From Power Up To Bash Prompt

Wang Xiaolin
wx672ster@gmail.com

June 18, 2013

Contents

Textbook:

- Appendix A, *System Startup*, [BC05]
- Appendix D, *System Startup*, [Mau08]
- Chapter 8, *Booting the Kernel*, [RFS05]

References

[Bar09] J. Bartlett. *Programming from the Ground Up*. University Press of Florida, 2009.

[BC05] D.P. Bovet and M. Cesatí. *Understanding The Linux Kernel*. 3rd ed. O'Reilly, 2005.

[Dua08a] Gustavo Duarte. *How Computers Boot up*. 2008.

[Dua08b] Gustavo Duarte. *Motherboard Chipsets and the Memory Map*. June 2008.

[Int86] Intel. *INTEL 80386 Programmer's Reference Manual*. 1986.

[Mau08] W. Mauerer. *Professional Linux Kernel Architecture*. John Wiley & Sons, 2008.

[NGC02] Abhishek Nayani, Mel Gorman, and Rodrigo S. de Castro. *Memory Management in Linux: Desktop Companion to the Linux Source Code*. Free book, 2002.

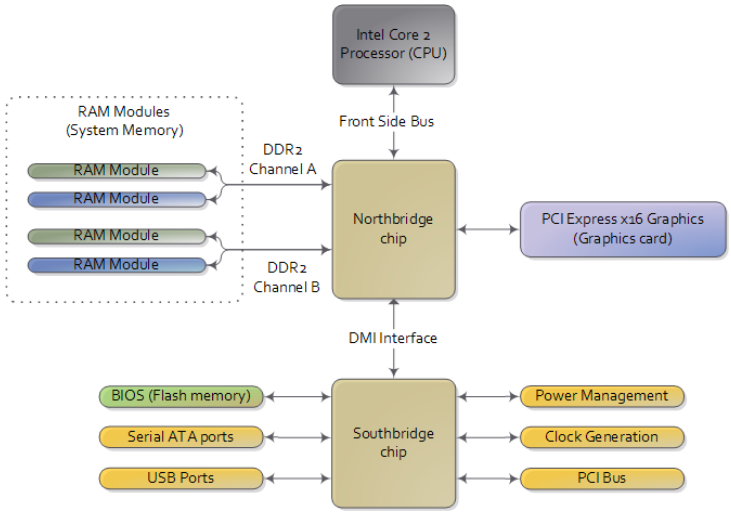
[RFS05] C.S. Rodriguez, G. Fischer, and S. Smolski. *The Linux kernel primer: a top-down approach for x86 and PowerPC architectures*. Prentice Hall Professional Technical Reference, 2005.

[sou08] Kernel source. *The Linux Boot Protocol*. 2008.

1 Motherboard Chipsets And The Memory Map

See [Dua08b]

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 ($2^{33} \times 8B$)

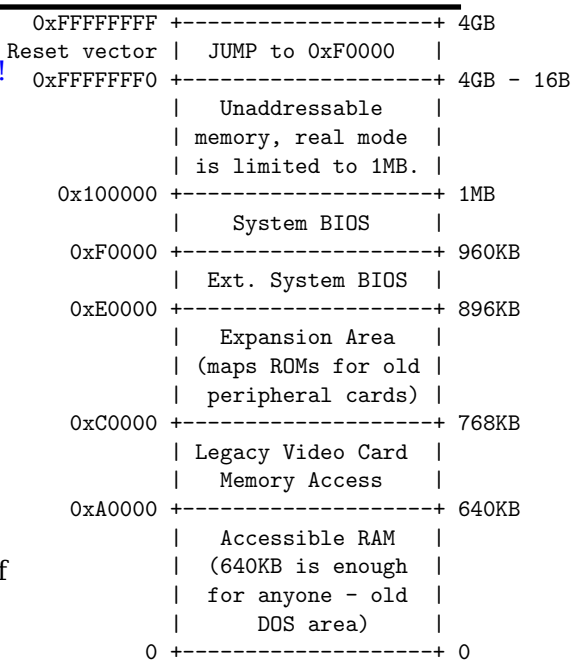
More info:

- Datasheet for Intel Core 2 Quad-Core Q6000 Sequence

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.



What if you don't have 4G RAM?

- OSDev: Memory Map (x86)

---

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

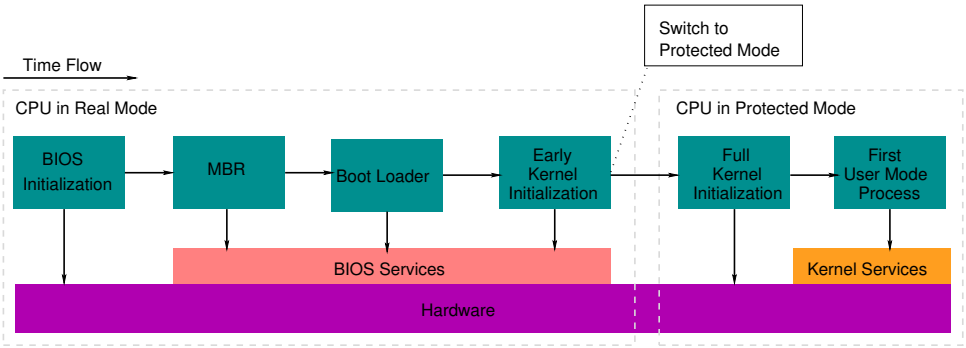
- In the comments of Motherboard Chipsets and the Memory Map

  > The *physical* stuff is determined by hard-core limits: the actual metal pins that stick out of the processor. It is *those* pins that limit the CPU to 64 gigabytes. That is completely independent of the operating system or even the mode (real-mode, 32-bit protected, 64-bit) the CPU is running in. It's a physical limit. That is the limit for which that little multiplication is done. There are 33 metal pins to transmit an address and 8 metal pins to send and receive data. So $2^{33} \times 2^3 = 2^{36} = 64GB$. The "unit" of transfer in this case is 8 bytes, that's the smallest chunk of data the CPU can address on the physical bus. In actuality, the CPU usually works in terms of cache lines, which hold 64 bytes in the Core 2s. Due to performance, the CPU reads a whole cache line at a time. So if a program reads one byte, the CPU actually reads 64 bytes and stores them in the cache. The 4-gb limit is logical, not physical. It happens because the registers and instructions in the CPU are limited to 32 bits *when it's running in 32-bit mode*, which *does* depend on the OS. Programs need to be able to address individual bytes in memory, so the "unit" of addressing is 1 byte. So *that* equation becomes $2^{32}\ addresses \times 1\ byte\ chunks = 2^{32}\ bytes$, or 4 GB total addressing.

2 How Computers Boot Up

See [Dua08a]

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

2.1 Motherboard power up

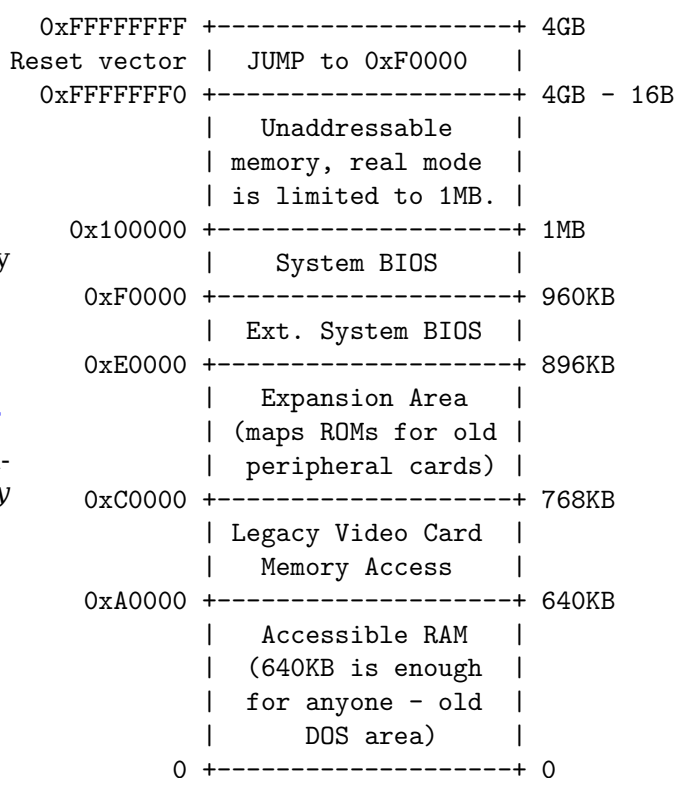
Motherboard power up

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).



More info:

- stackoverflow: How is the BIOS rom mapped into address space on PC?
- The PC Guide: System Boot Sequence
- stackoverflow: Do normal x86 or AMD PCs run startup/BIOS code directly from ROM, or do they copy it first to RAM?
- Wikipedia: Reset vector
- What Happens When A CPU Starts
- FreeBSD: BIOS POST
- FreeBSD: Bootstrapping and Kernel Initialization

2.2 BIOS

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:

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

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)

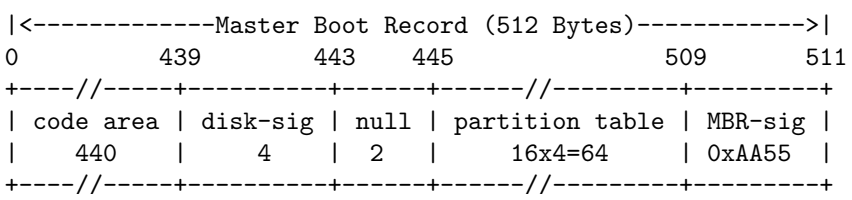
- only 80286 and later x86 CPUs can address up to:

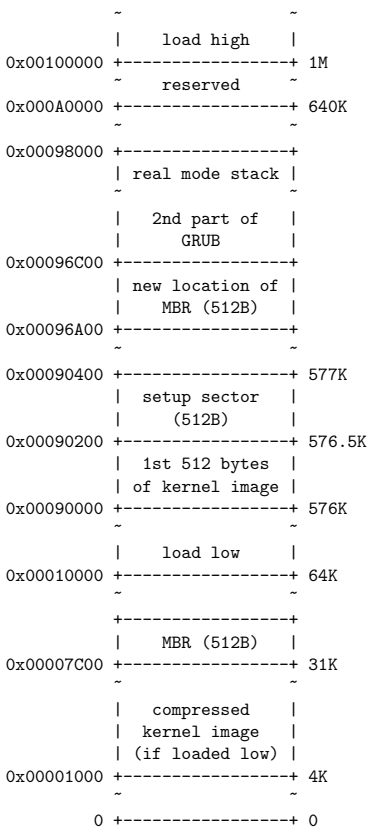
$$FFFF0 + FFFF = 10FFEF$$

- Wikipedia: Real mode
- OSDev: Real Mode

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)





- The MBR includes a small boot loader, which is loaded into RAM starting from address 0x00007c00 by the BIOS. (Why 0x7c00?)
- This small program moves itself to the address 0x00096a00, sets up the Real Mode stack (ranging from 0x00098000 to 0x000969ff), loads the second part of the boot loader into RAM starting from address 0x00096c00, and jumps into it.

  Why move? because the boot loader may copy the boot sector of a boot partition into RAM (0x7c00) and execute it

2.3 The Boot Loader

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*
   - 1st 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

- GRUB bootloader - Full tutorial
- jmp trampoline was used in 2.6.11 for calling start_of_setup. In newer kernels, a *2-byte jump* is used instead.
  - `http://lxr.linux.no/linux+v2.6.34/arch/x86/boot/header.S#L112`
  - Using SHORT (Two-byte) Relative Jump Instructions
  - 2-byte jump in header.S

More about *unreal mode*:

- (OSDev: Descriptor cache) Unreal Mode is a 'mode' where the processor runs in real mode while the segment limit does not equal 64KB (in most cases, its 4GB). Since real mode doesn't update the limit field (of the cache), this state persists across segment register loads. Entering this mode is achieved easily by entering protected mode (where the limit can be changed), load the desired limit into the descriptor cache, then switch back to real mode.
- (A great post in OSDev forum: Unreal mode that deserves a detailed look) The benefits of unreal mode are quite well known: access to the 32-bit address space while simultaneously being able to call BIOS and real mode programs.
- OSDev: Segment Registers: Real mode vs. Protected mode covers *unreal mode*, *NULL selector*, *mode switching*
- Wikipedia: Unreal mode
- OSDev: Unreal mode

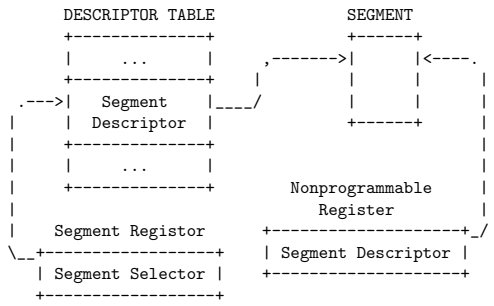


Figure 1: Descriptor cache register

More about *bootloader*:

- Linux Boot Loaders Compared
- How Boot Loader Works?

1. display a "Loading" message
2. load an initial portion of the kernel image from disk:
   - the first 512 bytes of the kernel image are put in RAM at address 0x00090000 (576K, INITSEG)
     - hd -n512 /boot/vmlinuz-3.2.0-1-amd64
     - /usr/src/linux/arch/i386/boot/bootsect.S
     - it was a floppy boot loader, and no longer valid since 2.6

- nowadays to make a bootable floppy, you have to use a bootloader as you do with a hard disk

- the code of the setup() function (see below) is put in RAM starting from address 0x00090200 ($576K + 512$, SETUPSEG)

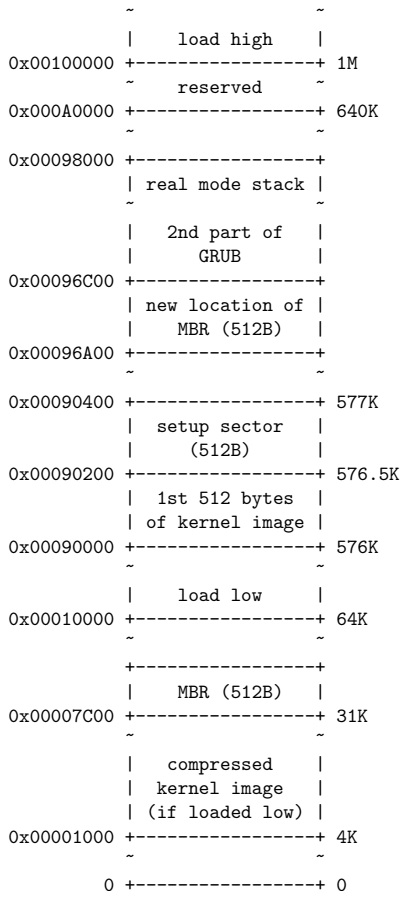
3. load the rest of the kernel image from disk and puts the image in RAM starting from either low address 0x00010000 (64K, SYSSEG) (for small kernel images ($< 512K$) compiled with make zImage) or high address 0x00100000 (1M)(for big kernel images ($> 512K$) compiled with make bzImage).

   ISA hole: Physical addresses ranging from 0x000a0000 (640K) to 0x000fffff ($1M - 1$) are usually reserved to BIOS routines and to map the internal memory of ISA graphics cards

4. Jumps to the setup() code. (/usr/src/linux/arch/i386/boot/setup.S)

BIOS interrupt call `int $0x13` is oftenly seen in 2.6.11/arch/i386/boot/setup.S. INT 13H, INT 13h, or INT 19 is shorthand for BIOS interrupt call 13_{hex}, the 20th interrupt vector in an x86-based computer system. The BIOS typically sets up a real mode interrupt handler at this vector that provides sector-based hard disk and floppy disk read and write services using cylinder-head-sector (CHS) addressing.

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

- More info about boot-time memory arrangement, see [sou08].

3 The Kernel Boot process

3.1 setup()

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()

- *Kernel attributes* are stored at the end of the boot block (1^{st} sector). (line 90 in bootsect.S)
- From which point onwards the kernel execution starts?
- Insight into GNU/Linux boot process
- Linux Boot Process in a nutshell
- The setup() function:
  1. Builds system's physical memory map
     - find the amount of memory present in the system (sec 1.1 in [NGC02], arch/i386/boot/setup.S (2.4.19), line 289-389)
  2. Sets the keyboard repeat delay and rate
  3. Initializes the video adapter card
  4. Reinitializes the disk controller and determines the hard disk parameters
  5. Checks for an IBM Micro Channel bus (MCA)
  6. Checks for a PS/2 pointing device (bus mouse)
  7. Checks for Advanced Power Management (APM) BIOS support
  8. If the BIOS supports the Enhanced Disk Drive Services (EDD), builds a table in RAM describing the hard disks available in the system
  9. If the kernel image was loaded low in RAM (at physical address 0x00010000), the function moves it to physical address `0x00001000` (was used by boot loader).

Why? (Sec A.3, *Middle Ages: the setup() Function*, [BC05]) This step is necessary because to be able to store the kernel image on a floppy disk and to reduce the booting time, the kernel image stored on disk is compressed, and the decompression routine needs some free space to use as a temporary buffer following the kernel image in RAM.

- BOOTSEG = 0x07C0. This is 27K above 0x1000. It was too small to hold the kernel image. After boot loader is done, BOOTSEG (0x7C00) is free. So kernel image can be stuffed here.

Memory layout (Insight into GNU/Linux Boot Process)

uncompressed image: ... Later, all the kernel is moved from 0×10000 (64K) to 0×1000 (4K). This move overwrites BIOS data stored in RAM, so BIOS calls can no longer be performed. We don't care because linux dosen't use BIOS to acces the hardware. The first physical page is not touched because it is the so-called "zero-page", used in handling virtual memory. At this point, setup.S enters protected mode and jumps to 0×1000, where the kernel lives. All the available memory can be accessed now, and the system can begin to run.

The steps just described were once the whole story of booting when the kernel was small enough to fit in half a megabyte of memory — the address range between 0×10000 and 0×90000. As features were added to the system, the kernel became larger than half a megabyte and could no longer be moved to 0×1000. Thus, code at 0×1000 is no longer the Linux kernel, instead the "gunzip" part of the gzip program resides at that address.

Compressed image [zimage]: When the kernel is moved to 0×1000 (4K), head.S in the compressed directory is sitting at this address. It's in charge of gunzipping the kernel, this done by a function decompress_kernel(), defined in compressed/misc.c, which in turns calls inflate() which writes its output starting at address 0×100000 (1MB). High memory can now be accessed, because setup.S han take us to the protected mode now. After decompression, head.S jumps to the actual beginning of the kernel. The relevant code is in ../kernel/head.S. head.S (i.e., the code found at 0×100000) can complete processor initialization and call start_kernel().

The boot steps shown above rely on the assumption that the compressed kernel can fit in half a megabyte of space. While this is true most of the time, a system stuffed with device drivers might not fit into this space. For example, kernels used in installation disks can easily outgrow the available space. To solve this problem problem bzImage kernel images were introduced.

Big Compressed Image [bzImage]: This kind of kernel image boots similarly to zImage, with a few changes.

When the system is loaded at 0×10000 (64K) a special helper routine is called which does some special BIOS calls to move the kernel to 0×100000 (1Mb). setup.S doesn't move the system back to 0×1000 (4K) but, after entering protected mode, jumps instead directly to address 0×100000 (1MB) where data has been moved by the BIOS in the previous step. The decompresser found at 1MB writes the uncompressed kernel image into low memory until it is exhausted, and then into high memory after the compressed image. The two pieces are then reassembled to the address 0×100000 (1MB). Several memory moves are needed to perform the task the address 0×100000 (1MB). Several memory moves are needed to perform the task correctly.

```
1     code32_start:               # here loaders can put a different
2                                 # start address for 32-bit code.
3     #ifndef __BIG_KERNEL__
4             .long     0x1000    #   0x1000 = default for zImage
5     #else
6             .long     0x100000  # 0x100000 = default for big kernel
7     #endif
```

The default value of code32 is __BOOT_CS:0x1000 (__BOOT_CS = 16). (Line 855-857)

```
1       code32: .long 0x1000    # will be set to 0x100000 for big kernels
2       .word __BOOT_CS
```

It will be changed to __BOOT_CS:0x100000. (Line 594-595)

```
1       movl %cs:code32_start, %eax
2       movl %eax, %cs:code32
```

%cs:code32_start ⇒ %cs:code32 = 0x10:0x100000 (for load high, i.e. bzImage)

10. Sets the A20 pin located on the 8042 keyboard controller (for switching to pmode)
11. Sets up a provisional Interrupt Descriptor Table (IDT) and a provisional Global Descriptor Table (GDT).
    - line 792 in setup.S
    - line 83-89 in include/asm-i386/segment.h

    The provisional GDT is created with 2 useful entries, each covering the whole 4GB address space [NGC02]. The code that loads the GDT is:

```
1       xorl %eax, %eax # Compute gdt_base
2       movw %ds, %ax   # (Convert %ds:gdt to a linear ptr)
3       shll $4, %eax
4       addl $gdt, %eax
5       movl %eax, (gdt_48+2)
6       lgdt gdt_48     # load gdt with whatever is appropriate
```

    - %ds = %cs = SETUPSEG = 0x9020
    - $gdt — beginning address of the GDT table (somewhere offsetting in %ds). Its actual value will be determined at assemble time by the assembler (p24 in [Bar09])
    - gdt_48: a label in setup.S (line 1006). gdt_48+2 will be filled with the *gdt base* computed above.
    - lgdt: loads the value in gdt_48 into GDTR
    - gdt_48 = limit,base ⇒ GDTR
      * limit = gdt_end - gdt - 1 = 31 (16 bits)
      * base = %ds ≪ 4 + gdt (32 bits)

```
1  gdt_48:
2       .word      gdt_end - gdt - 1   # gdt limit
3       .word      0, 0                # gdt base (filled in later)
4
5  gdt:
6       .fill GDT_ENTRY_BOOT_CS,8,0
7
8       .word      0xFFFF   # 4Gb - (0x100000*0x1000 = 4Gb)
9       .word      0        # base address = 0
10      .word      0x9A00   # code read/exec
11      .word      0x00CF   # granularity = 4096, 386
12                          #  (+5th nibble of limit)
13
14      .word      0xFFFF   # 4Gb - (0x100000*0x1000 = 4Gb)
15      .word      0        # base address = 0
16      .word      0x9200   # data read/write
17      .word      0x00CF   # granularity = 4096, 386
18                          #  (+5th nibble of limit)
19 gdt_end:
```

    - gdt: the provisional GDT has 4 entries.
      * The 1st and 2nd entries are initialized to 0[1] (line 983), as required by Intel.

        .fill GDT_ENTRY_BOOT_CS,8,0

      * 3rd is __BOOT_CS

[1] .fill REPEAT,SIZE,VALUE

* 4th is __BOOT_DS

(Linux HOWTO: Prepare to move to protected mode) Calculate the linear base address of the kernel GDT (table) and load the GDT pointer register with its base address and limit.

Figure 2: Provisional GDT in RAM

* This early kernel GDT describes kernel code as 4 GB, with base address 0, code/readable/executable, with granularity of 4 KB.

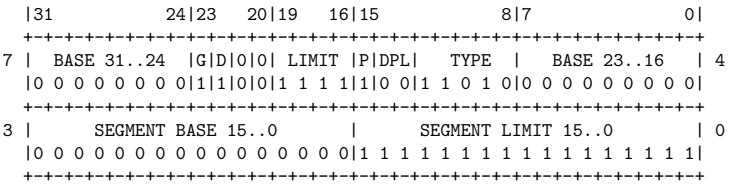


Figure 3: Code segment descriptor value: `0x00CF9A000000FFFF`

* The kernel data segment is described as 4 GB, with base address 0, data/readable/writable, with granularity of 4 KB.

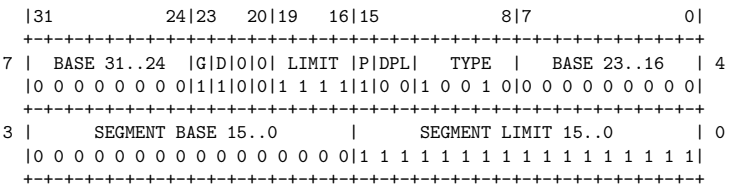


Figure 4: Data segment descriptor value: `0x00CF92000000FFFF`

12. Resets the floating-point unit (FPU), if any.
13. Reprograms the Programmable Interrupt Controllers (PIC) to mask all interrupts, except IRQ2 which is the cascading interrupt between the two PICs.
14. Switches the CPU from Real Mode to Protected Mode by setting the PE bit in the cr0 status register. The PG bit in the cr0 register is cleared, so paging is still disabled. (line 832-833)

```
1  movw $1, %ax # protected mode (PE) bit
2  lmsw %ax     # This is it!
```

- lmsw — Load Machine Status Word (part of cr0)
- in later kernel (e.g. 2.6.34) the switching code is like this (line 39-41 in pmjump.S):

```
1  movl %cr0, %edx
2  orb $X86_CR0_PE, %dl # Protected mode
3  movl %edx, %cr0
```

line 29 in processor-flags.h:

#define X86_CR0_PE 0x00000001

NOTE: We are now in 32-bit protected mode. From now on, the address translation will be done by looking up the GDT table.

15. Jumps to the startup_32() assembly language function. (line 854)

```
        .byte 0x66, 0xea    # prefix + jmpi-opcode
code32: .long   0x1000      # will be set to 0x100000
                            # for big kernels
        .word   __BOOT_CS
```

- jmpi 0x100000,__BOOT_CS (far jump)
  * jump to 0x10:0x100000 (segment number: 0x10; offset: 0x100000.)
- (Insight into GNU/Linux boot process) At the end of the initial assembly code in arch/i386/boot/setup.S a jump to offset 0x100000 in segment KERNEL_CS is called. This is where the version of startup_32() found in arch/i386/boot/compressed/head.S. But the jump is a little tricky, as we haven't yet reloaded the CS register, the default size of the target offset still is 16 bit. However, using an operand prefix (0×66), the CPU will properly take our 48 bit far pointer [.byte 0x66, 0xea].

3.2 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

———— ℹ ————

- Initializing registers (line 31 in head.S)

```
startup_32:
        cld
        cli
        movl $(__BOOT_DS),%eax
        movl %eax,%ds
        movl %eax,%es
        movl %eax,%fs
        movl %eax,%gs

        lss stack_start,%esp
```

  - cld: clear direction flag

- cli: clear interrupt flag
- lss stack_start,%esp: load %ss and %esp pair (%ss:%esp) in a single instruction using the value stored in stack_start. (line 40 in head.S)
  * lss — read a full pointer from memory and store it in the selected segment register:register pair. (p332, [Int86])
- stack_start (line 296-303 in misc.c)

```
1       struct {
2         long * a;
3         short b;
4       } stack_start = { & user_stack [STACK_SIZE] , __BOOT_DS };
```

- STACK_SIZE = 4096
- __BOOT_DS = 24 (segment selector) ⇒ %ss
  * the 4^{th} (2^{nd} non-zero) entry in provisional GDT (sec 1.2 in [NGC02]). Each entry is 8 bytes.
- &user_stack[4096] ⇒ %esp

- Clear BSS (line 55 in head.S)

```
1 xorl %eax,%eax
2 movl $_edata,%edi
3 movl $_end,%ecx
4 subl %edi,%ecx
5 cld
6 rep
7 stosb
```

- stosb copies the value in AL into the location pointed to by ES:DI. DI is then incremented (if the direction flag is cleared) or decremented (if the direction flag is set), in preparation for storing AL in the next location.
- rep — repeat
- (Stackoverflow: What does rep stos do?) For ecx repetitions, stores the contents of eax into where edi points to, incrementing or decrementing edi (depending on the direction flag) by 4 bytes each time. Normally, this is used for a memset-type operation.

  Usually, that instruction is simply written rep stosd. Experienced assembly coders know all the details mentioned above just by seeing that. :-)

  ETA for completeness (thanks PhiS): Each iteration, ecx is decremented by 1, and the loop stops when it reaches zero. For stos, the only thing you will observe is that ecx is cleared at the end.

---

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()

- Sets up the final GDT (line 63, 303, 448-450, 459-463, 470-EOF in head.S)

```
1         lgdt boot_gdt_descr - __PAGE_OFFSET
2
3         ...
4         lgdt cpu_gdt_descr
5         ...
6
7 boot_gdt_descr:
8         .word __BOOT_DS+7                    # limit (31, end of DS)
9         .long boot_gdt_table - __PAGE_OFFSET  # base location
10
11 ENTRY(boot_gdt_table)
12        .fill GDT_ENTRY_BOOT_CS,8,0
13        .quad 0x00cf9a000000ffff    # kernel 4GB code at 0x00000000
14        .quad 0x00cf92000000ffff    # kernel 4GB data at 0x00000000
15
16 cpu_gdt_descr:
17        .word GDT_ENTRIES*8-1
18        .long cpu_gdt_table
19
20        .fill NR_CPUS-1,8,0             # space for the other GDT descriptors
21
22 ENTRY(cpu_gdt_table)
23        ...
24        # Entry 12-15
25        .quad 0x00cf9a000000ffff    # 0x60 kernel 4GB code at 0x00000000
26        .quad 0x00cf92000000ffff    # 0x68 kernel 4GB data at 0x00000000
27        .quad 0x00cffa000000ffff    # 0x73 user 4GB code at 0x00000000
28        .quad 0x00cff2000000ffff    # 0x7b user 4GB data at 0x00000000
29        ...
```

  - .fill REPEAT, SIZE, VALUE
    * .fill 2,8,0 fills the 1st and 2nd entry with 0
  - .quad 8-byte datas (the 3rd and 4th entry)
  - The addresses of boot_gdt_table and cpu_gdt_table will be assigned by assembler at compile time.

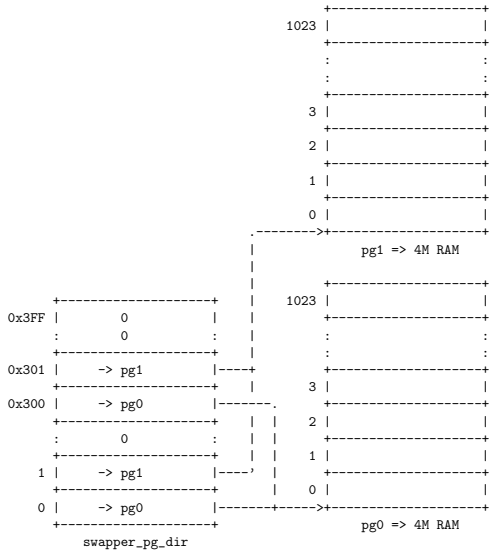
- Zeroes the kernel BSS (line 74-79 in arch/i386/kernel/head.S)

```
1         xorl %eax,%eax
2         movl $__bss_start - __PAGE_OFFSET,%edi
3         movl $__bss_stop - __PAGE_OFFSET,%ecx
4         subl %edi,%ecx
5         shrl $2,%ecx
6         rep ; stosl
```

  - subl %edi, %ecx — get BSS size, put into %ecx
  - shrl $2,%ecx — $\frac{\%ecx}{4}$, get number of 4-byte chunks (repeat times)
  - stosl/stosd — Store EAX in dword ES:EDI, update EDI

- Initialize page tables (Sec 2.5.5, *Kernel Page Tables*, [BC05])
  - In the first phase, the kernel creates a limited address space including the kernel's code and data segments, the initial Page Tables, and 128 KB for some dynamic data structures. This minimal address space is just large enough to install the kernel in RAM and to initialize its core data structures.

- The provisional Page Global Directory is contained in the swapper_pg_dir variable. swapper_pg_dir is at the beginning of BSS (uninitialized data area) because BSS is no longer used after system start up.
- The provisional Page Tables are stored starting from pg0, right after the end of the kernel's uninitialized data segments (_end).
- For the sake of simplicity, let's assume that the kernel's segments, the provisional Page Tables, and the 128 KB memory area fit in the first 8 MB of RAM. In order to map 8 MB of RAM, two Page Tables are required.
- The objective of this first phase of paging is to allow these 8 MB of RAM to be easily addressed both in real mode and protected mode.



Therefore, the kernel must create a mapping from both the linear addresses 0x00000000 through 0x007fffff (8M) and the linear addresses 0xc0000000 through 0xc07fffff (8M) into the physical addresses 0x00000000 through 0x007fffff. In other words, the kernel during its first phase of initialization can address the first 8 MB of RAM by either linear addresses identical to the physical ones or 8 MB worth of linear addresses, starting from 0xc0000000.

Why? (Sec 1.3.2, *Provisional Kernel Page Tables*, [NGC02])

* All pointers in the compiled kernel refer to addresses $> PAGE\_OFFSET$. That is, the kernel is linked under the assumption that its base address will be start_text (I think; I don't have the code on hand at the moment), which is defined to be $PAGE\_OFFSET + (some\ small\ constant,\ call\ it\ C)$.
* All the kernel bootstrap code (mostly real mode code) is linked assuming that its base address is $0 + C$.

head.S is part of the bootstrap code. It's running in protected mode with paging turned off, so all addresses are physical. In particular, the instruction pointer is fetching instructions based on physical address. The instruction that turns on paging (movl %eax, %cr0) is located, say, at some physical address A.

As soon as we set the paging bit in cr0, paging is enabled, and starting at the very next instruction, all addressing, including instruction fetches, pass through the address translation mechanism (page tables). IOW, all address are henceforth virtual. That means that

1. We must have valid page tables, and
2. Those tables must properly map the instruction pointer to the next instruction to be executed.

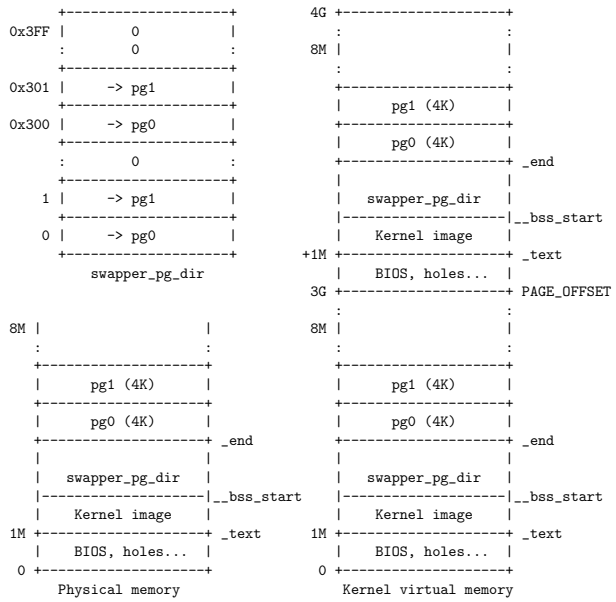
That next instruction is physically located at address A+4 (the address immediately after the ”movl %eax, %cr0” instruction), but from the point of view of all the kernel code — which has been linked at PAGE_OFFSET — that instruction is located at virtual address PAGE_OFFSET+(A+4). Turning on paging, however, does not magically change the value of EIP [2]. The CPU fetches the next instruction from ***virtual*** address A+4; that instruction is the beginning of a short sequence that effectively relocates the instruction pointer to point to the code at PAGE_OFFSET+A+(something).
But since the CPU is, for those few instructions, fetching instructions based on physical addresses ***but having those instructions pass through address translation***, we must ensure that both the physical addresses and the virtual addresses are :

1. Valid virtual addresses, and
2. Point to the same code.

That means that at the very least, the initial page tables must

1. map virtual address PAGE_OFFSET+(A+4) to physical address (A+4), and must
2. map virtual address A+4 to physical address A+4.

This dual mapping for the first 8MB of physical RAM is exactly what the initial page tables accomplish. The 8MB initally mapped is more or less arbitrary. It’s certain that no bootable kernel will be greater than 8MB in size. The identity mapping is discarded when the MM system gets initialized.



line 91-111 in head.S

```
page_pde_offset = (__PAGE_OFFSET >> 20);

        movl $(pg0 - __PAGE_OFFSET), %edi
        movl $(swapper_pg_dir - __PAGE_OFFSET), %edx
        movl $0x007, %eax               # 0x007 = PRESENT+RW+USER
10:
        leal 0x007(%edi),%ecx           # Create PDE entry
        movl %ecx,(%edx)                # Store identity PDE entry
        movl %ecx,page_pde_offset(%edx) # Store kernel PDE entry
        addl $4,%edx
```

[2] The value of EIP is still physically A+4, not PAGE_OFFSET+(A+4) yet. But since paging is just enabled, CPU could pass A+4 through address translation.

```
11          movl $1024, %ecx
12      11:
13          stosl
14          addl $0x1000,%eax
15          loop 11b
16          # End condition: we must map up to and including INIT_MAP_BEYOND_END
17          # bytes beyond the end of our own page tables; the +0x007 is the attribute bits
18          leal (INIT_MAP_BEYOND_END+0x007)(%edi),%ebp
19          cmpl %ebp,%eax
20          jb 10b
21          movl %edi,(init_pg_tables_end - __PAGE_OFFSET)
```

```
 31                                  12 11                    0
+--------------------------------------+-------+---+-+-+---+-+-+-+
|                                      |       |   | | |   |U|R| |
|      PAGE FRAME ADDRESS 31..12       | AVAIL |0 0|D|A|0 0|/|/|P|
|                                      |       |   | | |   |S|W| |
+--------------------------------------+-------+---+-+-+---+-+-+-+
        P      - PRESENT
        R/W    - READ/WRITE
        U/S    - USER/SUPERVISOR
        A      - ACCESSED
        D      - DIRTY
        AVAIL  - AVAILABLE FOR SYSTEMS PROGRAMMER USE

        NOTE: 0 INDICATES INTEL RESERVED. DO NOT DEFINE.
```

- `page_pde_offset = (__PAGE_OFFSET >> 20); # 0xC00, the 3K point.`

  The PGDir is one page (4K) in size. It's divided into two parts:

  1. first 3K (768 entries) for user mode
  2. last 1K (256 entries) for kernel mode

- $(pg0 - __PAGE_OFFSET) yields the physical address of pg0 since here it is a linear address. Same case for $(swapper_pg_dir - __PAGE_OFFSET).
  * swapper_pg_dir starts at the beginning of BSS
  * pg0 starts at _end
- Registers:

  %edi address of each page table entry, i.e. pg0[0]..pg0[1023], pg1[0]..pg1[1023].

  %edx address of swapper_pg_dir[0], and then to swapper_pg_dir[1].

  %ecx has two uses
  1. contents of swapper_pg_dir[0], swapper_pg_dir[1], swapper_pg_dir[768], swapper_pg_dir[769].
  2. loop counter (1024 -> 0)

  %eax 7, 4k+7, 8k+7 ... 8M-4k+7 for 2k page table entries in pg0 and pg1 respectively.

  %ebp = 128k + 7 + &pg0[1023] in the first round of loop. Its value cannot be determined at coding time, because the address of pg0 is not known until compile/link time.

- stosl: stores the contents of EAX at the address pointed by EDI, and increments EDI. Equivalent to:
  1. movl %eax, (%edi)
  2. addl $4, %edi
- cmpl, jb: if %eax < %ebp, jump to 10;
  * jb: (Wikipedia: x86 assembly language) jump on below/less than, unsigned
  * At the end of the 1st round of loop, the value of %eax is 4M-4k+7, while the value of %ebp depends on the address of pg0.
    If the kernel image is small enough (e.g. a `zImage`), pg0 could be low enough to let the 128KB covered without a pg1.

- INIT_MAP_BEYOND_END: 128KB, used as a bitmap covering all pages. For 1M pages (4GB RAM), we need 1M bits (128K bytes). (`http://kerneldiy.com/blog/?p=201`)

Equivalent pseudo C code:

```
/*
 * Provisional PGDir and page tables setup
 *
 * for mapping two linear address ranges to the same physical address range
 *
 *  + Linear address ranges:
 *              -   User mode: i × 4M ∼ (i + 1) × 4M − 1
 *              - Kernel mode: 3G + i × 4M ∼ 3G + (i + 1) × 4M − 1
 *  + Physical address range: i × 4M ∼ (i + 1) × 4M − 1
 */
typedef unsigned int PTE;
PTE *pg = pg0;     /* physical address of pg0 */
PTE pte = 0x007;   /* 0x007 = PRESENT+RW+USER */
for(i=0;;i++){
  swapper_pg_dir[i] = pg + 0x007;         /* store identity PDE entry */
  swapper_pg_dir[i+page_pde_offset] = pg + 0x007; /* kernel PDE entry */
  for(j=0;j<1024;j++){                /* populating one page table */
    pg[i*1024 + j] = pte;             /* fill up one page table entry */
    pte += 0x1000;                    /* next 4k */
  }
  if(pte >= ((char*)pg + i*1024 + j)*4 + 0x007 + INIT_MAP_BEYOND_END)
    {
      init_pg_tables_end = pg + i*0x1000 + j;
      break;
    }
  }
```

- Linux HOWTO: Enable paging (line 186-194 in head.S)

```
# Enable paging
movl $swapper_pg_dir - __PAGE_OFFSET, %eax
movl %eax, %cr3          # set the page table pointer..
movl %cr0, %eax
 orl $0x80000000, %eax
movl %eax, %cr0          # ..and set paging (PG) bit
```

3.3 start_kernel()

---

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

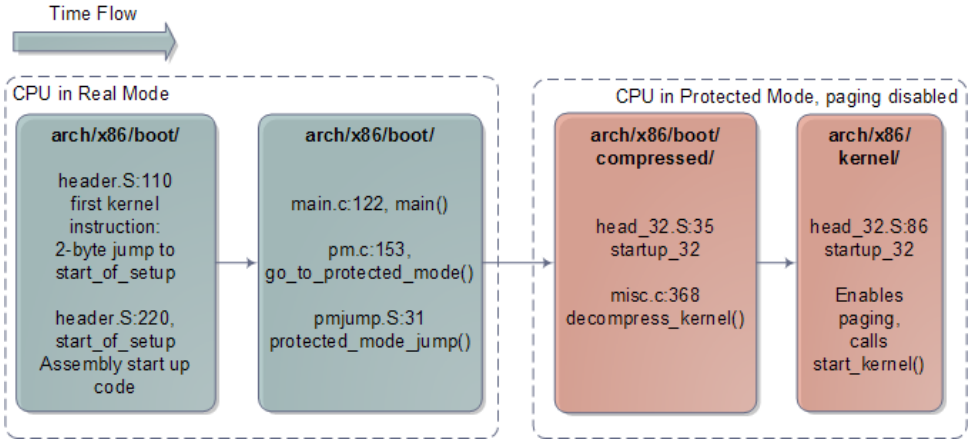
- sched_init() — scheduler
- build_all_zonelists() — memory zones
- page_alloc_init(), mem_init() — buddy system
- trap_init(), init_IRQ() — IDT
- time_init() — time keeping
- kmem_cache_init() — slab allocator
- calibrate_delay() — CPU clock
- kernel_thread() — The kernel thread for process 1

- login prompt

The Kernel Boot Process

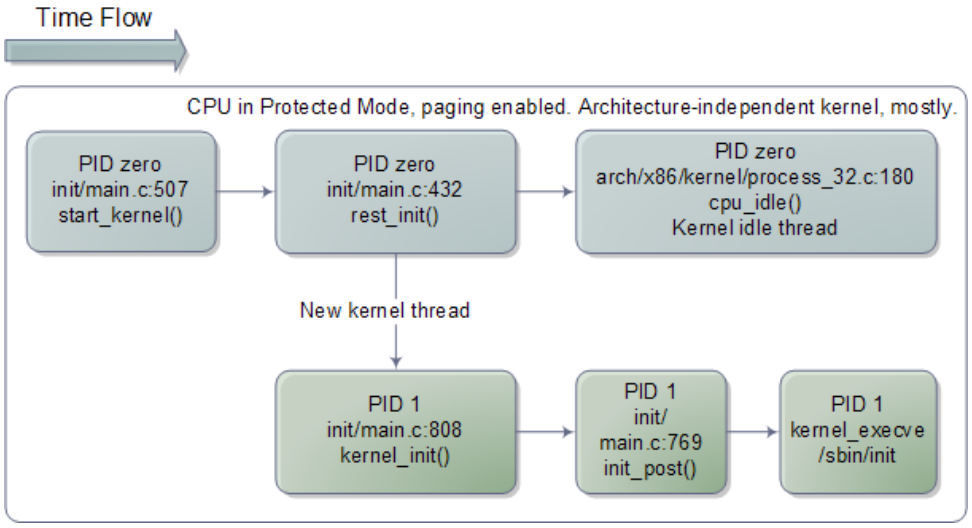
```
+----------------+
|     BIOS       |
+----------------+
        |
        v
+----------------+
|  Bootloader    |
+----------------+
        |
        v
+----------------+
|    setup()     |
+----------------+ PE=1; PG=0
        |
        v
+----------------+
|  startup_32()  |
+----------------+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

The Kernel Boot Process (I)



- This picture is not fully compatible with 2.6.11.

The Kernel Boot Process (II)



- This picture is not fully compatible with 2.6.11.