2nd part of    |
           |      GRUB        |
0x00096C00 +------------------+
           | new location of  |
           |    MBR (512B)    |
0x00096A00 +------------------+
           ~                  ~
0x00090400 +------------------+ 577K
           |   setup sector   |
           |      (512B)      |
0x00090200 +------------------+ 576.5K
           |  1st 512 bytes   |
           | of kernel image  |
0x00090000 +------------------+ 576K
           ~                  ~
           |     load low     |
0x00010000 +------------------+ 64K
           ~                  ~
           +------------------+
           |    MBR (512B)    |
0x00007C00 +------------------+ 31K
           ~                  ~
           |    compressed    |
           |   kernel image   |
           | (if loaded low)  |
0x00001000 +------------------+ 4K
           ~                  ~
         0 +------------------+ 0
```

The setup() Function

boots and loads the executable image to $(0x9000 \ll 4)$ and jumps to $(0x9020 \ll 4)$

```
/*
*      setup.S         Copyright (C) 1991, 1992 Linus Torvalds
*
* setup.s is responsible for getting the system data from the BIOS,
* and putting them into the appropriate places in system memory.
* both setup.s and system has been loaded by the bootblock.
*
* This code asks the bios for memory/disk/other parameters, and
* puts them in a "safe" place: 0x90000-0x901FF, ie where the
* boot-block used to be. It is then up to the protected mode
* system to read them from there before the area is overwritten
* for buffer-blocks.
```

- 2.6.11/arch/i386/boot/setup.S
- Re-initialize all the hardware devices
- Sets the A20 pin (turn off *wrapping around*)
- Sets up a provisional IDT and a provisional GDT
- PE=1, PG=0 in cr0
- jump to startup_32()

setup() -> startup_32()

startup_32() for compressed kernel

- in arch/i386/boot/compressed/head.S
  - physically at

    0x00100000 — load high, or
    0x00001000 — load low
  - does some basic register initialization
  - decompress_kernel()
- the uncompressed kernel image has overwritten the compressed one starting at 1MB
- jump to the protected-mode kernel entry point at 1MB of RAM ($0\times10000 \ll 4$)
  - startup_32() for real kernel

startup_32() for real kernel

startup_32() in arch/i386/kernel/head.S

- Zeroes the kernel BSS for protected mode
- sets up the final GDT
- builds provisional kernel page tables so that paging can be turned on
- enables paging (cr3->PGDir; PG=1 in cr0)
- initializes a stack
- setup_idt() — creates the final interrupt descriptor table
- gdtr->GDT; idtr->IDT
- start_kernel()

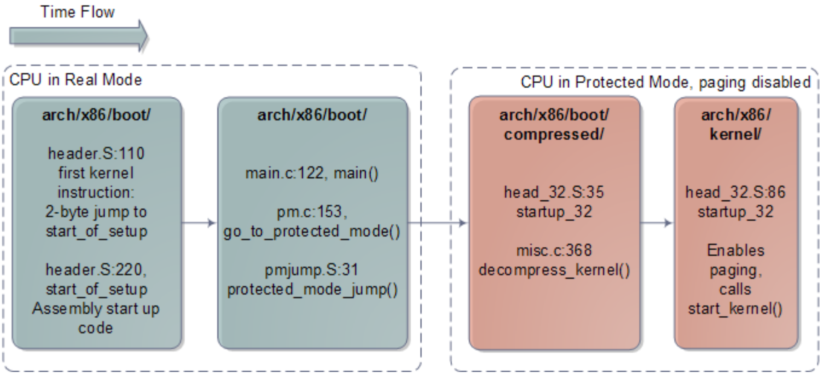
start_kernel() — a long list of calls to initialize various kernel subsystems and data structures

- sched_init() — scheduler
- build_all_zonelists() — memory zones
- page_alloc_init(), mem_init() — buddy system
- trap_init(), init_IRQ() — IDT
- time_init() — time keeping
- kmem_cache_init() — slab allocator
- calibrate_delay() — CPU clock
- kernel_thread() — The kernel thread for process 1
- login prompt

The Kernel Boot Process

```
+----------------+
|      BIOS      |
+----------------+
        |
        v
+----------------+
|   Bootloader   |
+----------------+
        |
        v
+----------------+
|    setup()     |
+----------------+ PE=1; PG=0
        |
        v
+----------------+
|  startup_32()  |
+----------------+
        |
        v
+----------------+
|  startup_32()  | PG=1
+----------------+
        |
        v
+----------------+
| start_kernel() |
+----------------+
```

The Kernel Boot Process (I)



The Kernel Boot Process (II)