Linux 2.4.x Initialization for IA–32 HOWTO

Table of Contents

Table of Contents

Table of Contents

Linux 2.4.x Initialization for IA−32 HOWTO

Randy Dunlap, `rddunlap@ieee.org`

v1.0, 2001−05−17

This document contains a description of the Linux 2.4 kernel initialization sequence on IA−32 processors.

---

6. Glossary

7. References

---

1. Introduction

Portions of this text come from comments in the kernel source files (obviously). I have added annotations in many places. I hope that this will be useful to kernel developers −− either new ones or experienced ones who need more of this type of information. However, if there's not enough detail here for you, "Use the Source."

1.1 Overview

This description is organized as a brief overview which lists the sections that are described later in more detail.

The description is in three main sections. The first section covers early kernel initialization on IA−32 (but only after your boot loader of choice and other intermediate loaders have run; i.e., this description does not cover loading the kernel). This section is based on the code in "linux/arch/i386/boot/setup.S" and "linux/arch/i386/boot/video.S".

The second major section covers Linux initialization that is x86− (or i386− or IA−32−) specific. This section is based on the source files "linux/arch/i386/kernel/head.S" and "linux/arch/i386/kernel/setup.c".

The third major section covers Linux initialization that is architecture−independent. This section is based on the flow in the source file "linux/init/main.c".

See the References section for other valuable documents about booting, loading, and initialization.

1.2 This document

This document describes Linux 2.4.x initialization on IA−32 (or i386 or x86) processors −− after one or more kernel boot loaders (if any) have done their job.

You can format it using the commands (for example):

```
% sgml2txt ia32_init_240.sgml
```

or

```
% sgml2html ia32_init_240.sgml
```

This will produce plain ASCII or HTML files respectively. You can also produce LaTeX, GNU, and RTF info by using the proper sgmltool (man sgmltools).

1.3 Contributions

Additions and corrections are welcome. Please send them to me (rddunlap@ieee.org). Contributions of section descriptions that are used will be credited to their author(s).

1.4 Trademarks

All trademarks are the property of their respective owners.

1.5 License

Copyright (C) 2001 Randy Dunlap.

This document may be distributed only subject to the terms and conditions set forth in the LDP (Linux Documentation Project) License at "http://www.linuxdoc.org/COPYRIGHT.html".

---

2. Linux init ("ASCII art")

Pictorially (loosely speaking :), Linux initialization looks like this, where "[...]" means optional (depends on the kernel's configuration) and "{...}" is a comment.

```
+-------------------------------+
| arch/i386/boot/setup.S:: +    |
| arch/i386/boot/video.S::      |
|-------------------------------|
| start_of_setup:               |
|   check that loaded OK        |
|   get system memory size      |
|   get video mode(s)           |
|   get hard disk parameters    |
|   get MC bus information      |
|   get mouse information       |
|   get APM BIOS information    |
|   enable address line A20     |
|   reset coprocessor           |
|   mask all interrupts         |
|   move to protected mode      |
|   jmp to startup_32           |
+-------------------------------+
                |
                v
+-------------------------------+
| arch/i386/kernel/head.S::     |
|-------------------------------|
| startup_32:                   |
|   set segment registers to    |
|     known values              |
|   init basic page tables      |
|   setup the stack pointer     |
|   clear kernel BSS            |
|   setup the IDT               |
|   checkCPUtype                |
```

```
|   load GDT, IDT, and LDT       |
|     pointer registers          |
|   start_kernel                 |
|     {it does not return}       |
+--------------------------------+
                |
                v
+--------------------------------+     +--------------------------------+
| init/main.c::                  |  +->| arch/i386/kernel/setup.c::     |
|--------------------------------|  |  |--------------------------------|
| start_kernel():                |  |  | setup_arch():                  |
|   lock_kernel                  |  |  |   copy boot parameters         |
|   setup_arch                   |--+  |   init ramdisk                 |
|   parse_options                |<-+  |   setup_memory_region          |
|   trap_init                    |  |  |   parse_cmd_line               |
|     cpu_init                   |  |  |   use the BIOS memory map to   |
|   init_IRQ                     |  |  |     setup page frame info.     |
|   sched_init                   |  |  |   reserve physical page 0      |
|     init_timervecs             |  |  |  [find_smp_config]             |
|   time_init                    |  |  |   paging_init                  |
|   softirq_init                 |  |  |  [get_smp_config]              |
|   console_init                 |  |  |  [init_apic_mappings]          |
|  [init_modules]                |  |  |  [reserve INITRD memory]       |
|  [profiling setup]             |  |  |   probe_roms to search         |
|   kmem_cache_init              |  |  |     for option ROMs            |
|   sti                          |  |  |   request_resource to          |
|   calibrate_delay              |  |  |     reserve video RAM memory   |
|  [INITRD setup]                |  |  |   request_resource to          |
|   mem_init                     |  |  |     reserve all standard PC    |
|     free_all_bootmem           |  +--|     I/O system board resources|
|   kmem_cache_sizes_init        |     +--------------------------------+
|  [proc_root_init]              |
|   fork_init                    |
|   proc_caches_init             |
|   vfs_caches_init              |
|   buffer_init                  |
|   page_cache_init              |
|   kiobuf_setup                 |
|   signals_init                 |     +--------------------------------+
|   bdev_init                    |     | init/main.c::                  |
|   inode_init                   |     | init(): {...init thread...}    |
|  [ipc_init]                    |     |   do_basic_setup               |
|  [dquot_init_hash]             |     |     {bus/dev init & initcalls}|
|   check_bugs                   |     |   free_initmem                 |
|  [smp_init] {*below}           |     |   open /dev/console            |
|   start init thread {---->}    |.....| exec init script or shell      |
|   unlock_kernel                |     |   or panic                     |
|   cpu_idle                     |     +--------------------------------+
+--------------------------------+


+--------------------------------+
| smpboot.c::smp_init            |
|--------------------------------|
| arch/i386/kernel/smpboot.c::   |
| smp_boot_cpus():               |
|  [mtrr_init_boot_cpu]          |
|   smp_store_cpu_info           |
|   print_cpu_info               |
|   save CPU ID/APIC ID mappings|
|   verify_local_APIC            |
|   connect_bsp_APIC             |
```

```
|   setup_local_APIC             |
|   foreach valid APIC ID        |
|     do_boot_cpu(apicid)        |
|   setup_IO_APIC                |
|   setup_APIC_clocks            |
|   synchronize_tsc_bp           |
+--------------------------------+
```

3. Linux early setup

(from linux/arch/i386/boot/setup.S and linux/arch/i386/boot/video.S)

NOTE: Register notation is %regname and constant notation is a number, with or without a leading '$' sign.

3.1 IA−32 Kernel Setup

"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 the kernel have been loaded by the boot block.

"setup.S" is assembled as 16−bit real−mode code. It switches the processor to 32−bit protected mode and jumps to the 32−bit kernel code.

This code asks the BIOS for memory/disk/other parameters, and puts them in a "safe" place: 0x90000−0x901FF, that is, 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.

The "setup.S" code begins with a jmp instruction around the "setup header", which must begin at location %cs:2.

This is the setup header:

```
                .ascii  "HdrS"          # header signature
                .word   0x0202          # header version number
realmode_swtch: .word   0, 0            # default_switch, SETUPSEG
start_sys_seg:  .word   SYSSEG
                .word   kernel_version  # pointer to kernel version string
type_of_loader: .byte   0
loadflags:
LOADED_HIGH     = 1             # If set, the kernel is loaded high
#ifndef __BIG_KERNEL__
                .byte   0
#else
                .byte   LOADED_HIGH
#endif
setup_move_size: .word  0x8000  # size to move, when setup is not
                                # loaded at 0x90000.
code32_start:                   # here loaders can put a different
                                # start address for 32-bit code.
#ifndef __BIG_KERNEL__
                .long   0x1000  # default for zImage
```

```
#else
                .long   0x100000# default for big kernel
#endif
ramdisk_image:  .long   0       # address of loaded ramdisk image
ramdisk_size:   .long   0       # its size in bytes
bootsect_kludge: .word  bootsect_helper, SETUPSEG
heap_end_ptr:   .word   modelist+1024   # (Header version 0x0201 or later)
                                        # space from here (exclusive) down to
                                        # end of setup code can be used by setup
                                        # for local heap purposes.
pad1:           .word   0
cmd_line_ptr:   .long   0       # (Header version 0x0202 or later)
                                # If nonzero, a 32-bit pointer
                                # to the kernel command line.
trampoline:     call    start_of_setup  # no return from start_of_setup
                .space  1024
# End of setup header #####################################################
```

start_of_setup:

Read second hard drive DASD type

Read the DASD type of the second hard drive (BIOS int. 0x13, %ax=0x1500, %dl=0x81).

Bootlin depends on this being done early. [TBD:why?]

Check that LILO loaded us right

Check the signature words at the end of setup. Signature words are used to ensure that LILO loaded us right. If the two words are not found correctly, copy the setup sectors and check for the signature words again. If they still aren't found, panic("No setup signature found ...").

Check old loader trying to load a big kernel

If the kernel image is "big" (and hence is "loaded high"), then if the loader cannot handle "loaded high" images, then panic ("Wrong loader, giving up...").

Determine system memory size

Get the extended memory size {above 1 MB} in KB. First clear the extended memory size to 0.

#ifndef STANDARD_MEMORY_BIOS_CALL

Clear the E820 memory area counter.

Try three different memory detection schemes.
First, try E820h, which lets us assemble a memory map, then try E801h, which returns a 32−bit memory size, and finally 88h, which returns 0−64 MB.

Method E820H populates a table in the empty_zero_block that contains a list of usable address/size/type tuples. In "linux/arch/i386/kernel/setup.c", this information is transferred into the e820map, and in "linux/arch/i386/mm/init.c", that new information is used to mark pages reserved or not.

Method E820H:
Get the BIOS memory map. E820h returns memory classified into different types and allows memory holes. We scan through this memory map and build a list of the first 32 memory areas {up to 32 entries or BIOS says that there are no more entries}, which we return at "E820MAP". [See URL: http://www.teleport.com/ acpi/acpihtml/topic245.htm]

Method E801H:
We store the 0xe801 memory size in a completely different place, because it will most likely be longer than 16 bits.

This is the sum of 2 registers, normalized to 1 KB chunk sizes: %ecx = memory size from 1 MB to 16 MB range, in 1 KB chunks + %edx = memory size above 16 MB, in 64 KB chunks.

Ye Olde Traditional Methode:
BIOS int. 0x15/AH=0x88 returns the memory size (up to 16 MB or 64 MB, depending on the BIOS). We always use this method, regardless of the results of the other two methods.

#endif

Set the keyboard repeat rate to the maximum rate using using BIOS int. 0x16.

Video adapter modes

Find the video adapter and its supported modes and allow the user to browse video modes.

call video # {see Video section below}

Get Hard Disk parameters

Get hd0 data: Save the hd0 descriptor (from int. vector 0x41) at INITSEG:0x80 length 0x10.

Get hd1 data: Save the hd1 descriptor (from int. vector 0x46) at INITSEG:0x90 length 0x10.

Check that there IS an hd1, using BIOS int. 0x13. If not, clear its descriptor.

Get Micro Channel bus information

Check for Micro Channel (MCA) bus:

- Set MCA feature table length to 0 in case not found.
- Get System Configuration Parameters (BIOS int. 0x15/%ah=0xc0). This sets %es:%bx to point to the system feature table.
- We keep only the first 16 bytes of the system feature table if found: Structure size, Model byte, Submodel byte, BIOS revision, and Feature information bytes 1−5. Bit 0 or 1 (either one) of Feature byte 1 indicates that the system contains a Micro Channel bus.

Check for mouse

Check for PS/2 pointing device by using BIOS int. 0x11 {get equipment list}.

- Clear the pointing device flag (default).
- BIOS int. 0x11: get equipment list.
- If bit 2 (value 0x04) is set, then a mouse is installed and the pointing device flag is set to indicate that the device is present.

Check for APM BIOS support

Check for an APM BIOS (if kernel is configured for APM support):

- start: clear version field to 0, which means no APM BIOS present.
- Check for APM BIOS installation using BIOS int. 0x15.
- If not present, done.
- Check for "PM" signature returned in %bx.
- If no signature, then no APM BIOS: done.
- Check for 32−bit support in %cx.
- If no 32−bit support, no (good) APM BIOS: done. Must have 32−bit APM BIOS support to be used by Linux.
- Save the BIOS code segment, BIOS entry point offset, BIOS 16−bit code segment, BIOS data segment, BIOS code segment length, and BIOS data segment length.
- Record the APM BIOS version and flags.

Prepare to move to protected mode

We build a jump instruction to the kernel's code32_start address. (The loader may have changed it.)

Move the kernel to its correct place if necessary.

Load the segment descriptors (load %ds = %cs).

Make sure that we are at the right position in memory, to accommodate the command line and boot parameters at their fixed locations.

Load the IDT pointer register with 0,0.

Calculate the linear base address of the kernel GDT (table) and load the GDT pointer register with its base address and limit. This early kernel GDT describes kernel code as 4 GB, with base address 0, code/readable/executable, with granularity of 4 KB. The kernel data segment is described as 4 GB, with base address 0, data/readable/writable, with granularity of 4 KB.

Enable address line A20

- Empty the 8042 (keyboard controller) of any queued keys.
- Write 0xd1 (Write Output Port) to Command Register port 0x64.
- Empty the 8042 (keyboard controller) of any queued keys.
- Write 0xdf (Gate A20 + more) to Output port 0x60.
- Empty the 8042 (keyboard controller) of any queued keys.

* Set bit number 1 (value 0x02: FAST_A20) in the "port 0x92" system control register. This enables A20 on some systems, depending on the chipset used in them.
* Wait until A20 really *is* enabled; it can take a fair amount of time on certain systems. The memory location used here (0x200) is the int 0x80 vector, which should be safe to use. When A20 is disabled, the test memory locations are an alias of each other (segment 0:offset 0x200 and segment 0xffff:offset 0x210). {0xffff0 + 0x210 = 0x100200, but if A20 is disabled, this becomes 0x000200.} We just wait (busy wait/loop) until these memory locations are no longer aliased.

Make sure any possible coprocessor is properly reset

* Write 0 to port 0xf0 to clear the Math Coprocessor '−busy' signal.
* Write 0 to port 0xf1 to reset the Math Coprocessor.

Mask all interrupts

Now we mask all interrupts; the rest is done in init_IRQ().

* Mask off all interrupts on the slave PIC: write 0xff to port 0xa1.
* Mask off all interrupts on the master PIC except for IRQ2, which is the cascaded IRQ input from the slave PIC: write 0xfb to port 0x21.

Move to Protected Mode

Now is the time to actually move into protected mode. To make things as simple as possible, we do no register setup or anything, we let the GNU−compiled 32−bit programs do that. We just jump to absolute address 0x1000 (or the loader supplied one), in 32−bit protected mode.

Note that the short jump isn't strictly needed, although there are reasons why it might be a good idea. It won't hurt in any case.

Set the PE (Protected mode Enable) bit in the MSW and jump to the following instruction to flush the instruction fetch queue.

Clear %bx to indicate that this is the BSP (first CPU only).

Jump to startup_32 code

Jump to the 32−bit kernel code (startup_32).

NOTE: For high−loaded big kernels we need:

```
        jmpi    0x100000,__KERNEL_CS
```

but we yet haven't reloaded the %cs register, so the default size of the target offset still is 16 bit. However, using an operand prefix (0x66), the CPU will properly take our 48−bit far pointer. (INTeL 80386 Programmer's Reference Manual, Mixing 16−bit and 32−bit code, page 16−6).

```
        .byte 0x66, 0xea                # prefix + jmpi-opcode
code32: .long   0x1000                  # or 0x100000 for big kernels
        .word   __KERNEL_CS
```

This jumps to "startup_32" in "linux/arch/i386/kernel/head.S".

3.2 Video Setup

"linux/arch/i386/boot/video.S" is included into "linux/arch/i386/boot/setup.S", so they are assembled together. The file separation is a logical module separation even though the two modules aren't built separately.

"video.S" handles Linux/i386 display adapter and video mode setup. For more information about Linux/i386 video modes, see "linux/Documentation/svga.txt" by Martin Mares [mj@ucw.cz].

Video mode selection is a kernel build option. When it is enabled, You can select a specific (fixed) video mode to be used during kernel booting or you can ask to view a selection menu and then choose a video mode from that menu.

There are a few esoteric (!) "video.S" build options that not covered here. See "linux/Documentation/svga.txt" for all of them.

CONFIG_VIDEO_SVGA (for automatic detection of SVGA adapters and modes) is normally #undefined. The normal method of video adapter detection on Linux/i386 is VESA (CONFIG_VIDEO_VESA, for autodetection of VESA modes).

"video:" is the main entry point called by "setup.S". The %ds register *must* be pointing to the bootsector. The "video.S" code uses different segments from the main "setup.S" code.

This is a simplified description of the code flow in "video.S". It does not address the CONFIG_VIDEO_LOCAL, CONFIG_VIDEO_400_HACK, and CONFIG_VIDEO_GFX_HACK build options and it does not dive deep into video BIOS calls or video register accesses.

video:

- %fs is set to the original %ds value
- %ds and %es are set to %cs
- %gs is set to zero
- Detect the video adapter type and supported modes. (call basic_detect)
- #ifdef CONFIG_VIDEO_SELECT
- If the user wants to see a list of the supported VGA adapter modes, list them. (call mode_menu)
- Set the selected video mode. (call mode_set)
- #ifdef CONFIG_VIDEO_RETAIN
- Restore the screen contents. (call restore_screen)
- #endif /* CONFIG_VIDEO_RETAIN */
- #endif /* CONFIG_VIDEO_SELECT */
- Store mode parameters for kernel. (call mode_params)
- Restore original DS register value.

basic_detect:

- Detect if we have CGA, MDA, HGA, EGA, or VGA and pass it to the kernel.
- Check for EGA/VGA using BIOS int. 0x10 calls. This also tells whether the video adapter is CGA/MDA/HGA.

- The "adapter" variable is returned as 0 for CGA/MDA/HGA, 1 for EGA, and 2 for VGA.

mode_params:

- Store the video mode parameters for later use by the kernel. This is done by asking the BIOS for mode parameters except for the rows/columns parameters in the default 80x25 mode −− these are set directly, because some very obscure BIOSes supply insane values.
- #ifdef CONFIG_VIDEO_SELECT
- For graphics mode with a linear frame buffer, goto mopar_gr.
- #endif /* CONFIG_VIDEO_SELECT */
- For MDA/CGA/HGA/EGA/VGA:
- Read and save cursor position.
- Read and save video page/mode/width.
- For MDA/HGA, change the video_segment to $0xb000. (Leave it at its initial value of $0xb800 for all other adapters.)
- Get the Font size (valid only on EGA/VGA).
- Save the number of video columns and lines.

#ifdef CONFIG_VIDEO_SELECT

mopar_gr:

- Get VESA frame buffer parameters.
- Get video mem size and protected mode interface information using BIOS int. 0x10 calls.

mode_menu:

Build the mode list table and display the mode menu.

mode_set:

For the selected video mode, use BIOS int. 0x10 calls or register writes as needed to set some or all of:

- Reset the video mode
- Number of scan lines
- Font pixel size
- Save the screen size in force_size. "force_size" is used to override possibly broken video BIOS interfaces and is used instead of the BIOS variables.

Some video modes require register writes to set:

- Location of the cursor scan lines
- Vertical sync start
- Vertical sync end
- Vertical display end
- Vertical blank start
- Vertical blank end
- Vertical total
- (Vertical) overflow
- Correct sync polarity

* Preserve clock select bits and color bit

{end of mode_set}

#ifdef CONFIG_VIDEO_RETAIN /* Normally _IS_ #defined */

store_screen:

CONFIG_VIDEO_RETAIN is used to retain screen contents when switching modes. This option stores the screen contents to a temporary memory buffer (if there is enough memory) so that they can be restored later.

* Save the current number of video lines and columns, cursor position, and video mode.
* Calculate the image size.
* Save the screen image.
* Set the "do_restore" flag so that the screen contents will be restored at the end of video mode detection/selection.

restore_screen:

Restores screen contents from temporary buffer (if already saved).

* Get parameters of current mode.
* Set cursor position.
* Restore the screen contents.

#endif /* CONFIG_VIDEO_RETAIN */

mode_table:

Build the table of video modes at `modelist'.

* Store standard modes.
* Add modes for standard VGA.
* #ifdef CONFIG_VIDEO_LOCAL
* Add locally−defined video modes. (call local_modes)
* #endif /* CONFIG_VIDEO_LOCAL */
* #ifdef CONFIG_VIDEO_VESA
* Auto−detect VESA VGA modes. (call vesa_modes)
* #endif /* CONFIG_VIDEO_VESA */
* #ifdef CONFIG_VIDEO_SVGA
* Detect SVGA cards & modes. (call svga_modes)
* #endif /* CONFIG_VIDEO_SVGA */
* #ifdef CONFIG_VIDEO_COMPACT
* Compact the video modes list, removing duplicate entries.
* #endif /* CONFIG_VIDEO_COMPACT */

mode_scan:

Scans for video modes.

* Start with mode 0.
* Test the mode.
* Test if it's a text mode.
* OK, store the mode.
* Restore back to mode 3.

#ifdef CONFIG_VIDEO_SVGA

svga_modes:

Try to detect the type of SVGA card and supply (usually approximate) video mode table for it.

* Test all known SVGA adapters.
* Call the test routine for each adapter.
* If adapter is found, copy the video modes.
* Store pointer to card name.

#endif /* CONFIG_VIDEO_SVGA */

#endif /* CONFIG_VIDEO_SELECT */

---

4. Linux architecture−specific initialization

(from "linux/arch/i386/kernel/head.S")

The boot code in "linux/arch/i386/boot/setup.S" transfers execution to the beginning code in "linux/arch/i386/kernel/head.S" (labeled "startup_32:").

To get to this point, a small uncompressed kernel function decompresses the remaining compressed kernel image and then it jumps to the new kernel code.

This is a description of what the "head.S" code does.

4.1 startup_32:

swapper_pg_dir is the top−level page directory, address 0x00101000.

On entry, %esi points to the real−mode code as a 32−bit pointer.

4.2 Set segment registers to known values

Set the %ds, %es, %fs, and %gs registers to __KERNEL_DS.

4.3 SMP BSP (Bootstrap Processor) check

#ifdef CONFIG_SMP

If %bx is zero, this is a boot on the Bootstrap Processor (BSP), so skip this. Otherwise, for an AP (Application Processor):

If the desired %cr4 setting is non−zero, turn on the paging options (PSE, PAE, ...) and skip "Initialize page tables" (jump to "Enable paging").

#endif /* CONFIG_SMP */

4.4 Initialize page tables

Begin at pg0 (page 0) and init all pages to 007 (PRESENT + RW + USER).

4.5 Enable paging

Set %cr3 (page table pointer) to swapper_pg_dir.

Set the paging ("PG") bit of %cr0 to
********** **enable paging** **********.

Jump $ to flush the prefetch queue.

Jump *[$] to make sure that %eip is relocated.

Setup the stack pointer (lss stack_start, %esp).

#ifdef CONFIG_SMP

If this is not the BSP (Bootstrap Processor), clear all flags bits and jump to checkCPUtype.

#endif /* CONFIG_SMP */

4.6 Clear BSS

The BSP clears all of BSS (area between __bss_start and _end) for the kernel.

4.7 32−bit setup

Setup the IDT for 32−bit mode (call setup_idt). setup_idt sets up an IDT with 256 entries pointing to the default interrupt handler "ignore_int" as interrupt gates. It doesn't actually load the IDT; that can be done only after paging has been enabled and the kernel moved to PAGE_OFFSET. Interrupts are enabled elsewhere, when we can be relatively sure everything is OK.

Clear the eflags register (before switching to protected mode).

4.8 Copy boot parameters and command line out of the way

First 2 KB of _empty_zero_page is for boot parameters, second 2 KB is for the command line.

4.9 checkCPUtype

Initialize X86_CPUID to −1.

Use Flags register, push/pop results, and CPUID instruction(s) to determine CPU type and vendor: Sets X86, X86_CPUID, X86_MODEL, X86_MASK, and X86_CAPABILITY. Sets bits in %cr0 accordingly.

Also checks for presence of an 80287 or 80387 coprocessor. Sets X86_HARD_MATH if a math coprocessor or floating point unit is found.

4.10 Count this processor

For CONFIG_SMP builds, increment the "ready" counter to keep a tally of the number of CPUs that have been initialized.

4.11 Load descriptor table pointer registers

Load GDT with gdt_descr and IDT with idt_descr. The GDT contains 2 entries for the kernel (4 GB each for code and data, beginning at 0) and 2 userspace entries (4 GB each for code and data, beginning at 0). There are 2 null descriptors between the userspace descriptors and the APM descriptors.

The GDT also contains 4 entries for APM segments. The APM segments have byte granularity and their bases and limits are set at runtime.

The rest of the gdt_table (after the APM segments) is space for TSSes and LDTs.

Jump to __KERNEL_CS:%eip to cause the GDT to be used. Now in
********** **protected mode** **********.

Reload all of the segment registers: Set the %ds, %es, %fs, and %gs registers to __KERNEL_DS.

#ifdef CONFIG_SMP

Reload the stack pointer segment only (%ss) with __KERNEL_DS.

#else /* not CONFIG_SMP */

Reload the stack pointer (%ss:%esp) with stack_start.

#endif /* CONFIG_SMP */

Clear the LDT pointer to 0.

Clear the processor's Direction Flag (DF) to 0 for gcc.

4.12 Start other processors

For CONFIG_SMP builds, if this is not the first (Bootstrap) CPU, call initialize_secondary(), which does not return. The secondary (AP) processor(s) are initialized and then enter idle state until processes are scheduled on them.

If this is the first or only CPU, call start_kernel(). (see below)

/* the calls above should never return, but in case they do: */

L6: jmp L6

5. Linux architecture−independent initialization

(from "linux/init/main.c")

"linux/init/main.c" begins execution with the start_kernel() function, which is called from "linux/arch/i386/kernel/head.S". start_kernel() never returns to its caller. It ends by calling the cpu_idle() function.

5.1 start_kernel:

Interrupts are still disabled. Do necessary setups, then enable them.

Lock the kernel (BKL: big kernel lock).

Print the linux_banner string (this string resides in "linux/init/version.c") using printk(). NOTE: printk() doesn't actually print this to the console yet; it just buffers the string until a console device registers itself with the kernel, then the kernel passes the buffered console log contents to the registered console device(s). There can be multiple registered console devices.

********** printk() can be called very early because it doesn't actually print to anywhere. It just logs the message to "log_buf", which is allocated statically in "linux/kernel/printk.c". The messages that are saved in "log_buf" are passed to registered console devices as they register. **********

More architecture−specific init

Call setup_arch(&command_line):

This performs architecture−specific initializations (details below). Then back to architecture−independent initialization....

The remainder of start_kernel() is done as follows for all processor architecures, although several of these function calls are to architecture−specific setup/init functions.

Continue architecture−independent init

Print the kernel command line.

Parsing command line options

parse_options(command_line): Parse the kernel options on the command line. This is a simple kernel command line parsing function. It parses the command line and fills in the arguments and environment to init (thread) as appropriate. Any command−line option is taken to be an environment variable if it contains the character '='. It also checks for options meant for the kernel by calling checksetup(), which checks the command line for kernel parameters, these being specified by declaring them using "__setup", as in:

```
__setup("debug", debug_kernel);
```

This declaration causes the debug_kernel() function to be called when the string "debug" is scanned. See "linux/Documentation/kernel−parameters.txt" for the list of kernel parameters.

These options are not given to init −− they are for internal kernel use only. The default argument list for the init thread is {"init", NULL}, with a maximum of 8 command−line arguments. The default environment list for the init thread is {"HOME=/", "TERM=linux", NULL}, with a maximum of 8 command−line environment variable settings. In case LILO is going to boot us with default command line, it prepends "auto" before the whole cmdline which makes the shell think it should execute a script with such name. So we ignore all arguments entered _before_ init=... [MJ]

trap_init

(in linux/arch/i386/kernel/traps.c)

Install exception handlers for basic processor exceptions, i.e., not hardware device interrupt handlers.

Install the handler for the system call software interrupt.

Install handlers for lcall7 (for iBCS) and lcall27 (for Solaris/x86 binaries).

Call cpu_init() to do:

- initialize per−CPU state
- reload the GDT and IDT
- mask off the eflags NT (Nested Task) bit
- set up and load the per−CPU TSS and LDT
- clear 6 debug registers (0, 1, 2, 3, 6, and 7)
- stts(): set the 0x08 bit (TS: Task Switched) in CR0 to enable lazy register saves on context switches

init_IRQ

(in linux/arch/i386/kernel/i8259.c)

Call init_ISA_irqs() to initialize the two 8259A interrupt controllers and install default interrupt handlers for the ISA IRQs.

Set an interrupt gate for all unused interrupt vectors.

For CONFIG_SMP configurations, set up IRQ 0 early, since it's used before the IO APIC is set up.

For CONFIG_SMP, install the interrupt handler for CPU−to−CPU IPIs that are used for the "reschedule helper."

For CONFIG_SMP, install the interrupt handler for the IPI that is used to invalidate TLBs.

For CONFIG_SMP, install the interrupt handler for the IPI that is used for generic function calls.

For CONFIG_X86_LOCAL_APIC configurations, install the interrupt handler for the self−generated local APIC timer IPI.

For CONFIG_X86_LOCAL_APIC configurations, install interrupt handlers for spurious and error interrupts.

Set the system's clock chip to generate a timer tick interrupt every HZ Hz.

If the system has an external FPU, set up IRQ 13 to handle floating point exceptions.

sched_init

(in linux/kernel/sched.c)

- Set the init_task's processor ID.
- Clear the pidhash table. TBD: Why? isn't it in BSS?
- call init_timervecs()
- call init_bh() to init "bottom half" queues for timer_bh, tqueue_bh, and immediate_bh.

time_init

(in linux/arch/i386/kernel/time.c)

Initialize the system's current time of day (xtime) from CMOS.

Install the irq0 timer tick interrupt handler.

softirq_init

(in linux/kernel/softirq.c)

console_init

(in linux/drivers/char/tty_io.c)

HACK ALERT! This is early. We're enabling the console before we've done PCI setups etc., and console_init() must be aware of this. But we do want output early, in case something goes wrong.

init_modules

(in linux/kernel/module.c)

For CONFIG_MODULES configurations, call init_modules(). This initializes the size (or number of symbols) of the kernel symbol table.

Profiling setup

if profiling ("profile=#" on the kernel command line): calculate the kernel text (code) profile "segment" size; calculate the profile buffer size in pages (round up); allocate the profile buffer: prof_buffer = alloc_bootmem(size);

kmem_cache_init

(in linux/mm/slab.c)

sti

********** **Interrupts are now enabled.** **********
This allows "calibrate_delay()" (below) to work.

calibrate_delay

Calculate the "loops_per_jiffy" delay loop value and print it in BogoMIPS.

INITRD setup

```
#ifdef CONFIG_BLK_DEV_INITRD

        if (initrd_start && !initrd_below_start_ok &&
                        initrd_start < (min_low_pfn << PAGE_SHIFT)) {
                printk("initrd overwritten (initrd_start < (min_low_pfn << PAGE_SHIFT)) - disabling it.\n");
                initrd_start = 0;       // mark initrd as disabled
        }

#endif /* CONFIG_BLK_DEV_INITRD */
```

mem_init

(in linux/arch/i386/mm/init.c)

- Clear the empty_zero_page.
- Call free_all_bootmem() and add that released memory to totalram_pages.
- Count the number of reserved RAM pages.
- Print the system memory sizes (free/total), kernel code size, reserved memory size, kernel data size, kernel "init" size, and the highmem size.
- For CONFIG_SMP, call zap_low_mappings().

********** get_free_pages() can be used after mem_init(). **********

kmem_cache_sizes_init

(in linux/mm/slab.c)

Set up remaining internal and general caches. Called after the "get_free_page()" functions have been enabled and before smp_init().

********** kmalloc() can be used after kmem_cache_sizes_init(). **********

proc_root_init

(in linux/fs/proc/root.c)

For CONFIG_PROC_FS configurations:

- call proc_misc_init()
- mkdir /proc/net
- for CONFIG_SYSVIPC, mkdir /proc/sysvipc
- for CONFIG_SYSCTL, mkdir /proc/sys
- mkdir /proc/fs
- mkdir /proc/driver
- call proc_tty_init()
- mkdir /proc/bus

mempages = num_physpages;

fork_init(mempages)

(in linux/kernel/fork.c)

The default maximum number of threads is set to a safe value: the thread structures can take up at most half of memory.

proc_caches_init()

(in linux/kernel/fork.c)

Call kmem_cache_create() to create slab caches for signal_act (signal action), files_cache (files_struct), fs_cache (fs_struct), vm_area_struct, and mm_struct.

vfs_caches_init(mempages)

(in linux/fs/dcache.c)

Call kmem_cache_create() to create slab caches for buffer_head, names_cache, filp, and for CONFIG_QUOTA, dquot.

Call dcache_init() to create the dentry_cache and dentry_hashtable.

buffer_init(mempages)

(in linux/fs/buffer.c)

Allocate the buffer cache hash table and init the free list.
Use get_free_pages() for the hash table to decrease TLB misses; use SLAB cache for buffer heads.
Setup the hash chains, free lists, and LRU lists.

page_cache_init(mempages)

(in linux/mm/filemap.c)

Allocate and clear the page–cache hash table.

kiobuf_setup()

(in linux/fs/iobuf.c)

Call kmem_cache_create() to create the kernel iobuf cache.

signals_init()

(in linux/kernel/signal.c)

Call kmem_cache_create() to create the "sigqueue" SLAB cache.

bdev_init()

(in linux/fs/block_dev.c)

Initialize the bdev_hashtable list heads.

Call kmem_cache_create() to create the "bdev_cache" SLAB cache.

inode_init(mempages)

(in linux/fs/inode.c)

- Allocate memory for the inode_hashtable.
- Intialize the inode_hashtable list heads.
- Call kmem_cache_create() to create the inode SLAB cache.

ipc_init()

(in linux/ipc/util.c)

For CONFIG_SYSVIPC configurations, call ipc_init().

The various System V IPC resources (semaphores, messages, and shared memory) are initialized.

dquot_init_hash()

(in linux/fs/dquot.c)

For CONFIG_QUOTA configurations, call dquot_init_hash().

- Clear dquot_hash. TBD: Why? Is it in BSS? Yes.
- Clear dqstats. TBD: Why? Is it in BSS? Yes.

check_bugs()

(in linux/include/asm−i386/bugs.h)

- identify_cpu()
- For non−CONFIG_SMP configurations, print_cpu_info()
- check_config()
- check_fpu()
- check_hlt()
- check_popad()
- Update system_utsname.machine{byte 1} with boot_cpu_data.x86

Start other SMP processors (as applicable)

smp_init() works in one of three ways, depending upon the kernel configuration.

For a uniprocessor (UP) system without an IO APIC (CONFIG_X86_IO_APIC is not defined), smp_init() is empty −− it has nothing to do.

For a UP system with (an) IO APIC for interrupt routing, it calls IO_APIC_init_uniprocessor().

For an SMP system, its main job is to call the architecture−specific function "smp_boot_cpus()", which does the following.

- For CONFIG_MTRR kernels, calls mtrr_init_boot_cpu(), which must be done before the other processors are booted.
- Stores and prints the BSP CPU information.
- Saves the BSP APIC ID and BSP logical CPU ID (latter is 0).
- If an MP BIOS interrupt routing table was not found, revert to using only one CPU and exit.
- Verify existence of a local APIC for the BSP.
- If the "maxcpus" boot option was used to limit the number of CPUs actually used to 1 (not SMP), then ignore the MP BIOS interrupt routing table.
- Switch the system from PIC mode to symmetric I/O interrupt mode.
- Setup the BSP's local APIC.
- Use the CPU present map to boot the APs serially. Wait for each AP to finish booting before starting the next one.
- If using (an) IO APIC {which is True unless the "noapic" boot option was used}, setup the IO APIC(s).

Start init thread

We count on the initial thread going OK.

Like idlers, init is an unlocked kernel thread, which will make syscalls (and thus be locked).

```
kernel_thread(init, NULL, CLONE_FS | CLONE_FILES | CLONE_SIGNAL);
```

{details below}

unlock_kernel()

Release the BKL.

current−>need_resched = 1;

cpu_idle()

This function remains as process number 0. Its purpose is to use up idle CPU cycles. If the kernel is configured for APM support or ACPI support, cpu_idle() invokes the supported power−saving features of these specifications. Otherwise it nominally executes a "hlt" instruction.

{end of start_kernel()}

5.2 setup_arch

(in "linux/arch/i386/kernel/setup.c")

Copy and convert system parameter data

Copy and convert parameter data passed from 16−bit real mode to the 32−bit startup code.

For RAMdisk−enabled configs (CONFIG_BLK_DEV_RAM)

Initialize rd_image_start, rd_prompt, and rd_doload from the real−mode parameter data.

setup_memory_region

Use the BIOS−supplied memory map to setup memory regions.

Set memory limits

Set values for the start of kernel code, end of kernel code, end of kernel data, and "_end" (end of kernel code = the "brk" address).

Set values for code_resource start and end and data_resource start and end.

parse_mem_cmdline

Parse any "mem=" parameters on the kernel command line and remember them.

Setup Page Frames

Use the BIOS−supplied memory map to setup page frames.

Register available low RAM pages with the bootmem allocator.

Reserve physical page 0: "it's a special BIOS page on many boxes, enabling clean reboots, SMP operation, laptop functions."

Handle SMP and IO APIC Configurations

For CONFIG_SMP, reserve the page immediately above page 0 for stack and trampoline usage, then call smp_alloc_memory() to allocate low memory for AP processor(s) real mode trampoline code.

For CONFIG_X86_IO_APIC configurations, call find_smp_config() to find and reserve any boot−time SMP configuration information memory, such as MP (Multi Processor) table data from the BIOS.

paging_init()

paging_init() sets up the page tables − note that the first 8 MB are already mapped by head.S.

This routine also unmaps the page at virtual kernel address 0, so that we can trap those pesky NULL−reference errors in the kernel.

Save the boot−time SMP configuration

For CONFIG_X86_IO_APIC configurations, call get_smp_config() to read and save the MP table IO APIC interrupt routing configuration data.

For CONFIG_X86_LOCAL_APIC configurations, call init_apic_mappings().

Reserve INITRD memory

For CONFIG_BLK_DEV_INITRD configurations, if there is enough memory for the initial RamDisk, call reserve_bootmem() to reserve RAM for the initial RamDisk.

Scan for option ROMs

Call probe_roms() and reserve their memory space resource(s) if found and valid. This is done for the standard video BIOS ROM image, any option ROMs found, and for the system board extension ROM (space).

Reserve system resources

Call request_resource() to reserve video RAM memory.

Call request_resource() to reserve all standard PC I/O system board resources.

{end of setup_arch()}

5.3 init thread

The init thread begins at the init() function in "linux/init/main.c". This is always expected to be process number 1.

init() first locks the kernel and then calls do_basic_setup() to perform lots of bus and/or device initialization {more detail below}. After do_basic_setup(), most kernel initialization has been completed. init() then frees any memory that was specified as being for initialization only [marked with "__init", "__initdata", "__init_call", or "__initsetup"] and unlocks the kernel (BKL).

init() next opens /dev/console and duplicates that file descriptor two times to create stdin, stdout, and stderr files for init and all of its children.

Finally init() tries to execute the command specified on the kernel parameters command line if there was one, or an init program or script if it can find one in {/sbin/init, /etc/init, /bin/init}, and lastly /bin/sh. If init() cannot execute any of these, it panics ("No init found. Try passing init= option to kernel.")

5.4 do_basic_setup {part of the init thread}

The machine is now initialized. None of the devices have been touched yet, but the CPU subsystem is up and running, and memory and process management works.

Be the reaper of orphaned children

The init process handles all orphaned tasks.

MTRRs

// SMP init is completed before this.
For CONFIG_MTRR, call mtrr_init() [in linux/arch/i386/kernel/mtrr.c].

SYSCTLs

For CONFIG_SYSCTL configurations, call sysctl_init() [in linux/kernel/sysctl.c].

Init Many Devices

```
/*
 * Ok, at this point all CPU's should be initialized, so
 * we can start looking into devices..
 */
```

PCI

For CONFIG_PCI configurations, call pci_init() [in linux/drivers/pci/pci.c].

Micro Channel

For CONFIG_MCA configurations, call mca_init() [in linux/arch/i386/kernel/mca.c].

ISA PnP

For CONFIG_ISAPNP configurations, call isapnp_init() [in linux/drivers/pnp/isapnp.c].

Networking Init

```
        /* Networking initialization needs a process context */
        sock_init();
```

[in linux/net/socket.c]

Initial RamDisk

```
#ifdef CONFIG_BLK_DEV_INITRD

        real_root_dev = ROOT_DEV;
        real_root_mountflags = root_mountflags;
        if (initrd_start && mount_initrd)
                root_mountflags &= ~MS_RDONLY;      // change to read/write
        else
                mount_initrd =0;

#endif /* CONFIG_BLK_DEV_INITRD */
```

Start the kernel "context" thread (keventd)

[in linux/kernel/context.c]

Initcalls

Call all functions marked as "__initcall":

```
        do_initcalls();
```

[in linux/init/main.c]

This initializes many functions and some subsystems −−− in no specific or guaranteed order unless fixed in their Makefiles −−− if they were built into the kernel, such as:

- APM: apm_init() {in linux/arch/i386/kernel/apm.c}
- cpuid: cpuid_init() {in linux/arch/i386/kernel/cpuid.c}
- DMI: dmi_scan_machine() {in linux/arch/i386/kernel/dmi_scan.c}
- microcode: microcode_init() {in linux/arch/i386/kernel/microcode.c}

- MSR: msr_init() {in linux/arch/i386/kernel/msr.c}
- partitions: partition_setup() {in linux/fs/partitions/check.s}
- file systems, pipes, buffer and cache management, various binary format loaders, NLS character sets: too numerous to list {in linux/fs/*}
- user cache (for limits): uid_cache_init() {in linux/kernel/user.c}
- kmem_cpu_cache: kmem_cpucache_init() {in linux/mm/slab.c}
- shmem: init_shmem_fs() {in linux/mm/shmem.c}
- kswapd: kswapd_init() {in linux/mm/vmscan.c}
- networking, TCP/IP, IPv6, sockets, 802.2, SNAP, LLC, X.25, AX.25, IPX, kHTTPd, ATM LAN emulation (LANE), IP chains/forwarding, NAT/masquerading, packet matching/filtering/logging, firewalling, DECnet, bridging, and other networking protocols too numerous to list {in linux/net/*}
- drivers, some of which are not exactly device drivers, but help out with bus/device enumeration and initialization, such as:
- ACPI: acpi_init() {in linux/drivers/acpi/*}
- PCI: pci_proc_init() {in linux/drivers/pci/*}
- PCMCIA controllers {in linux/drivers/pcmcia/*}
- and...
- atm drivers {in linux/drivers/atm/*}
- block drivers {in linux/drivers/block/*}
- CD–ROM drivers {in linux/drivers/cdrom/*}
- character drivers {in linux/drivers/char/*}
- I2O drivers {in linux/drivers/i2o/*}
- IDE drivers {in linux/drivers/ide/*}
- input drivers (keyboard/mouse/joystick) {in linux/drivers/input/*}
- ISDN drivers {in linux/drivers/isdn/*}
- md, LVM, and RAID drivers {in linux/drivers/md/*}
- radio drivers {in linux/drivers/media/radio/*}
- video drivers {in linux/drivers/media/video/*}
- MTD drivers {in linux/drivers/mtd/*}
- network drivers, including PLIP, PPP, dummy, Ethernet, bonding, Arcnet, hamradio, PCMCIA, Token Ring, and WAN
- SCSI logical and physical drivers {in linux/drivers/scsi/*}
- sound drivers {in linux/drivers/sound/*}
- telephony drivers {in linux/drivers/telephony/*}
- USB host controllers and device drivers {in linux/drivers/usb/*}
- video frame buffer drivers {in linux/drivers/video/*}

Filesystems

Call filesystem_setup():

- init_devfs_fs(); /* Header file may make this empty */
- For CONFIG_NFS_FS configurations, call init_nfs_fs().
- For CONFIG_DEVPTS_FS configurations, call init_devpts_fs().

[in linux/fs/filesystems.c]

IRDA

For CONFIG_IRDA configurations, call irda_device_init().
/* Must be done after protocol initialization */

[in linux/net/irda/irda_device.c]

PCMCIA

/* Do this last */
For CONFIG_PCMCIA configurations, call init_pcmcia_ds().
[in linux/drivers/pcmcia/ds.c]

Mount the root filesystem

```
        mount_root();
```

[in linux/fs/super.c]

Mount the dev (device) filesystem

```
        mount_devfs_fs ();
```

[in linux/fs/devfs/base.c]

Switch to the Initial RamDisk

```
#ifdef CONFIG_BLK_DEV_INITRD

        if (mount_initrd && MAJOR(ROOT_DEV) == RAMDISK_MAJOR && MINOR(ROOT_DEV) == 0) {
                // Start the linuxrc thread.
                pid = kernel_thread(do_linuxrc, "/linuxrc", SIGCHLD);
                if (pid > 0)
                        while (pid != wait(&i));
                if (MAJOR(real_root_dev) != RAMDISK_MAJOR
                     || MINOR(real_root_dev) != 0) {
                        error = change_root(real_root_dev,"/initrd");
                        if (error)
                                printk(KERN_ERR "Change root to /initrd: "
                                    "error %d\n",error);
                }
        }

#endif /* CONFIG_BLK_DEV_INITRD */
```

See "linux/Documentation/initrd.txt" for more information on initial RAM disks.

{end of do_basic_setup()}

---

6. Glossary

AP: Application Processor, any x86 processor other than the Bootstrap Processor on IA−32 SMP systems

ACPI: Advanced Configuration and Power Interface

APIC: Advanced Programmable Interrupt Controller

APM: Advanced Power Management, a BIOS−managed power management specification for personal computers

BSP: Bootstrap Processor, the primary booting processor on IA−32 SMP systems

BSS: Block Started by Symbol: the uninitialized data segment

BKL: Big Kernel Lock, the Linux global kernel lock

CRn: Control Register n, i386−specific control registers

FPU: Floating Point Unit, a separate math coprocessor device

GB: gigabyte (1024 * 1024 * 1024 bytes)

GDT: Global Descriptor Table, an i386 memory management table

IA: Intel Architecture (also i386, x86)

IDT: Interrupt Descriptor Table, an i386−specific table that contains information used in handling interrupts

initrd: initial RAM disk (see "linux/Documentation/initrd.txt")

IPC: Inter−Process Communication

IPI: Inter−processor Interrupt, a method of signaling interrupts between multiple processors on an SMP system

IRDA: InfraRed Data Association

IRQ: Interrupt ReQuest

ISA: Industry Standard Architecture

KB: kilobyte (1024 bytes)

LDT: Local Descriptor Table, an i386−specific memory management table that is used to describe memory for each non−kernel process

MB: megabyte (1024 * 1024 bytes)

MCA: Micro Channel Architecture, used in IBM PS/2 computers

MP: Multi−processor

MSW: Machine Status Word

MTRR: Memory Type Range Registers

PAE: Physical Address Extension: extends the address space to 64 GB instead of 4 GB

PCI: Peripheral Component Interconnect, an industry standard for connecting devices on a local bus in a computer system

PCMCIA: Personal Computer Memory Card International Association; defines standards for PCMCIA cards and CardBus PC Cards

PIC: Programmable Interrupt Controller

PNP: Plug aNd Play

PSE: Page Size Extension: allows 4 MB pages

SMP: Symmetric Multi Processor/Processing

TLB: Translation Lookaside Buffer, i386−specific processor cache of recent page directory and page table entries

TSS: Task State Segment, an i386−specific task data structure

UP: Uniprocessor (single CPU) system.

---

7. References

1. Tigran Aivazian, "Linux Kernel Internals" (URL: http://www.moses.uklinux.net/patches/lki.html)
2. Werner Almesberger, x86 Booting. (URL: ftp://icaftp.epfl.ch/pub/people/almesber/booting/)
3. Werner Almesberger, "LILO Generic boot loader for Linux: Technical overview." December 4, 1998. Included in LILO distribution.
4. Werner Almesberger and Hans Lermen, Using the initial RAM disk (initrd). (file: linux/Documentation/initrd.txt)
5. H. Peter Anvin, "The Linux/I386 Boot Protocol (file: linux/Documentation/i386/boot.txt)
6. Michael Beck et al, "Linux Kernel Internals," second edition. Addison−Wesley, 1998.
7. Ralf Brown's Interrupt List, URL: http://www.ctyme.com/intr/int.htm {browsable}
8. Ralf Brown's Interrupt List, URL: http://www.delorie.com/djgpp/doc/rbinter/ix/ {browsable}
9. Ralf Brown's Interrupt List, URL: http://www.cs.cmu.edu/ ralf/files.html {zipped, not browsable}
10. E820 memory sizing method: URL: http://www.teleport.com/ acpi/acpihtml/topic245.htm
11. IBM Personal Computer AT Technical Reference. 1985.
12. IBM Personal System/2(r) and Personal Computer BIOS Interface Technical Reference, second edition. 1988.
13. Hans Lermen and Martin Mares, "Summary of empty_zero_page layout." (file: linux/Documentation/i386/zero−page.txt)
14. linux/Documentation directory files
15. Martin Mares, "Video Mode Selection Support." (file: linux/Documentation/svga.txt)
16. Scott Maxwell, "Linux Core Kernel Commentary." Coriolis Press, 1999.
17. Mindshare, Inc., Tom Shanley, "Pentium(r) Pro and Pentium(r) II System Architecture," second edition. Addison−Wesley, 1998.
18. Allesandro Rubini, "Linux Device Drivers." O'Reilly and Associates, 1998.
19. URL: ftp://linux01.gwdg.de/pub/cLIeNUX/interim/Janet_Reno.tgz
20. URL: http://www.eecs.wsu.edu/~cs640/ (was dead at last check)
21. URL: http://www.linuxbios.org + "Papers"