Linux i386 Boot Code HOWTO

Feiyun Wang

<feiyunw@yahoo.com>

2004−01−23

Revision History

Revision 0.3.3	2004−01−23	Revised by: fyw
Add decompress_kernel() details; Fix bugs reported in TLDP final review.		
Revision 0.3	2003−12−07	Revised by: fyw
Add contents on SMP, GRUB and LILO; Fix and enhance.		
Revision 0.2	2003−08−17	Revised by: fyw
Adapt to Linux 2.4.20.		
Revision 0.1	2003−04−20	Revised by: fyw
Change to DocBook XML format.		

This document describes Linux i386 boot code, serving as a study guide and source commentary. In addition to C−like pseudocode source commentary, it also presents keynotes of toolchains and specs related to kernel development. It is designed to help:

- kernel newbies to understand Linux i386 boot code, and
- kernel veterans to recall Linux boot procedure.

Table of Contents

Table of Contents

1. Introduction

This document serves as a study guide and source commentary for Linux i386 boot code. In addition to C–like pseudocode source commentary, it also presents keynotes of toolchains and specs related to kernel development. It is designed to help:

- kernel newbies to understand Linux i386 boot code, and
- kernel veterans to recall Linux boot procedure.

Current release is based on Linux 2.4.20.

The project homepage for this document is hosted by China Linux Forum. Working documents may also be found at the author's personal webpage at Yahoo! GeoCities.

1.1. Copyright and License

1.2. Disclaimer

No liability for the contents of this document can be accepted. Use the concepts, examples and information at your own risk. There may be errors and inaccuracies which could be damaging to your system. Proceed with caution, and although this is highly unlikely, the author(s) do not take any responsibility.

Owners hold all copyrights, unless specifically noted otherwise. Use of a term in this document should not be regarded as affecting the validity of any trademark or service mark. Naming of particular products or brands should not be seen as endorsements.

1.3. Credits / Contributors

In this document, I have the pleasure of acknowledging:

- Jennifer Riley `<kevten@email.com>`
- Tabatha Marshall `<tabatha@merlinmonroe.com>`
- Randy Dunlap `<rddunlap@ieee.org>`

Names will remain on this list for a year.

1.4. Feedback

Feedback is most certainly welcome for this document. Send your additions, comments and criticisms to the following email address:

* Feiyun Wang <feiyunw@yahoo.com>

1.5. Translations

English is the only version available now.

2. Linux Makefiles

Before perusing Linux code, we should get some basic idea about how Linux is composed, compiled and linked. A straightforward way to achieve this goal is to understand Linux makefiles. Check Cross−Referencing Linux if you prefer online source browsing.

2.1. linux/Makefile

Here are some well−known targets in this top−level makefile:

- *xconfig, menuconfig, config, oldconfig*: generate kernel configuration file `linux/.config`;
- *depend, dep*: generate dependency files, like `linux/.depend`, `linux/.hdepend` and `.depend` in subdirectories;
- *vmlinux*: generate resident kernel image `linux/vmlinux`, the most important target;
- *modules, modules_install*: generate and install modules in `/lib/modules/$(KERNELRELEASE)`;
- *tags*: generate tag file `linux/tags`, for source browsing with vim.

Overview of `linux/Makefile` is outlined below:

```
include .depend
include .config
include arch/i386/Makefile

vmlinux: generate linux/vmlinux
        /* entry point "stext" defined in arch/i386/kernel/head.S */
        $(LD) -T $(TOPDIR)/arch/i386/vmlinux.lds -e stext
        /* $(HEAD) */
        + from arch/i386/Makefile
                arch/i386/kernel/head.o
                arch/i386/kernel/init_task.o
        init/main.o
        init/version.o
        init/do_mounts.o
        --start-group
        /* $(CORE_FILES) */
        + from arch/i386/Makefile
                arch/i386/kernel/kernel.o
                arch/i386/mm/mm.o
        kernel/kernel.o
        mm/mm.o
        fs/fs.o
        ipc/ipc.o
        /* $(DRIVERS) */
        drivers/...
                char/char.o
                block/block.o
                misc/misc.o
                net/net.o
                media/media.o
                cdrom/driver.o
                and other static linked drivers
                + from arch/i386/Makefile
                        arch/i386/math-emu/math.o (ifdef CONFIG_MATH_EMULATION)
        /* $(NETWORKS) */
        net/network.o
```

```
        /* $(LIBS) */
        + from arch/i386/Makefile
                arch/i386/lib/lib.a
        lib/lib.a
        --end-group
        -o vmlinux
        $(NM) vmlinux | grep ... | sort > System.map
tags: generate linux/tags for vim
modules: generate modules
modules_install: install modules
clean mrproper distclean: clean up build directory
psdocs pdfdocs htmldocs mandocs: generate kernel documents

include Rules.make

rpm: generate an rpm
```

"−−start−group" and "−−end−group" are **ld** command line options to resolve symbol reference problem. Refer to Using LD, the GNU linker: Command Line Options for details.

`Rules.make` contains rules which are shared between multiple Makefiles.

2.2. linux/arch/i386/vmlinux.lds

After compilation, **ld** combines a number of object and archive files, relocates their data and ties up symbol references. `linux/arch/i386/vmlinux.lds` is designated by `linux/Makefile` as the linker script used in linking the resident kernel image `linux/vmlinux`.

```
/* ld script to make i386 Linux kernel
 * Written by Martin Mares <mj@atrey.karlin.mff.cuni.cz>;
 */
OUTPUT_FORMAT("elf32-i386", "elf32-i386", "elf32-i386")
OUTPUT_ARCH(i386)
/* "ENTRY" is overridden by command line option "-e stext" in linux/Makefile */
ENTRY(_start)
/* Output file (linux/vmlinux) layout.
 * Refer to Using LD, the GNU linker: Specifying Output Sections */
SECTIONS
{
/* Output section .text starts at address 3G+1M.
 * Refer to Using LD, the GNU linker: The Location Counter */
  . = 0xC0000000 + 0x100000;
  _text = .;                    /* Text and read-only data */
  .text : {
        *(.text)
        *(.fixup)
        *(.gnu.warning)
        } = 0x9090
/* Unallocated holes filled with 0x9090, i.e. opcode for "NOP NOP".
 * Refer to Using LD, the GNU linker: Optional Section Attributes */

  _etext = .;                   /* End of text section */

  .rodata : { *(.rodata) *(.rodata.*) }
  .kstrtab : { *(.kstrtab) }

/* Aligned to next 16-bytes boundary.
 * Refer to Using LD, the GNU linker: Arithmetic Functions */
```

```
  . = ALIGN(16);                /* Exception table */
  __start___ex_table = .;
  __ex_table : { *(__ex_table) }
  __stop___ex_table = .;

  __start___ksymtab = .;        /* Kernel symbol table */
  __ksymtab : { *(__ksymtab) }
  __stop___ksymtab = .;

  .data : {                     /* Data */
        *(.data)
        CONSTRUCTORS
        }
/* For "CONSTRUCTORS", refer to
 * Using LD, the GNU linker: Option Commands */

  _edata = .;                   /* End of data section */

  . = ALIGN(8192);              /* init_task */
  .data.init_task : { *(.data.init_task) }

  . = ALIGN(4096);              /* Init code and data */
  __init_begin = .;
  .text.init : { *(.text.init) }
  .data.init : { *(.data.init) }
  . = ALIGN(16);
  __setup_start = .;
  .setup.init : { *(.setup.init) }
  __setup_end = .;
  __initcall_start = .;
  .initcall.init : { *(.initcall.init) }
  __initcall_end = .;
  . = ALIGN(4096);
  __init_end = .;

  . = ALIGN(4096);
  .data.page_aligned : { *(.data.idt) }

  . = ALIGN(32);
  .data.cacheline_aligned : { *(.data.cacheline_aligned) }

  __bss_start = .;              /* BSS */
  .bss : {
        *(.bss)
        }
  _end = . ;

/* Output section /DISCARD/ will not be included in the final link output.
 * Refer to Using LD, the GNU linker: Section Definitions */
  /* Sections to be discarded */
  /DISCARD/ : {
        *(.text.exit)
        *(.data.exit)
        *(.exitcall.exit)
        }

/* The following output sections are addressed at memory location 0.
 * Refer to Using LD, the GNU linker: Optional Section Attributes */
  /* Stabs debugging sections.  */
  .stab 0 : { *(.stab) }
  .stabstr 0 : { *(.stabstr) }
  .stab.excl 0 : { *(.stab.excl) }
```

```
  .stab.exclstr 0 : { *(.stab.exclstr) }
  .stab.index 0 : { *(.stab.index) }
  .stab.indexstr 0 : { *(.stab.indexstr) }
  .comment 0 : { *(.comment) }
}
```

2.3. linux/arch/i386/Makefile

`linux/arch/i386/Makefile` is included by `linux/Makefile` to provide i386 specific items and terms.

All the following targets depend on target *vmlinux* of `linux/Makefile`. They are accomplished by making corresponding targets in `linux/arch/i386/boot/Makefile` with some options.

Table 1. Targets in linux/arch/i386/Makefile

Target	Command
zImage [a]	**@$(MAKE) –C arch/i386/boot zImage** [b]
bzImage	**@$(MAKE) –C arch/i386/boot bzImage**
zlilo	**@$(MAKE) –C arch/i386/boot BOOTIMAGE=zImage zlilo**
bzlilo	**@$(MAKE) –C arch/i386/boot BOOTIMAGE=bzImage zlilo**
zdisk	**@$(MAKE) –C arch/i386/boot BOOTIMAGE=zImage zdisk**
bzdisk	**@$(MAKE) –C arch/i386/boot BOOTIMAGE=bzImage zdisk**
install	**@$(MAKE) –C arch/i386/boot BOOTIMAGE=bzImage install**
Notes:a. *zImage* alias: *compressed*; b. "–C" is a MAKE command line option to change directory before reading makefiles; Refer to GNU make: Summary of Options and GNU make: Recursive Use of make.	

It is worth noticing that this makefile redefines some environment variables which are exported by `linux/Makefile`, specifically:

```
OBJCOPY=$(CROSS_COMPILE)objcopy -O binary -R .note -R .comment -S
```

The effect will be passed to subdirectory makefiles and will change the tool's behavior. Refer to GNU Binary Utilities: objcopy for **objcopy** command line option details.

Not sure why *$(LIBS)* includes "$(TOPDIR)/arch/i386/lib/lib.a" twice:

```
LIBS := $(TOPDIR)/arch/i386/lib/lib.a $(LIBS) $(TOPDIR)/arch/i386/lib/lib.a
```

It may be employed to work around linking problems with some toolchains.

2.4. linux/arch/i386/boot/Makefile

`linux/arch/i386/boot/Makefile` is somehow independent as it is not included by either `linux/arch/i386/Makefile` or `linux/Makefile`.

However, they do have some relationship:

- `linux/Makefile`: provides resident kernel image `linux/vmlinux`;
- `linux/arch/i386/boot/Makefile`: provides bootstrap;
- `linux/arch/i386/Makefile`: makes sure `linux/vmlinux` is ready before the bootstrap is constructed, and exports targets (like *bzImage*) to `linux/Makefile`.

$(BOOTIMAGE) value, which is for target *zdisk, zlilo* or *zdisk*, comes from `linux/arch/i386/Makefile`.

Table 2. Targets in linux/arch/i386/boot/Makefile

Target	Command
zImage	`$(OBJCOPY) compressed/vmlinux compressed/vmlinux.out` `tools/build bootsect setup compressed/vmlinux.out $(ROOT_DEV) > zImage`

bzImage	`$(OBJCOPY) compressed/bvmlinux compressed/bvmlinux.out`
	`tools/build -b bbootsect bsetup compressed/bvmlinux.out $(ROOT_DEV) \`
	`        > bzImage`

zdisk	`dd bs=8192 if=$(BOOTIMAGE) of=/dev/fd0`

zlilo	`if [ -f $(INSTALL_PATH)/vmlinuz ]; then mv $(INSTALL_PATH)/vmlinuz`
	`        $(INSTALL_PATH)/vmlinuz.old; fi`
	`if [ -f $(INSTALL_PATH)/System.map ]; then mv $(INSTALL_PATH)/System.map`
	`        $(INSTALL_PATH)/System.old; fi`
	`cat $(BOOTIMAGE) > $(INSTALL_PATH)/vmlinuz`
	`cp $(TOPDIR)/System.map $(INSTALL_PATH)/`
	`if [ -x /sbin/lilo ]; then /sbin/lilo; else /etc/lilo/install; fi`

install	`sh -x ./install.sh $(KERNELRELEASE) $(BOOTIMAGE) $(TOPDIR)/System.map` `        "$(INSTALL_PATH)"`

tools/build builds boot image *zImage* from {bootsect, setup, compressed/vmlinux.out}, or *bzImage* from {bbootsect, bsetup, compressed/bvmlinux,out}. `linux/Makefile` "export ROOT_DEV = CURRENT". Note that $(OBJCOPY) has been redefined by `linux/arch/i386/Makefile` in Section 2.3.

Table 3. Supporting targets in linux/arch/i386/boot/Makefile

Target: Prerequisites	Command
compressed/vmlinux: linux/vmlinux	**@$(MAKE) –C compressed vmlinux**
compressed/bvmlinux: linux/vmlinux	**@$(MAKE) –C compressed bvmlinux**
tools/build: tools/build.c	**$(HOSTCC) $(HOSTCFLAGS) –o $@ $< –I$(TOPDIR)/include** [a]
bootsect: bootsect.o	**$(LD) –Ttext 0x0 –s ––oformat binary bootsect.o** [b]
bootsect.o: bootsect.s	**$(AS) –o $@ $<**

bootsect.s: bootsect.S ...	**$(CPP) $(CPPFLAGS) −traditional $(SVGA_MODE) $(RAMDISK) $< −o $@**
bbootsect: bbootsect.o	**$(LD) −Ttext 0x0 −s −−oformat binary $< −o $@**
bbootsect.o: bbootsect.s	**$(AS) −o $@ $<**
bbootsect.s: bootsect.S ...	**$(CPP) $(CPPFLAGS) −D__BIG_KERNEL__ −traditional $(SVGA_MODE) $(RAMDISK) $< −o $@**
setup: setup.o	**$(LD) −Ttext 0x0 −s −−oformat binary −e begtext −o $@ $<**
setup.o: setup.s	**$(AS) −o $@ $<**
setup.s: setup.S video.S ...	**$(CPP) $(CPPFLAGS) −D__ASSEMBLY__ −traditional $(SVGA_MODE) $(RAMDISK) $< −o $@**
bsetup: bsetup.o	**$(LD) −Ttext 0x0 −s −−oformat binary −e begtext −o $@ $<**
bsetup.o: bsetup.s	**$(AS) −o $@ $<**
bsetup.s: setup.S video.S ...	**$(CPP) $(CPPFLAGS) −D__BIG_KERNEL__ −D__ASSEMBLY__ −traditional $(SVGA_MODE) $(RAMDISK) $< −o $@**
Notes:a. "$@" means target, "$<" means first prerequisite; Refer to GNU make: Automatic Variables; b. "−−oformat binary" asks for raw binary output, which is identical to the memory dump of the executable; Refer to Using LD, the GNU linker: Command Line Options.	

Note that it has "−D__BIG_KERNEL__" when compile `bootsect.S` to `bbootsect.s`, and `setup.S` to `bsetup.s`. They must be Place Independent Code (PIC), thus what "−Ttext" option is doesn't matter.

2.5. linux/arch/i386/boot/compressed/Makefile

This makefile handles image (de)compression mechanism.

It is good to separate (de)compression from bootstrap. This divide−and−conquer solution allows us to easily improve (de)compression mechanism or to adopt a new bootstrap method.

Directory `linux/arch/i386/boot/compressed/` contains two source files: `head.S` and `misc.c`.

Table 4. Targets in linux/arch/i386/boot/compressed/Makefile

Target	Command
vmlinux[a]	**$(LD) −Ttext 0x1000 −e startup_32 −o vmlinux head.o misc.o piggy.o**
bvmlinux	**$(LD) −Ttext 0x100000 −e startup_32 −o bvmlinux head.o misc.o piggy.o**
head.o	**$(CC) $(AFLAGS) −traditional −c head.S**
misc.o	`$(CC) $(CFLAGS) -DKBUILD_BASENAME=$(subst $(comma),_,$(subst -,_,$(*F)))` `        -c misc.c[b]`

piggy.o	`tmppiggy=_tmp_$$$$piggy; \`
	`rm -f $$tmppiggy $$tmppiggy.gz $$tmppiggy.lnk; \`
	`$(OBJCOPY) $(SYSTEM) $$tmppiggy; \`
	`gzip -f -9 < $$tmppiggy > $$tmppiggy.gz; \`
	`echo "SECTIONS { .data : { input_len = .; \`
	`        LONG(input_data_end - input_data) input_data = .; \`
	`        *(.data) input_data_end = .; }}" > $$tmppiggy.lnk; \`
	`$(LD) -r -o piggy.o -b binary $$tmppiggy.gz -b elf32-i386 \`

	`        -T $$tmppiggy.lnk; \` `rm -f $$tmppiggy $$tmppiggy.gz $$tmppiggy.lnk`
Notes:a. Target *vmlinux* here is different from that defined in `linux/Makefile`; b. "subst" is a MAKE function; Refer to GNU make: Functions for String Substitution and Analysis.	

`piggy.o` contains variable *input_len* and gzipped `linux/vmlinux`. *input_len* is at the beginning of `piggy.o`, and it is equal to the size of `piggy.o` excluding *input_len* itself. Refer to Using LD, the GNU linker: Section Data Expressions for "LONG(expression)" in *piggy.o* linker script.

To be exact, it is not `linux/vmlinux` itself (in ELF format) that is gzipped but its binary image, which is generated by **objcopy** command. Note that $(OBJCOPY) has been redefined by `linux/arch/i386/Makefile` in Section 2.3 to output raw binary using "−O binary" option.

When linking {*bootsect, setup*} or {*bbootsect, bsetup*}, $(LD) specifies "−−oformat binary" option to output them in binary format. When making *zImage* (or *bzImage*), $(OBJCOPY) generates an intermediate binary output from *compressed/vmlinux* (or *compressed/bvmlinux*) too. It is vital that all components in *zImage* or *bzImage* are in raw binary format, so that the image can run by itself without asking a loader to load and relocate it.

Both *vmlinux* and *bvmlinux* prepend `head.o` and `misc.o` before `piggy.o`, but they are linked against different start addresses (0x1000 vs 0x100000).

2.6. linux/arch/i386/tools/build.c

`linux/arch/i386/tools/build.c` is a host utility to generate *zImage* or *bzImage*.

In `linux/arch/i386/boot/Makefile`:

```
tools/build bootsect setup compressed/vmlinux.out $(ROOT_DEV) > zImage

tools/build −b bbootsect bsetup compressed/bvmlinux.out $(ROOT_DEV) > bzImage
```

"−b" means is_big_kernel, used to check whether system image is too big.

tools/build outputs the following components to stdout, which is redirected to *zImage* or *bzImage*:

1. bootsect or bbootsect: from `linux/arch/i386/boot/bootsect.S`, 512 bytes;
2. setup or bsetup: from `linux/arch/i386/boot/setup.S`, 4 sectors or more, sector aligned;
3. compressed/vmlinux.out or compressed/bvmlinux.out, including:

    a. head.o: from `linux/arch/i386/boot/compressed/head.S`;
    b. misc.o: from `linux/arch/i386/boot/compressed/misc.c`;
    c. piggy.o: from *input_len* and gzipped `linux/vmlinux`.

tools/build will change some contents of *bootsect* or *bbootsect* when outputting to stdout:

Table 5. Modification made by tools/build

Offset	Byte	Variable	Comment

1F1 (497)	1	setup_sectors	number of setup sectors, >=4
1F4 (500)	2	sys_size	system size in 16−bytes, little−endian
1FC (508)	1	minor_root	root dev minor
1FD (509)	1	major_root	root dev major

In the following chapters, compressed/vmlinux will be referred as *vmlinux* and compressed/bvmlinux as *bvmlinux*, if not confusing.

2.7. Reference

- Linux Kernel Makefiles: `linux/Documentation/kbuild/makefiles.txt`
- The Linux Kernel HOWTO
- GNU make
- Using LD, the GNU linker
- GNU Binary Utilities
- GNU Bash

3. linux/arch/i386/boot/bootsect.S

Given that we are booting up *bzImage*, which is composed of *bbootsect*, *bsetup* and *bvmlinux (head.o, misc.o, piggy.o)*, the first floppy sector, *bbootsect* (512 bytes), which is compiled from `linux/arch/i386/boot/bootsect.S`, is loaded by BIOS to 07C0:0. The reset of *bzImage* (*bsetup* and *bvmlinux*) has not been loaded yet.

3.1. Move Bootsect

```
SETUPSECTS      = 4                     /* default nr of setup-sectors */
BOOTSEG         = 0x07C0                /* original address of boot-sector */
INITSEG         = DEF_INITSEG  (0x9000) /* we move boot here - out of the way */
SETUPSEG        = DEF_SETUPSEG (0x9020) /* setup starts here */
SYSSEG          = DEF_SYSSEG   (0x1000) /* system loaded at 0x10000 (65536) */
SYSSIZE         = DEF_SYSSIZE  (0x7F00) /* system size: # of 16-byte clicks */
                                        /* to be loaded */
ROOT_DEV        = 0                     /* ROOT_DEV is now written by "build" */
SWAP_DEV        = 0                     /* SWAP_DEV is now written by "build" */

.code16
.text

///////////////////////////////////////////////////////////////////////////////
_start:
{
        // move ourself from 0x7C00 to 0x90000 and jump there.
        move BOOTSEG:0 to INITSEG:0 (512 bytes);
        goto INITSEG:go;
}
```

bbootsect has been moved to INITSEG:0 (0x9000:0). Now we can forget BOOTSEG.

3.2. Get Disk Parameters

```
///////////////////////////////////////////////////////////////////////////////
// prepare stack and disk parameter table
go:
{
        SS:SP = INITSEG:3FF4;   // put stack at INITSEG:0x4000-12
        /* 0x4000 is an arbitrary value >=
         *   length of bootsect + length of setup + room for stack;
         * 12 is disk parm size. */
        copy disk parameter (pointer in 0:0078) to INITSEG:3FF4 (12 bytes);
        // int1E: SYSTEM DATA - DISKETTE PARAMETERS
        patch sector count to 36 (offset 4 in parameter table, 1 byte);
        set disk parameter table pointer (0:0078, int1E) to INITSEG:3FF4;
}
```

Make sure SP is initialized immediately after SS register. The recommended method of modifying SS is to use "lss" instruction according to IA−32 Intel Architecture Software Developer's Manual (Vol.3. Ch.5.8.3. Masking Exceptions and Interrupts When Switching Stacks).

Stack operations, such as push and pop, will be OK now. First 12 bytes of disk parameter have been copied to INITSEG:3FF4.

```
/////////////////////////////////////////////////////////////////////////////////
// get disk drive parameters, specifically number of sectors/track.
        char disksizes[] = {36, 18, 15, 9};
        int sectors;
{
        SI = disksizes;                         // i = 0;
        do {
probe_loop:
                sectors = DS:[SI++];            // sectors = disksizes[i++];
                if (SI>=disksizes+4) break;     // if (i>=4) break;
                int13/AH=02h(AL=1, ES:BX=INITSEG:0200, CX=sectors, DX=0);
                // int13/AH=02h: DISK - READ SECTOR(S) INTO MEMORY
        } while (failed to read sectors);
}
```

"lodsb" loads a byte from DS:[SI] to AL and increases SI automatically.

The number of sectors per track has been saved in variable *sectors*.

3.3. Load Setup Code

bsetup (*setup_sects* sectors) will be loaded right after *bbootsect*, i.e. SETUPSEG:0. Note that INITSEG:0200==SETUPSEG:0 and *setup_sects* has been changed by **tools/build** to match *bsetup* size in Section 2.6.

```
/////////////////////////////////////////////////////////////////////////////////
got_sectors:
        word sread;             // sectors read for current track
        char setup_sects;       // overwritten by tools/build
{
        print out "Loading";
        /* int10/AH=03h(BH=0): VIDEO - GET CURSOR POSITION AND SIZE
         * int10/AH=13h(AL=1, BH=0, BL=7, CX=9, DH=DL=0, ES:BP=INITSEG:$msg1):
         *   VIDEO - WRITE STRING */

        // load setup-sectors directly after the moved bootblock (at 0x90200).
        SI = &sread;            // using SI to index sread, head and track
        sread = 1;              // the boot sector has already been read

        int13/AH=00h(DL=0);     // reset FDC

        BX = 0x0200;            // read bsetup right after bbootsect (512 bytes)
        do {
next_step:
                /* to prevent cylinder crossing reading,
                 *   calculate how many sectors to read this time */
                uint16 pushw_ax = AX = MIN(sectors-sread, setup_sects);
no_cyl_crossing:
                read_track(AL, ES:BX);          // AX is not modified
                // set ES:BX, sread, head and track for next read_track()
                set_next(AX);
                setup_sects -= pushw_ax;        // rest - for next step
        } while (setup_sects);
}
```

SI is set to the address of *sread* to index variables *sread*, *head* and *track*, as they are contiguous in memory. Check Section 3.6 for read_track() and set_next() details.

3.4. Load Compressed Image

bvmlinux (head.o, misc.o, piggy.o) will be loaded at 0x100000, *syssize**16 bytes.

```
/////////////////////////////////////////////////////////////////////////////
// load vmlinux/bvmlinux (head.o, misc.o, piggy.o)
{
        read_it(ES=SYSSEG);
        kill_motor();                           // turn off floppy drive motor
        print_nl();                             // print CR LF
}
```

Check Section 3.6 for read_it() details. If we are booting up *zImage*, *vmlinux* is loaded at 0x10000 (SYSSEG:0).

bzImage (bbootsect, bsetup, bvmlinux) is in the memory as a whole now.

3.5. Go Setup

```
/////////////////////////////////////////////////////////////////////////////
// check which root-device to use and jump to setup.S
        int root_dev;                           // overwritten by tools/build
{
        if (!root_dev) {
                switch (sectors) {
                case 15: root_dev = 0x0208;     // /dev/ps0 - 1.2Mb
                        break;
                case 18: root_dev = 0x021C;     // /dev/PS0 - 1.44Mb
                        break;
                case 36: root_dev = 0x0220;     // /dev/fd0H2880 - 2.88Mb
                        break;
                default: root_dev = 0x0200;     // /dev/fd0 - auto detect
                        break;
                }
        }

        // jump to the setup-routine loaded directly after the bootblock
        goto SETUPSEG:0;
}
```

It passes control to *bsetup*. See *linux/arch/i386/boot/setup.S:start* in Section 4.

3.6. Read Disk

The following functions are used to load *bsetup* and *bvmlinux* from disk. Note that *syssize* has been changed by **tools/build** in Section 2.6 too.

```
sread:  .word 0                         # sectors read of current track
head:   .word 0                         # current head
track:  .word 0                         # current track
/////////////////////////////////////////////////////////////////////////////
// load the system image at address SYSSEG:0
read_it(ES=SYSSEG)
        int syssize;                    /* system size in 16-bytes,
                                         *   overwritten by tools/build */
```

```
{
        if (ES & 0x0fff) die;           // not 64KB aligned

        BX = 0;
        for (;;) {
rp_read:
#ifdef __BIG_KERNEL__
                bootsect_helper(ES:BX);
                /* INITSEG:0220==SETUPSEG:0020 is bootsect_kludge,
                 *   which contains pointer SETUPSEG:bootsect_helper().
                 * This function initializes some data structures
                 *   when it is called for the first time,
                 *   and moves SYSSEG:0 to 0x100000, 64KB each time,
                 *   in the following calls.
                 * See Section 3.7. */
#else
                AX = ES - SYSSEG + ( BX >> 4);  // how many 16-bytes read
#endif
                if (AX > syssize) return;       // everything loaded
ok1_read:
                /* Get proper AL (sectors to read) for this time
                 *   to prevent cylinder crossing reading and BX overflow. */
                AX = sectors - sread;
                CX = BX + (AX << 9);            // 1 sector = 2^9 bytes
                if (CX overflow && CX!=0) {     // > 64KB
                        AX = (-BX) >> 9;
                }
ok2_read:
                read_track(AL, ES:BX);
                set_next(AX);
        }
}

/////////////////////////////////////////////////////////////////////////////
// read disk with parameters (sread, track, head)
read_track(AL sectors, ES:BX destination)
{
        for (;;) {
                printf(".");
                // int10/AH=0Eh: VIDEO - TELETYPE OUTPUT

                // set CX, DX according to (sread, track, head)
                DX = track;
                CX = sread + 1;
                CH = DL;

                DX = head;
                DH = DL;
                DX &= 0x0100;

                int13/AH=02h(AL, ES:BX, CX, DX);
                // int13/AH=02h: DISK - READ SECTOR(S) INTO MEMORY
                if (read disk success) return;
                // "addw $8, %sp" is to cancel previous 4 "pushw" operations.
bad_rt:
                print_all();            // print error code, AX, BX, CX and DX
                int13/AH=00h(DL=0);     // reset FDC
        }
}

/////////////////////////////////////////////////////////////////////////////
// set ES:BX, sread, head and track for next read_track()
```

```
set_next(AX sectors_read)
{
        CX = AX;                        // sectors read
        AX += sread;
        if (AX==sectors) {
                head = 1 ^ head;        // flap head between 0 and 1
                if (head==0) track++;
ok4_set:
                AX = 0;
        }
ok3_set:
        sread = AX;
        BX += CX && 9;
        if (BX overflow) {              // > 64KB
                ES += 0x1000;
                BX = 0;
        }
set_next_fn:
}
```

3.7. Bootsect Helper

setup.S:bootsect_helper() is only used by *bootsect.S:read_it()*.

Because *bbootsect* and *bsetup* are linked separately, they use offsets relative to their own code/data segments. We have to "call far" (lcall) for *bootsect_helper()* in different segment, and it must "return far" (lret) then. This results in CS change in calling, which makes CS!=DS, and we have to use segment modifier to specify variables in `setup.S`.

```
//////////////////////////////////////////////////////////////////////////////
// called by bootsect loader when loading bzImage
bootsect_helper(ES:BX)
        bootsect_es = 0;                // defined in setup.S
        type_of_loader = 0;             // defined in setup.S
{
        if (!bootsect_es) {             // called for the first time
                type_of_loader = 0x20;  // bootsect-loader, version 0
                AX = ES >> 4;
                *(byte*)(&bootsect_src_base+2) = AH;
                bootsect_es = ES;
                AX = ES - SYSSEG;
                return;
        }
bootsect_second:
        if (!BX) {                      // 64KB full
                // move from SYSSEG:0 to destination, 64KB each time
                int15/AH=87h(CX=0x8000, ES:SI=CS:bootsect_gdt);
                // int15/AH=87h: SYSTEM - COPY EXTENDED MEMORY
                if (failed to copy) {
                        bootsect_panic() {
                                prtstr("INT15 refuses to access high mem, "
                                        "giving up.");
bootsect_panic_loop:            goto bootsect_panic_loop;   // never return
                        }
                }
                ES = bootsect_es;       // reset ES to always point to 0x10000
                *(byte*)(&bootsect_dst_base+2)++;
        }
bootsect_ex:
```

```
        // have the number of moved frames (16-bytes) in AX
        AH = *(byte*)(&bootsect_dst_base+2) << 4;
        AL = 0;
}

//////////////////////////////////////////////////////////////////////////////
// data used by bootsect_helper()
bootsect_gdt:
        .word   0, 0, 0, 0
        .word   0, 0, 0, 0

bootsect_src:
        .word   0xffff

bootsect_src_base:
        .byte   0x00, 0x00, 0x01                # base = 0x010000
        .byte   0x93                            # typbyte
        .word   0                               # limit16,base24 =0

bootsect_dst:
        .word   0xffff

bootsect_dst_base:
        .byte   0x00, 0x00, 0x10                # base = 0x100000
        .byte   0x93                            # typbyte
        .word   0                               # limit16,base24 =0
        .word   0, 0, 0, 0                      # BIOS CS
        .word   0, 0, 0, 0                      # BIOS DS

bootsect_es:
        .word   0

bootsect_panic_mess:
        .string "INT15 refuses to access high mem, giving up."
```

Note that *type_of_loader* value is changed. It will be referenced in Section 4.3.

3.8. Miscellaneous

The rest are supporting functions, variables and part of "real−mode kernel header". Note that data is in .text segment as code, thus it can be properly initialized when loaded.

```
//////////////////////////////////////////////////////////////////////////////
// some small functions
print_all();  /* print error code, AX, BX, CX and DX */
print_nl();   /* print CR LF */
print_hex();  /* print the word pointed to by SS:BP in hexadecimal */
kill_motor()  /* turn off floppy drive motor */
{
#if 1
        int13/AH=00h(DL=0);     // reset FDC
#else
        outb(0, 0x3F2);         // outb(val, port)
#endif
}

//////////////////////////////////////////////////////////////////////////////
sectors:        .word 0
disksizes:      .byte 36, 18, 15, 9
```

```
msg1:           .byte 13, 10
                .ascii "Loading"
```

Bootsect trailer, which is a part of "real−mode kernel header", begins at offset 497.

```
.org 497
setup_sects:    .byte SETUPSECS         // overwritten by tools/build
root_flags:     .word ROOT_RDONLY
syssize:        .word SYSSIZE           // overwritten by tools/build
swap_dev:       .word SWAP_DEV
ram_size:       .word RAMDISK
vid_mode:       .word SVGA_MODE
root_dev:       .word ROOT_DEV          // overwritten by tools/build
boot_flag:      .word 0xAA55
```

This "header" must conform to the layout pattern in `linux/Documentation/i386/boot.txt`:

```
Offset  Proto   Name            Meaning
/Size
01F1/1  ALL     setup_sects     The size of the setup in sectors
01F2/2  ALL     root_flags      If set, the root is mounted readonly
01F4/2  ALL     syssize         DO NOT USE - for bootsect.S use only
01F6/2  ALL     swap_dev        DO NOT USE - obsolete
01F8/2  ALL     ram_size        DO NOT USE - for bootsect.S use only
01FA/2  ALL     vid_mode        Video mode control
01FC/2  ALL     root_dev        Default root device number
01FE/2  ALL     boot_flag       0xAA55 magic number
```

3.9. Reference

- THE LINUX/I386 BOOT PROTOCOL: `linux/Documentation/i386/boot.txt`
- IA−32 Intel Architecture Software Developer's Manual
- Ralf Brown's Interrupt List

As <IA−32 Intel Architecture Software Developer's Manual> is widely referenced in this document, I will call it "IA−32 Manual" for short.

4. linux/arch/i386/boot/setup.S

`setup.S` is responsible for getting the system data from the BIOS and putting them into appropriate places in system memory.

Other boot loaders, like GNU GRUB and LILO, can load *bzImage* too. Such boot loaders should load *bzImage* into memory and setup "real−mode kernel header", esp. *type_of_loader*, then pass control to *bsetup* directly. `setup.S` assumes:

- *bsetup* or *setup* may not be loaded at SETUPSEG:0, i.e. CS may not be equal to SETUPSEG when control is passed to `setup.S`;
- The first 4 sectors of *setup* are loaded right after *bootsect*. The reset may be loaded at SYSSEG:0, preceding *vmlinux*; This assumption does not apply to *bsetup*.

4.1. Header

```
/* Signature words to ensure LILO loaded us right */
#define SIG1    0xAA55
#define SIG2    0x5A5A

INITSEG  = DEF_INITSEG          # 0x9000, we move boot here, out of the way
SYSSEG   = DEF_SYSSEG           # 0x1000, system loaded at 0x10000 (65536).
SETUPSEG = DEF_SETUPSEG         # 0x9020, this is the current segment
                                # ... and the former contents of CS

DELTA_INITSEG = SETUPSEG - INITSEG      # 0x0020

.code16
.text

///////////////////////////////////////////////////////////////////////////////
start:
{
        goto trampoline();              // skip the following header
}

# This is the setup header, and it must start at %cs:2 (old 0x9020:2)
                .ascii  "HdrS"          # header signature
                .word   0x0203          # header version number (>= 0x0105)
                                        # or else old loadlin-1.5 will fail)
realmode_swtch: .word   0, 0            # default_switch, SETUPSEG
start_sys_seg:  .word   SYSSEG
                .word   kernel_version  # pointing to kernel version string
                                        # above section of header is compatible
                                        # with loadlin-1.5 (header v1.5). Don't
                                        # change it.
// kernel_version defined below
type_of_loader: .byte   0               # = 0, old one (LILO, Loadlin,
                                        #      Bootlin, SYSLX, bootsect...)
                                        # See Documentation/i386/boot.txt for
                                        # assigned ids
# flags, unused bits must be zero (RFU) bit within loadflags
loadflags:
LOADED_HIGH     = 1                     # If set, the kernel is loaded high
CAN_USE_HEAP    = 0x80                  # If set, the loader also has set
                                        # heap_end_ptr to tell how much
                                        # space behind setup.S can be used for
```

```
                                        # heap purposes.
                                        # Only the loader knows what is free
#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. We will move setup
                                        # to 0x90000 then just before jumping
                                        # into the kernel. However, only the
                                        # loader knows how much data behind
                                        # us also needs to be loaded.
code32_start:                           # here loaders can put a different
                                        # start address for 32-bit code.
#ifndef __BIG_KERNEL__
                .long   0x1000          #   0x1000 = default for zImage
#else
                .long   0x100000        # 0x100000 = default for big kernel
#endif
ramdisk_image:  .long   0               # address of loaded ramdisk image
                                        # Here the loader puts the 32-bit
                                        # address where it loaded the image.
                                        # This only will be read by the kernel.
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.
// modelist is at the end of .text section
pad1:           .word   0
cmd_line_ptr:   .long 0                 # (Header version 0x0202 or later)
                                        # If nonzero, a 32-bit pointer
                                        # to the kernel command line.
                                        # The command line should be
                                        # located between the start of
                                        # setup and the end of low
                                        # memory (0xa0000), or it may
                                        # get overwritten before it
                                        # gets read.  If this field is
                                        # used, there is no longer
                                        # anything magical about the
                                        # 0x90000 segment; the setup
                                        # can be located anywhere in
                                        # low memory 0x10000 or higher.
ramdisk_max:    .long __MAXMEM-1        # (Header version 0x0203 or later)
                                        # The highest safe address for
                                        # the contents of an initrd
```

The *__MAXMEM* definition in `linux/asm-i386/page.h`:

```
/*
 * A __PAGE_OFFSET of 0xC0000000 means that the kernel has
 * a virtual address space of one gigabyte, which limits the
 * amount of physical memory you can use to about 950MB.
 */
#define __PAGE_OFFSET           (0xC0000000)

/*
```

```
 * This much address space is reserved for vmalloc() and iomap()
 * as well as fixmap mappings.
 */
#define __VMALLOC_RESERVE       (128 << 20)

#define __MAXMEM                (-__PAGE_OFFSET-__VMALLOC_RESERVE)
```

It gives *__MAXMEM* = 1G − 128M.

The setup header must follow some layout pattern. Refer to `linux/Documentation/i386/boot.txt`:

```
Offset  Proto   Name            Meaning
/Size
0200/2  2.00+   jump            Jump instruction
0202/4  2.00+   header          Magic signature "HdrS"
0206/2  2.00+   version         Boot protocol version supported
0208/4  2.00+   realmode_swtch  Boot loader hook
020C/2  2.00+   start_sys       The load-low segment (0x1000) (obsolete)
020E/2  2.00+   kernel_version  Pointer to kernel version string
0210/1  2.00+   type_of_loader  Boot loader identifier
0211/1  2.00+   loadflags       Boot protocol option flags
0212/2  2.00+   setup_move_size Move to high memory size (used with hooks)
0214/4  2.00+   code32_start    Boot loader hook
0218/4  2.00+   ramdisk_image   initrd load address (set by boot loader)
021C/4  2.00+   ramdisk_size    initrd size (set by boot loader)
0220/4  2.00+   bootsect_kludge DO NOT USE - for bootsect.S use only
0224/2  2.01+   heap_end_ptr    Free memory after setup end
0226/2  N/A     pad1            Unused
0228/4  2.02+   cmd_line_ptr    32-bit pointer to the kernel command line
022C/4  2.03+   initrd_addr_max Highest legal initrd address
```

4.2. Check Code Integrity

As *setup* code may not be contiguous, we should check code integrity first.

```
/////////////////////////////////////////////////////////////////////////////
trampoline()
{
        start_of_setup();       // never return
        .space 1024;
}

/////////////////////////////////////////////////////////////////////////////
// check signature to see if all code loaded
start_of_setup()
{
        // Bootlin depends on this being done early, check bootlin:technic.doc
        int13/AH=15h(AL=0, DL=0x81);
        // int13/AH=15h: DISK - GET DISK TYPE

#ifdef SAFE_RESET_DISK_CONTROLLER
        int13/AH=0(AL=0, DL=0x80);
        // int13/AH=00h: DISK - RESET DISK SYSTEM
#endif

        DS = CS;
        // check signature at end of setup
        if (setup_sig1!=SIG1 || setup_sig2!=SIG2) {
                goto bad_sig;
```

```
        }
        goto goodsig1;
}

/////////////////////////////////////////////////////////////////////////////
// some small functions
prtstr();  /* print asciiz string at DS:SI */
prtsp2();  /* print double space */
prtspc();  /* print single space */
prtchr();  /* print ascii in AL */
beep();    /* print CTRL-G, i.e. beep */
```

Signature is checked to verify code integrity.

If signature is not found, the rest *setup* code may precede *vmlinux* at SYSSEG:0.

```
no_sig_mess: .string "No setup signature found ..."

goodsig1:
        goto goodsig;                           // make near jump

/////////////////////////////////////////////////////////////////////////////
// move the rest setup code from SYSSEG:0 to CS:0800
bad_sig()
        DELTA_INITSEG = 0x0020 (= SETUPSEG - INITSEG)
        SYSSEG = 0x1000
        word start_sys_seg = SYSSEG;            // defined in setup header
{
        DS = CS - DELTA_INITSEG;                // aka INITSEG
        BX = (byte)(DS:[497]);                  // i.e. setup_sects

        // first 4 sectors already loaded
        CX = (BX - 4) << 8;                     // rest code in word (2-bytes)
        start_sys_seg = (CX >> 3) + SYSSEG;     // real system code start
        move SYSSEG:0 to CS:0800 (CX*2 bytes);

        if (setup_sig1!=SIG1 || setup_sig2!=SIG2) {
no_sig:
                prtstr("No setup signature found ...");
no_sig_loop:
                hlt;
                goto no_sig_loop;
        }
}
```

"hlt" instruction stops instruction execution and places the processor in halt state. The processor generates a special bus cycle to indicate that halt mode has been entered. When an enabled interrupt (including NMI) is issued, the processor will resume execution after the "hlt" instruction, and the instruction pointer (CS:EIP), pointing to the instruction following the "hlt", will be saved to stack before the interrupt handler is called. Thus we need a "jmp" instruction after the "hlt" to put the processor back to halt state again.

The *setup* code has been moved to correct place. Variable *start_sys_seg* points to where real system code starts. If "bad_sig" does not happen, *start_sys_seg* remains SYSSEG.

4.3. Check Loader Type

Check if the loader is compatible with the image.

```
/////////////////////////////////////////////////////////////////////////////
good_sig()
        char loadflags;                 // in setup header
        char type_of_loader;            // in setup header
        LOADHIGH = 1
{
        DS = CS - DELTA_INITSEG;        // aka INITSEG
        if ( (loadflags & LOADHIGH) && !type_of_loader ) {
                // Nope, old loader tries to load big-kernel
                prtstr("Wrong loader, giving up...");
                goto no_sig_loop;       // defined above in bad_sig()
        }
}

loader_panic_mess: .string "Wrong loader, giving up..."
```

Note that *type_of_loader* has been changed to 0x20 by *bootsect_helper()* when it loads *bvmlinux*.

4.4. Get Memory Size

Try three different memory detection schemes to get the extended memory size (above 1M) in KB.

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 0−64M.

```
/////////////////////////////////////////////////////////////////////////////
// get memory size
loader_ok()
        E820NR  = 0x1E8
        E820MAP = 0x2D0
{
        // when entering this function, DS = CS-DELTA_INITSEG, aka INITSEG
        (long)DS:[0x1E0] = 0;

#ifndef STANDARD_MEMORY_BIOS_CALL
        (byte)DS:[0x1E8] = 0;                   // E820NR

        /* method E820H: see ACPI spec
         * the memory map from hell.  e820h returns memory classified into
         * a whole bunch of different types, and allows memory holes and
         * everything.  We scan through this memory map and build a list
         * of the first 32 memory areas, which we return at [E820MAP]. */
meme820:
        EBX = 0;
        DI = 0x02D0;                            // E820MAP
        do {
jmpe820:
                int15/EAX=E820h(EDX='SMAP', EBX, ECX=20, ES:DI=DS:DI);
                // int15/AX=E820h: GET SYSTEM MEMORY MAP
                if (failed || 'SMAP'!=EAX) break;
                // if (1!=DS:[DI+16]) continue; // not usable
good820:
                if (DS:[1E8]>=32) break;        // entry# > E820MAX
```

```
                DS:[0x1E8]++;                   // entry# ++;
                DI += 20;                       // adjust buffer for next
again820:
        } while (!EBX)                          // not finished
bail820:

        /* method E801H:
         * memory size is in 1k chunksizes, to avoid confusing loadlin.
         * we store the 0xe801 memory size in a completely different place,
         * because it will most likely be longer than 16 bits.
         * (use 1e0 because that's what Larry Augustine uses in his
         * alternative new memory detection scheme, and it's sensible
         * to write everything into the same place.) */
meme801:
        stc;            // to work around buggy BIOSes
        CX = DX = 0;
        int15/AX=E801h;
        /* int15/AX=E801h: GET MEMORY SIZE FOR >64M CONFIGURATIONS
         *   AX = extended memory between 1M and 16M, in K (max 3C00 = 15MB)
         *   BX = extended memory above 16M, in 64K blocks
         *   CX = configured memory 1M to 16M, in K
         *   DX = configured memory above 16M, in 64K blocks */
        if (failed) goto mem88;
        if (!CX && !DX) {
                CX = AX;
                DX = BX;
        }
e801usecxdx:
        (long)DS:[0x1E0] = ((EDX & 0xFFFF) << 6) + (ECX & 0xFFFF);      // in K
#endif

mem88:  // old traditional method
        int15/AH=88h;
        /* int15/AH=88h: SYSTEM - GET EXTENDED MEMORY SIZE
         *   AX = number of contiguous KB starting at absolute address 100000h */
        DS:[2] = AX;
}
```

4.5. Hardware Support

Check hardware support, like keyboard, video adapter, hard disk, MCA bus and pointing device.

```
{
        // set the keyboard repeat rate to the max
        int16/AX=0305h(BX=0);
        // int16/AH=03h: KEYBOARD - SET TYPEMATIC RATE AND DELAY

        /* Check for video adapter and its parameters and
         *   allow the user to browse video modes. */
        video();                        // see video.S

        // get hd0 and hd1 data
        copy hd0 data (*int41) to CS-DELTA_INITSEG:0080 (16 bytes);
        // int41: SYSTEM DATA - HARD DISK 0 PARAMETER TABLE ADDRESS
        copy hd1 data (*int46) to CS-DELTA_INITSEG:0090 (16 bytes);
        // int46: SYSTEM DATA - HARD DISK 1 PARAMETER TABLE ADDRESS
        // check if hd1 exists
        int13/AH=15h(AL=0, DL=0x81);
        // int13/AH=15h: DISK - GET DISK TYPE
        if (failed || AH!=03h) {        // AH==03h if it is a hard disk
```

```
no_disk1:
                clear CS-DELTA_INITSEG:0090 (16 bytes);
        }
is_disk1:

        // check for Micro Channel (MCA) bus
        CS-DELTA_INITSEG:[0xA0] = 0;    // set table length to 0
        int15/AH=C0h;
        /* int15/AH=C0h: SYSTEM - GET CONFIGURATION
         *   ES:BX = ROM configuration table */
        if (failed) goto no_mca;
        move ROM configuration table (ES:BX) to CS-DELTA_INITSEG:00A0;
        // CX = (table length<14)? CX:16;    first 16 bytes only
no_mca:

        // check for PS/2 pointing device
        CS-DELTA_INITSEG:[0x1FF] = 0;   // default is no pointing device
        int11h();
        // int11h: BIOS - GET EQUIPMENT LIST
        if (AL & 0x04) {                // mouse installed
                DS:[0x1FF] = 0xAA;
        }
}
```

4.6. APM Support

Check BIOS APM support.

```
#if defined(CONFIG_APM) || defined(CONFIG_APM_MODULE)
{
        DS:[0x40] = 0;                  // version = 0 means no APM BIOS
        int15/AX=5300h(BX=0);
        // int15/AX=5300h: Advanced Power Management v1.0+ - INSTALLATION CHECK
        if (failed || 'PM'!=BX || !(CX & 0x02)) goto done_apm_bios;
        // (CX & 0x02) means 32 bit is supported
        int15/AX=5304h(BX=0);
        // int15/AX=5304h: Advanced Power Management v1.0+ - DISCONNECT INTERFACE
        EBX = CX = DX = ESI = DI = 0;
        int15/AX=5303h(BX=0);
        /* int15/AX=5303h: Advanced Power Management v1.0+
         *   - CONNECT 32-BIT PROTMODE INTERFACE */
        if (failed) {
no_32_apm_bios:                         // I moved label no_32_apm_bios here
                DS:[0x4C] &= ~0x0002;   // remove 32 bit support bit
                goto done_apm_bios;
        }
        DS:[0x42] = AX, 32-bit code segment base address;
        DS:[0x44] = EBX, offset of entry point;
        DS:[0x48] = CX, 16-bit code segment base address;
        DS:[0x4A] = DX, 16-bit data segment base address;
        DS:[0x4E] = ESI, APM BIOS code segment length;
        DS:[0x52] = DI, APM BIOS data segment length;
        int15/AX=5300h(BX=0);           // check again
        // int15/AX=5300h: Advanced Power Management v1.0+ - INSTALLATION CHECK
        if (success &&  'PM'==BX) {
                DS:[0x40] = AX, APM version;
                DS:[0x4C] = CX, APM flags;
        } else {
apm_disconnect:
                int15/AX=5304h(BX=0);
```

```
                /* int15/AX=5304h: Advanced Power Management v1.0+
                 *   - DISCONNECT INTERFACE */
        }
done_apm_bios:
}
#endif
```

4.7. Prepare for Protected Mode

```
// call mode switch
{
        if (realmode_swtch) {
                realmode_swtch();               // mode switch hook
        } else {
rmodeswtch_normal:
                default_switch() {
                        cli;                    // no interrupts allowed
                        outb(0x80, 0x70);       // disable NMI
                }
        }
rmodeswtch_end:
}

// relocate code if necessary
{
        (long)code32 = code32_start;
        if (!(loadflags & LOADED_HIGH)) {       // low loaded zImage
                // 0x0100 <= start_sys_seg < CS-DELTA_INITSEG
do_move0:
                AX = 0x100;
                BP = CS - DELTA_INITSEG;        // aka INITSEG
                BX = start_sys_seg;
do_move:
                move system image from (start_sys_seg:0 .. CS-DELTA_INITSEG:0)
                        to 0100:0;              // move 0x1000 bytes each time
        }
end_move:
```

Note that *code32_start* is initialized to 0x1000 for *zImage*, or 0x100000 for *bzImage*. The *code32* value will be used in passing control to `linux/arch/i386/boot/compressed/head.S` in Section 4.9. If we boot up *zImage*, it relocates *vmlinux* to 0100:0; If we boot up *bzImage*, *bvmlinux* remains at start_sys_seg:0. The relocation address must match the "−Ttext" option in `linux/arch/i386/boot/compressed/Makefile`. See Section 2.5.

Then it will relocate code from CS−DELTA_INITSEG:0 (*bbootsect* and *bsetup*) to INITSEG:0, if necessary.

```
        DS = CS;                // aka SETUPSEG
        // Check whether we need to be downward compatible with version <=201
        if (!cmd_line_ptr && 0x20!=type_of_loader && SETUPSEG!=CS) {
                cli;            // as interrupt may use stack when we are moving
                // store new SS in DX
                AX = CS - DELTA_INITSEG;
                DX = SS;
                if (DX>=AX) {   // stack frame will be moved together
                        DX = DX + INITSEG - AX; // i.e. SS-CS+SETUPSEG
                }
move_self_1:
                /* move CS-DELTA_INITSEG:0 to INITSEG:0 (setup_move_size bytes)
```

```
                 *   in two steps in order not to overwrite code on CS:IP
                 * move up (src < dest) but downward ("std") */
                move CS-DELTA_INITSEG:move_self_here+0x200
                  to INITSEG:move_self_here+0x200,
                  setup_move_size-(move_self_here+0x200) bytes;
                // INITSEG:move_self_here+0x200 == SETUPSEG:move_self_here
                goto SETUPSEG:move_self_here;   // CS=SETUPSEG now
move_self_here:
                move CS-DELTA_INITSEG:0 to INITSEG:0,
                  move_self_here+0x200 bytes;   // I mean old CS before goto
                DS = SETUPSEG;
                SS = DX;
        }
end_move_self:
}
```

Note again, *type_of_loader* has been changed to 0x20 by *bootsect_helper()* when it loads *bvmlinux*.

4.8. Enable A20

For A20 problem and solution, refer to A20 – a pain from the past.

```
        A20_TEST_LOOPS          =  32   # Iterations per wait
        A20_ENABLE_LOOPS        = 255   # Total loops to try
{
#if defined(CONFIG_MELAN)
        // Enable A20. AMD Elan bug fix.
        outb(0x02, 0x92);               // outb(val, port)
a20_elan_wait:
        while (!a20_test());            // test not passed
        goto a20_done;
#endif

a20_try_loop:
        // First, see if we are on a system with no A20 gate.
a20_none:
        if (a20_test()) goto a20_done;  // test passed

        // Next, try the BIOS (INT 0x15, AX=0x2401)
a20_bios:
        int15/AX=2401h;
        // Int15/AX=2401h: SYSTEM - later PS/2s - ENABLE A20 GATE
        if (a20_test()) goto a20_done;  // test passed

        // Try enabling A20 through the keyboard controller
a20_kbc:
        empty_8042();
        if (a20_test()) goto a20_done;  // test again in case BIOS delayed
        outb(0xD1, 0x64);               // command write
        empty_8042();
        outb(0xDF, 0x60);               // A20 on
        empty_8042();
        // wait until a20 really *is* enabled
a20_kbc_wait:
        CX = 0;
a20_kbc_wait_loop:
        do {
                if (a20_test()) goto a20_done;  // test passed
        } while (--CX)
```

```
        // Final attempt: use "configuration port A"
        outb((inb(0x92) | 0x02) & 0xFE, 0x92);
        // wait for configuration port A to take effect
a20_fast_wait:
        CX = 0;
a20_fast_wait_loop:
        do {
                if (a20_test()) goto a20_done;  // test passed
        } while (--CX)

        // A20 is still not responding. Try frobbing it again.
        if (--a20_tries) goto a20_try_loop;
        prtstr("linux: fatal error: A20 gate not responding!");
a20_die:
        hlt;
        goto a20_die;
}

a20_tries:
        .byte   A20_ENABLE_LOOPS                // i.e. 255
a20_err_msg:
        .ascii  "linux: fatal error: A20 gate not responding!"
        .byte   13, 10, 0
```

For I/O port operations, take a look at related reference materials in Section 4.11.

4.9. Switch to Protected Mode

To ensure code compatibility with all 32−bit IA−32 processors, perform the following steps to switch to protected mode:

1. Prepare GDT with a null descriptor in the first GDT entry, one code segment descriptor and one data segment descriptor;
2. Disable interrupts, including maskable hardware interrupts and NMI;
3. Load the base address and limit of the GDT to GDTR register, using "lgdt" instruction;
4. Set PE flag in CR0 register, using "mov cr0" (Intel 386 and up) or "lmsw" instruction (for compatibility with Intel 286);
5. Immediately execute a far "jmp" or a far "call" instruction.

The stack can be placed in a normal read/write data segment, so no dedicated descriptor is required.

```
a20_done:
{
        lidt    idt_48;         // load idt with 0, 0;

        // convert DS:gdt to a linear ptr
        *(long*)(gdt_48+2) = DS << 4 + &gdt;
        lgdt    gdt_48;

        // reset coprocessor
        outb(0, 0xF0);
        delay();
        outb(0, 0xF1);
        delay();

        // reprogram the interrupts
        outb(0xFF, 0xA1);       // mask all interrupts
```

```
        delay();
        outb(0xFB, 0x21);       // mask all irq's but irq2 which is cascaded

        // protected mode!
        AX = 1;
        lmsw ax;                // machine status word, bit 0 thru 15 of CR0
                                // only affects PE, MP, EM & TS flags
        goto flush_instr;

flush_instr:
        BX = 0;                                 // flag to indicate a boot
        ESI = (CS - DELTA_INITSEG) << 4;        // pointer to real-mode code
        /* NOTE: For high loaded big kernels we need a
         * 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) */

        // goto __KERNEL_CS:[(uint32*)code32]; */
        .byte   0x66, 0xea
code32: .long   0x1000          // overwritten in Section 4.7
        .word   __KERNEL_CS     // segment 0x10
        // see linux/arch/i386/boot/compressed/head.S:startup_32
}
```

The far "jmp" instruction (0xea) updates CS register. The contents of the remaining segment registers (DS, SS, ES, FS and GS) should be reloaded later. The operand−size prefix (0x66) is used to enforce "jmp" to be executed upon the 32−bit operand *code32*. For operand−size prefix details, check IA−32 Manual (Vol.1. Ch.3.6. Operand−size and Address−size Attributes, and Vol.3. Ch.17. Mixing 16−bit and 32−bit Code).

Control is passed to *linux/arch/i386/boot/compressed/head.S:startup_32*. For *zImage*, it is at address 0x1000; For *bzImage*, it is at 0x100000. See Section 5.

ESI points to the memory area of collected system data. It is used to pass parameters from the 16−bit real mode code of the kernel to the 32−bit part. See `linux/Documentation/i386/zero-page.txt` for details.

For mode switching details, refer to IA−32 Manual Vol.3. (Ch.9.8. Software Initialization for Protected−Mode Operation, Ch.9.9.1. Switching to Protected Mode, and Ch.17.4. Transferring Control Among Mixed−Size Code Segments).

4.10. Miscellaneous

The rest are supporting functions and variables.

```
/* macros created by linux/Makefile targets:
 *   include/linux/compile.h and include/linux/version.h */
kernel_version: .ascii  UTS_RELEASE
                .ascii  " ("
                .ascii  LINUX_COMPILE_BY
                .ascii  "@"
                .ascii  LINUX_COMPILE_HOST
                .ascii  ") "
```

```
                .ascii  UTS_VERSION
                .byte   0

//////////////////////////////////////////////////////////////////////////////
default_switch() { cli; outb(0x80, 0x70); } /* disable interrupts and NMI */
bootsect_helper(ES:BX); /* see Section 3.7 */

//////////////////////////////////////////////////////////////////////////////
a20_test()
{
        FS = 0;
        GS = 0xFFFF;
        CX = A20_TEST_LOOPS;                    // i.e. 32
        AX = FS:[0x200];
        do {
a20_test_wait:
                FS:[0x200] = ++AX;
                delay();
        } while (AX==GS:[0x210] && --CX);
        return (AX!=GS[0x210]);
        // ZF==0 (i.e. NZ/NE, a20_test!=0) means test passed
}

//////////////////////////////////////////////////////////////////////////////
// check that the keyboard command queue is empty
empty_8042()
{
        int timeout = 100000;

        for (;;) {
empty_8042_loop:
                if (!--timeout) return;
                delay();
                inb(0x64, &AL);                 // 8042 status port
                if (AL & 1) {                   // has output
                        delay();
                        inb(0x60, &AL);         // read it
no_output:      } else if (!(AL & 2)) return;   // no input either
        }
}

//////////////////////////////////////////////////////////////////////////////
// read the CMOS clock, return the seconds in AL, used in video.S
gettime()
{
        int1A/AH=02h();
        /* int1A/AH=02h: TIME - GET REAL-TIME CLOCK TIME
         * DH = seconds in BCD */
        AL = DH & 0x0F;
        AH = DH >> 4;
        aad;
}

//////////////////////////////////////////////////////////////////////////////
delay() { outb(AL, 0x80); }                     // needed after doing I/O

// Descriptor table
gdt:
        .word   0, 0, 0, 0                      # dummy
        .word   0, 0, 0, 0                      # unused
        // segment 0x10, __KERNEL_CS
        .word   0xFFFF                          # 4Gb - (0x100000*0x1000 = 4Gb)
```

```
        .word   0                               # base address = 0
        .word   0x9A00                          # code read/exec
        .word   0x00CF                          # granularity = 4096, 386
                                                #  (+5th nibble of limit)
        // segment 0x18, __KERNEL_DS
        .word   0xFFFF                          # 4Gb - (0x100000*0x1000 = 4Gb)
        .word   0                               # base address = 0
        .word   0x9200                          # data read/write
        .word   0x00CF                          # granularity = 4096, 386
                                                #  (+5th nibble of limit)
idt_48:
        .word   0                               # idt limit = 0
        .word   0, 0                            # idt base = 0L
/* [gdt_48] should be 0x0800 (2048) to match the comment,
 *   like what Linux 2.2.22 does. */
gdt_48:
        .word   0x8000                          # gdt limit=2048,
                                                #  256 GDT entries
        .word   0, 0                            # gdt base (filled in later)

#include "video.S"

// signature at the end of setup.S:
{
setup_sig1:     .word   SIG1                    // 0xAA55
setup_sig2:     .word   SIG2                    // 0x5A5A
modelist:
}
```

Video setup and detection code in `video.S`:

```
ASK_VGA = 0xFFFD  // defined in linux/include/asm-i386/boot.h
///////////////////////////////////////////////////////////////////////////////
video()
{
        pushw DS;               // use different segments
        FS = DS;
        DS = ES = CS;
        GS = 0;
        cld;
        basic_detect();         // basic adapter type testing (EGA/VGA/MDA/CGA)
#ifdef CONFIG_VIDEO_SELECT
        if (FS:[0x01FA]!=ASK_VGA) {     // user selected video mode
                mode_set();
                if (failed) {
                        prtstr("You passed an undefined mode number.\n");
                        mode_menu();
                }
        } else {
vid2:           mode_menu();
        }
vid1:
#ifdef CONFIG_VIDEO_RETAIN
        restore_screen();               // restore screen contents
#endif /* CONFIG_VIDEO_RETAIN */
#endif /* CONFIG_VIDEO_SELECT */
        mode_params();                  // store mode parameters
        popw ds;                        // restore original DS
}
```

/* TODO: video() details */

4.11. Reference

- A20 − a pain from the past
- Real−time Programming Appendix A: Complete I/O Port List
- IA−32 Intel Architecture Software Developer's Manual
- Summary of empty_zero_page layout (kernel point of view): `linux/Documentation/i386/zero-page.txt`

5. linux/arch/i386/boot/compressed/head.S

We are in *bvmlinux* now! With the help of *misc.c:decompress_kernel()*, we are going to decompress *piggy.o* to get the resident kernel image `linux/vmlinux`.

This file is of pure 32−bit startup code. Unlike previous two files, it has no ".code16" statement in the source file. Refer to Using as: Writing 16−bit Code for details.

5.1. Decompress Kernel

The segment base addresses in segment descriptors (which correspond to segment selector __KERNEL_CS and __KERNEL_DS) are equal to 0; therefore, the logical address offset (in segment:offset format) will be equal to its linear address if either of these segment selectors is used. For *zImage*, CS:EIP is at logical address 10:1000 (linear address 0x1000) now; for *bzImage*, 10:100000 (linear address 0x100000).

As paging is not enabled, linear address is identical to physical address. Check IA−32 Manual (Vol.1. Ch.3.3. Memory Organization, and Vol.3. Ch.3. Protected−Mode Memory Management) and Linux Device Drivers: Memory Management in Linux for address issue.

It comes from `setup.S` that BX=0 and ESI=INITSEG<<4.

```
.text
///////////////////////////////////////////////////////////////////////////////
startup_32()
{
        cld;
        cli;
        DS = ES = FS = GS = __KERNEL_DS;
        SS:ESP = *stack_start;  // end of user_stack[], defined in misc.c
        // all segment registers are reloaded after protected mode is enabled

        // check that A20 really IS enabled
        EAX = 0;
        do {
1:              DS:[0] = ++EAX;
        } while (DS:[0x100000]==EAX);

        EFLAGS = 0;
        clear BSS;                              // from _edata to _end

        struct moveparams mp;                   // subl $16,%esp
        if (!decompress_kernel(&mp, ESI)) {     // return value in AX
                restore ESI from stack;
                EBX = 0;
                goto __KERNEL_CS:100000;
                // see linux/arch/i386/kernel/head.S:startup_32
        }

        /*
         * We come here, if we were loaded high.
         * We need to move the move-in-place routine down to 0x1000
         * and then start it with the buffer addresses in registers,
         * which we got from the stack.
         */
3:      move move_rountine_start..move_routine_end to 0x1000;
        // move_routine_start & move_routine_end are defined below
```

```
        // prepare move_routine_start() parameters
        EBX = real mode pointer;        // ESI value passed from setup.S
        ESI = mp.low_buffer_start;
        ECX = mp.lcount;
        EDX = mp.high_buffer_star;
        EAX = mp.hcount;
        EDI = 0x100000;
        cli;                    // make sure we don't get interrupted
        goto __KERNEL_CS:1000;  // move_routine_start();
}

/* Routine (template) for moving the decompressed kernel in place,
 * if we were high loaded. This _must_ PIC-code ! */
///////////////////////////////////////////////////////////////////////////////
move_routine_start()
{
        move mp.low_buffer_start to 0x100000, mp.lcount bytes,
          in two steps: (lcount >> 2) words + (lcount & 3) bytes;
        move/append mp.high_buffer_start, ((mp.hcount + 3) >> 2) words
        // 1 word == 4 bytes, as I mean 32-bit code/data.

        ESI = EBX;              // real mode pointer, as that from setup.S
        EBX = 0;
        goto __KERNEL_CS:100000;
        // see linux/arch/i386/kernel/head.S:startup_32()
move_routine_end:
}
```

For the meaning of "je 1b" and "jnz 3f", refer to Using as: Local Symbol Names.

Didn't find *_edata* and *_end* definitions? No problem, they are defined in the "internal linker script". Without −T (−−script=) option specified, **ld** uses this builtin script to link *compressed/bvmlinux*. Use "**ld −−verbose**" to display this script, or check Appendix B. *Internal Linker Script*.

Refer to Using LD, the GNU linker: Command Line Options for −T (−−script=), −L (−−library−path=) and −−verbose option description. "**man ld**" and "**info ld**" may help too.

piggy.o has been unzipped and control is passed to __KERNEL_CS:100000, i.e. *linux/arch/i386/kernel/head.S:startup_32()*. See Section 6.

```
#define LOW_BUFFER_START      0x2000
#define LOW_BUFFER_MAX       0x90000
#define HEAP_SIZE             0x3000
///////////////////////////////////////////////////////////////////////////////
asmlinkage int decompress_kernel(struct moveparams *mv, void *rmode)
|-- setup real_mode(=rmode), vidmem, vidport, lines and cols;
|-- if (is_zImage) setup_normal_output_buffer() {
|       output_data      = 0x100000;
|       free_mem_end_ptr = real_mode;
|   } else (is_bzImage) setup_output_buffer_if_we_run_high(mv) {
|       output_data      = LOW_BUFFER_START;
|       low_buffer_end   = MIN(real_mode, LOW_BUFFER_MAX) & ~0xfff;
|       low_buffer_size  = low_buffer_end - LOW_BUFFER_START;
|       free_mem_end_ptr = &end + HEAP_SIZE;
|       // get mv->low_buffer_start and mv->high_buffer_start
|       mv->low_buffer_start = LOW_BUFFER_START;
|       /* To make this program work, we must have
|        *   high_buffer_start > &end+HEAP_SIZE;
```

```
|         * As we will move low_buffer from LOW_BUFFER_START to 0x100000
|         *   (max low_buffer_size bytes) finally, we should have
|         *   high_buffer_start > 0x100000+low_buffer_size; */
|        mv->high_buffer_start = high_buffer_start
|            = MAX(&end+HEAP_SIZE, 0x100000+low_buffer_size);
|        mv->hcount =  0 if (0x100000+low_buffer_size >  &end+HEAP_SIZE);
|                   = -1 if (0x100000+low_buffer_size <= &end+HEAP_SIZE);
|        /* mv->hcount==0 : we need not move high_buffer later,
|         *   as it is already at 0x100000+low_buffer_size.
|         * Used by close_output_buffer_if_we_run_high() below. */
|    }
|-- makecrc();          // create crc_32_tab[]
|   puts("Uncompressing Linux... ");
|-- gunzip();
|   puts("Ok, booting the kernel.\n");
|-- if (is_bzImage) close_output_buffer_if_we_run_high(mv) {
|        // get mv->lcount and mv->hcount
|        if (bytes_out > low_buffer_size) {
|            mv->lcount = low_buffer_size;
|            if (mv->hcount)
|                mv->hcount = bytes_out - low_buffer_size;
|        } else {
|            mv->lcount = bytes_out;
|            mv->hcount = 0;
|        }
|    }
`-- return is_bzImage;  // return value in AX
```

end is defined in the "internal linker script" too.

decompress_kernel() has an "asmlinkage" modifer. In `linux/include/linux/linkage.h`:

```
#ifdef __cplusplus
#define CPP_ASMLINKAGE extern "C"
#else
#define CPP_ASMLINKAGE
#endif

#if defined __i386__
#define asmlinkage CPP_ASMLINKAGE __attribute__((regparm(0)))
#elif defined __ia64__
#define asmlinkage CPP_ASMLINKAGE __attribute__((syscall_linkage))
#else
#define asmlinkage CPP_ASMLINKAGE
#endif
```

Macro "asmlinkage" will force the compiler to pass all function arguments on the stack, in case some optimization method may try to change this convention. Check Using the GNU Compiler Collection (GCC): Declaring Attributes of Functions (regparm) and Kernelnewbies FAQ: What is asmlinkage for more details.

5.2. gunzip()

decompress_kernel() calls *gunzip() −> inflate()*, which are defined in `linux/lib/inflate.c`, to decompress resident kernel image to low buffer (pointed by *output_data*) and high buffer (pointed by *high_buffer_start*, for *bzImage* only).

The gzip file format is specified in RFC 1952.

Table 6. gzip file format

Component	Meaning	Byte	Comment
ID1	IDentification 1	1	31 (0x1f, \037)
ID2	IDentification 2	1	139 (0x8b, \213) [a]
CM	Compression Method	1	8 – denotes the "deflate" compression method
FLG	FLaGs	1	0 for most cases
MTIME	Modification TIME	4	modification time of the original file
XFL	eXtra FLags	1	2 – compressor used maximum compression, slowest algorithm [b]
OS	Operating System	1	3 – Unix
extra fields	–	–	variable length, field indicated by FLG [c]
compressed blocks	–	–	variable length
CRC32	–	4	CRC value of the uncompressed data
ISIZE	Input SIZE	4	the size of the uncompressed input data modulo 2^32
Notes:a. ID2 value can be 158 (0x9e, \236) for gzip 0.5; b. XFL value 4 – compressor used fastest algorithm; c. FLG bit 0, FTEXT, does not indicate any "extra field".			

We can use this file format knowledge to find out the beginning of gzipped `linux/vmlinux`.

```
[root@localhost boot]# hexdump -C /boot/vmlinuz-2.4.20-28.9 | grep '1f 8b 08 00'
00004c50  1f 8b 08 00 01 f6 e1 3f  02 03 ec 5d 7d 74 14 55  |.......?...]}t.U|
[root@localhost boot]# hexdump -C /boot/vmlinuz-2.4.20-28.9 -s 0x4c40 -n 64
00004c40  00 80 0b 00 00 fc 21 00  68 00 00 00 1e 01 11 00  |......!.h.......|
00004c50  1f 8b 08 00 01 f6 e1 3f  02 03 ec 5d 7d 74 14 55  |.......?...]}t.U|
00004c60  96 7f d5 a9 d0 1d 4d ac  56 93 35 ac 01 3a 9c 6a  |......M.V.5..:.j|
00004c70  4d 46 5c d3 7b f8 48 36  c9 6c 84 f0 25 88 20 9f  |MF\.{.H6.l..%. .|
00004c80
[root@localhost boot]# hexdump -C /boot/vmlinuz-2.4.20-28.9 | tail -n 4
00114d40  bd 77 66 da ce 6f 3d d6  33 5c 14 a2 9f 7e fa e9  |.wf..o=.3\...~..|
00114d50  a7 9f 7e fa ff 57 3f 00  00 00 00 00 d8 bc ab ea  |..~..W?.........|
00114d60  44 5d 76 d1 fd 03 33 58  c2 f0 00 51 27 00        |D]v...3X...Q'.|
00114d6e
```

We can see that the gzipped file begins at 0x4c50 in the above example. The four bytes before "1f 8b 08 00" is *input_len* (0x0011011e, in little endian), and 0x4c50+0x0011011e=0x114d6e equals to the size of *bzImage* (`/boot/vmlinuz-2.4.20-28.9`).

```
static uch *inbuf;           /* input buffer */
static unsigned insize = 0;  /* valid bytes in inbuf */
static unsigned inptr = 0;   /* index of next byte to be processed in inbuf */
///////////////////////////////////////////////////////////////////////////////
static int gunzip(void)
{
        Check input buffer for {ID1, ID2, CM}, must be
                {0x1f, 0x8b, 0x08} (normal case), or
                {0x1f, 0x9e, 0x08} (for gzip 0.5);
        Check FLG (flag byte), must not set bit 1, 5, 6 and 7;
        Ignore {MTIME, XFL, OS};
        Handle optional structures, which correspond to FLG bit 2, 3 and 4;
```

```
        inflate();              // handle compressed blocks
        Validate {CRC32, ISIZE};
}
```

When *get_byte()*, defined in `linux/arch/i386/boot/compressed/misc.c`, is called for the first time, it calls *fill_inbuf()* to setup input buffer *inbuf*=*input_data* and *insize*=*input_len*. Symbol *input_data* and *input_len* are defined in *piggy.o* linker script. See Section 2.5.

5.3. inflate()

```
// some important definitions in misc.c
#define WSIZE 0x8000            /* Window size must be at least 32k,
                                 * and a power of two */
static uch window[WSIZE];       /* Sliding window buffer */
static unsigned outcnt = 0;     /* bytes in output buffer */

// linux/lib/inflate.c
#define wp outcnt
#define flush_output(w) (wp=(w),flush_window())
STATIC unsigned long bb;        /* bit buffer */
STATIC unsigned bk;             /* bits in bit buffer */
STATIC unsigned hufts;          /* track memory usage */
static long free_mem_ptr = (long)&end;
///////////////////////////////////////////////////////////////////////////////
STATIC int inflate()
{
        int e;                  /* last block flag */
        int r;                  /* result code */
        unsigned h;             /* maximum struct huft's malloc'ed */
        void *ptr;

        wp = bb = bk = 0;

        // inflate compressed blocks one by one
        do {
                hufts = 0;
                gzip_mark() { ptr = free_mem_ptr; };
                if ((r = inflate_block(&e)) != 0) {
                        gzip_release() { free_mem_ptr = ptr; };
                        return r;
                }
                gzip_release() { free_mem_ptr = ptr; };
                if (hufts > h)
                h = hufts;
        } while (!e);

        /* Undo too much lookahead. The next read will be byte aligned so we
         * can discard unused bits in the last meaningful byte. */
        while (bk >= 8) {
                bk -= 8;
                inptr--;
        }

        /* write the output window window[0..outcnt-1] to output_data,
         * update output_ptr/output_data, crc and bytes_out accordingly, and
         * reset outcnt to 0. */
        flush_output(wp);

        /* return success */
        return 0;
```

```
}
```

free_mem_ptr is used in *misc.c:malloc()* for dynamic memory allocation. Before inflating each compressed block, *gzip_mark()* saves the value of *free_mem_ptr*; After inflation, *gzip_release()* will restore this value. This is how it "*free()*" the memory allocated in *inflate_block()*.

Gzip uses Lempel−Ziv coding (LZ77) to compress files. The compressed data format is specified in RFC 1951. *inflate_block()* will inflate compressed blocks, which can be treated as a bit sequence.

The data structure of each compressed block is outlined below:

```
BFINAL (1 bit)
    0  - not the last block
    1  - the last block
BTYPE  (2 bits)
    00 - no compression
        remaining bits until the byte boundary;
        LEN      (2 bytes);
        NLEN     (2 bytes, the one's complement of LEN);
        data     (LEN bytes);
    01 - compressed with fixed Huffman codes
        {
        literal  (7-9 bits, represent code 0..287, excluding 256);
                     // See RFC 1951, table in Paragraph 3.2.6.
        length   (0-5 bits if literal > 256, represent length 3..258);
                     // See RFC 1951, 1st alphabet table in Paragraph 3.2.5.
        data     (of literal bytes if literal < 256);
        distance (5 plus 0-13 extra bits if literal == 257..285, represent
                         distance 1..32768);
                     /* See RFC 1951, 2nd alphabet table in Paragraph 3.2.5,
                      *   but statement in Paragraph 3.2.6. */
                     /* Move backward "distance" bytes in the output stream,
                      * and copy "length" bytes */
        }*           // can be of multiple instances
        literal  (7 bits, all 0, literal == 256, means end of block);
    10 - compressed with dynamic Huffman codes
        HLIT     (5 bits, # of Literal/Length codes - 257, 257-286);
        HDIST    (5 bits, # of Distance codes - 1,         1-32);
        HCLEN    (4 bits, # of Code Length codes - 4,      4 - 19);
        Code Length sequence    ((HCLEN+4)*3 bits)
        /* The following two alphabet tables will be decoded using
         *   the Huffman decoding table which is generated from
         *   the preceeding Code Length sequence. */
        Literal/Length alphabet (HLIT+257 codes)
        Distance alphabet       (HDIST+1 codes)
        // Decoding tables will be built from these alphpabet tables.
        /* The following is similar to that of fixed Huffman codes portion,
         *   except that they use different decoding tables. */
        {
        literal/length
                 (variable length, depending on Literal/Length alphabet);
        data     (of literal bytes if literal < 256);
        distance (variable length if literal == 257..285, depending on
                         Distance alphabet);
        }*           // can be of multiple instances
        literal  (literal value 256, which means end of block);
    11 - reserved (error)
```

Note that data elements are packed into bytes starting from Least−Significant Bit (LSB) to Most−Significant Bit (MSB), while Huffman codes are packed starting with MSB. Also note that *literal* value 286−287 and *distance* codes 30−31 will never actually occur.

With the above data structure in mind and RFC 1951 by hand, it is not too hard to understand *inflate_block()*. Refer to related paragraphs in RFC 1951 for Huffman coding and alphabet table generation.

For more details, refer to `linux/lib/inflate.c`, gzip source code (many in−line comments) and related reference materials.

5.4. Reference

- Using as
- Using LD, the GNU linker
- IA−32 Intel Architecture Software Developer's Manual
- The gzip home page
- gzip (freshmeat.net)
- RFC 1951: DEFLATE Compressed Data Format Specification version 1.3
- RFC 1952: GZIP file format specification version 4.3

6. linux/arch/i386/kernel/head.S

Resident kernel image `linux/vmlinux` is in place finally! It requires two inputs:

- *ESI*, to indicate where the 16−bit real mode code is located, aka INITSEG<<4;
- *BX*, to indicate which CPU is running, 0 means BSP, other values for AP.

ESI points to the parameter area from the 16−bit real mode code, which will be copied to *empty_zero_page* later. ESI is only valid for BSP.

BSP (BootStrap Processor) and APs (Application Processors) are Intel terminologies. Check IA−32 Manual (Vol.3. Ch.7.5. Multiple−Processor (MP) Initialization) and MultiProcessor Specification for MP intialization issue.

From a software point of view, in a multiprocessor system, BSP and APs share the physical memory but use their own register sets. BSP runs the kernel code first, setups OS execution enviornment and triggers APs to run over it too. AP will be sleeping until BSP kicks it.

6.1. Enable Paging

```
.text
///////////////////////////////////////////////////////////////////////////////
startup_32()
{
        /* set segments to known values */
        cld;
        DS = ES = FS = GS = __KERNEL_DS;

#ifdef CONFIG_SMP
#define cr4_bits mmu_cr4_features-__PAGE_OFFSET
        /* long mmu_cr4_features defined in linux/arch/i386/kernel/setup.c
         * __PAGE_OFFSET = 0xC0000000, i.e. 3G */

        // AP with CR4 support (> Intel 486) will copy CR4 from BSP
        if (BX && cr4_bits) {
                // turn on paging options (PSE, PAE, ...)
                CR4 |= cr4_bits;
        } else
#endif
        {
                /* only BSP initializes page tables (pg0..empty_zero_page-1)
                 *   pg0 at .org 0x2000
                 *   empty_zero_page at .org 0x4000
                 *   total (0x4000-0x2000)/4 = 0x0800 entries */
                pg0 = {
                        0x00000007,             // 7 = PRESENT + RW + USER
                        0x00001007,             // 0x1000 = 4096 = 4K
                        0x00002007,
                        ...
                pg1:    0x00400007,
                        ...
                        0x007FF007              // total 8M
                empty_zero_page:
                };
        }
```

Why do we have to add "−__PAGE_OFFSET" when referring a kernel symbol, for example, like *pg0*?

In `linux/arch/i386/vmlinux.lds`, we have:

```
  . = 0xC0000000 + 0x100000;
  _text = .;                    /* Text and read-only data */
  .text : {
        *(.text)
...
```

As *pg0* is at offset 0x2000 of section *.text* in `linux/arch/i386/kernel/head.o`, which is the first file to be linked for `linux/vmlinux`, it will be at offset 0x2000 in output section *.text*. Thus it will be located at address 0xC0000000+0x100000+0x2000 after linking.

```
[root@localhost boot]# nm --defined /boot/vmlinux-2.4.20-28.9 | grep 'startup_32
\|mmu_cr4_features\|pg0\|\<empty_zero_page\>' | sort
c0100000 t startup_32
c0102000 T pg0
c0104000 T empty_zero_page
c0376404 B mmu_cr4_features
```

In protected mode without paging enabled, linear address will be mapped directly to physical address. "movl $pg0−__PAGE_OFFSET,%edi" will set EDI=0x102000, which is equal to the physical address of *pg0* (as `linux/vmlinux` is relocated to 0x100000). Without this "−PAGE_OFFSET" scheme, it will access physical address 0xC0102000, which will be wrong and probably beyond RAM space.

mmu_cr4_features is in *.bss* section and is located at physical address 0x376404 in the above example.

After page tables are initialized, paging can be enabled.

```
        // set page directory base pointer, physical address
        CR3 = swapper_pg_dir - __PAGE_OFFSET;
        // paging enabled!
        CR0 |= 0x80000000;      // set PG bit
        goto 1f;                // flush prefetch-queue
1:
        EAX = &1f;              // address following the next instruction
        goto *(EAX);            // relocate EIP
1:
        SS:ESP = *stack_start;
```

Page directory *swapper_pg_dir* (see definition in Section 6.5), together with page tables *pg0* and *pg1*, defines that both linear address 0..8M−1 and 3G..3G+8M−1 are mapped to physical address 0..8M−1. We can access kernel symbols without "−__PAGE_OFFSET" from now on, because kernel space (resides in linear address >=3G) will be correctly mapped to its physical addresss after paging is enabled.

"lss stack_start,%esp" (SS:ESP = *stack_start) is the first example to reference a symbol without "−PAGE_OFFSET", which sets up a new stack. For BSP, the stack is at the end of *init_task_union*. For AP, *stack_start.esp* has been redefined by *linux/arch/i386/kernel/smpboot.c:do_boot_cpu()* to be "(void *) (1024 + PAGE_SIZE + (char *)idle)" in Section 8.2.

For paging mechanism and data structures, refer to IA−32 Manual Vol.3. (Ch.3.7. Page Translation Using 32−Bit Physical Addressing, Ch.9.8.3. Initializing Paging, Ch.9.9.1. Switching to Protected Mode, and Ch.18.26.3. Enabling and Disabling Paging).

6.2. Get Kernel Parameters

```
#define OLD_CL_MAGIC_ADDR       0x90020
#define OLD_CL_MAGIC            0xA33F
#define OLD_CL_BASE_ADDR        0x90000
#define OLD_CL_OFFSET           0x90022
#define NEW_CL_POINTER          0x228   /* Relative to real mode data */

#ifdef CONFIG_SMP
        if (BX) {
                EFLAGS = 0;             // AP clears EFLAGS
        } else
#endif
        {
                // Initial CPU cleans BSS
                clear BSS;              // i.e. __bss_start .. _end
                setup_idt() {
                        /* idt_table[256] defined in arch/i386/kernel/traps.c
                         *   located in section .data.idt
                        EAX = __KERNEL_CS << 16 + ignore_int;
                        DX = 0x8E00;    // interrupt gate, dpl = 0, present
                        idt_table[0..255] = {EAX, EDX};
                }
                EFLAGS = 0;
                /*
                 * Copy bootup parameters out of the way. First 2kB of
                 * _empty_zero_page is for boot parameters, second 2kB
                 * is for the command line.
                 */
                move *ESI (real-mode header) to empty_zero_page, 2KB;
                clear empty_zero_page+2K, 2KB;
                ESI = empty_zero_page[NEW_CL_POINTER];
                if (!ESI) {             // 32-bit command line pointer
                        if (OLD_CL_MAGIC==(uint16)[OLD_CL_MAGIC_ADDR]) {
                                ESI = [OLD_CL_BASE_ADDR]
                                      + (uint16)[OLD_CL_OFFSET];
                                move *ESI to empty_zero_page+2K, 2KB;
                        }
                } else {                // valid in 2.02+
                        move *ESI to empty_zero_page+2K, 2KB;
                }
        }
}
```

For BSP, kernel parameters are copied from memory pointed by *ESI* to *empty_zero_page*. Kernel command line will be copied to *empty_zero_page+2K* if applicable.

6.3. Check CPU Type

Refer to IA−32 Manual Vol.1. (Ch.13. Processor Identification and Feature Determination) on how to identify processor type and processor features.

```
struct cpuinfo_x86;     // see include/asm-i386/processor.h
struct cpuinfo_x86 boot_cpu_data;       // see arch/i386/kernel/setup.c

#define CPU_PARAMS      SYMBOL_NAME(boot_cpu_data)
#define X86             CPU_PARAMS+0
#define X86_VENDOR      CPU_PARAMS+1
```

```
#define X86_MODEL       CPU_PARAMS+2
#define X86_MASK        CPU_PARAMS+3
#define X86_HARD_MATH   CPU_PARAMS+6
#define X86_CPUID       CPU_PARAMS+8
#define X86_CAPABILITY  CPU_PARAMS+12
#define X86_VENDOR_ID   CPU_PARAMS+28

checkCPUtype:
{
        X86_CPUID = -1;                 // no CPUID

        X86 = 3;                        // at least 386
        save original EFLAGS to ECX;
        flip AC bit (0x40000) in EFLAGS;
        if (AC bit not changed) goto is386;

        X86 = 4;                        // at least 486
        flip ID bit (0X200000) in EFLAGS;
        restore original EFLAGS;        // for AC & ID flags
        if (ID bit can not be changed) goto is486;

        // get CPU info
        CPUID(EAX=0);
        X86_CPUID = EAX;
        X86_VENDOR_ID = {EBX, EDX, ECX};
        if (!EAX) goto is486;

        CPUID(EAX=1);
        CL = AL;
        X86 = AH & 0x0f;                // family
        X86_MODEL = (AL & 0xf0) >> 4;   // model
        X86_MASK = CL & 0x0f;           // stepping id
        X86_CAPABILITY = EDX;           // feature
```

Refer to IA−32 Manual Vol.3. (Ch.9.2. x87 FPU Initialization, and Ch.18.14. x87 FPU) on how to setup x87 FPU.

```
is486:
        // save PG, PE, ET and set AM, WP, NE, MP
        EAX = (CR0 & 0x80000011) | 0x50022;
        goto 2f;                        // skip "is386:" processing
is386:
        restore original EFLAGS from ECX;
        // save PG, PE, ET and set MP
        EAX = (CR0 & 0x80000011) | 0x02;

        /* ET: Extension Type (bit 4 of CR0).
         * In the Intel 386 and Intel 486 processors, this flag indicates
         * support of Intel 387 DX math coprocessor instructions when set.
         * In the Pentium 4, Intel Xeon, and P6 family processors,
         * this flag is hardcoded to 1.
         *     -- IA-32 Manual Vol.3. Ch.2.5. Control Registers (p.2-14) */

2:      CR0 = EAX;
        check_x87() {
                /* We depend on ET to be correct.
                 * This checks for 287/387. */
                X86_HARD_MATH = 0;
                clts;                   // CR0.TS = 0;
                fninit;                 // Init FPU;
                fstsw AX;               // AX = ST(0);
```

```
                if (AL) {
                        CR0 ^= 0x04;    // no coprocessor, set EM
                } else {
                        ALIGN
1:                      X86_HARD_MATH = 1;
                        /* IA-32 Manual Vol.3. Ch.18.14.7.14. FSETPM Instruction
                         * inform 287 that processor is in protected mode
                         * 287 only, ignored by 387 */
                        fsetpm;
                }
        }
}
```

Macro ALIGN, defined in `linux/include/linux/linkage.h`, specifies 16−bytes alignment and fill value 0x90 (opcode for NOP). See also Using as: Assembler Directives for the meaning of directive *.align*.

6.4. Go Start Kernel

```
        ready:  .byte 0;        // global variable
{
        ready++;                // how many CPUs are ready
        lgdt gdt_descr;         // use new descriptor table in safe place
        lidt idt_descr;
        goto __KERNEL_CS:$1f;   // reload segment registers after "lgdt"
1:      DS = ES = FS = GS = __KERNEL_DS;
#ifdef CONFIG_SMP
        SS = __KERNEL_DS;       // reload segment only
#else
        SS:ESP = *stack_start;  /* end of init_task_union, defined
                                 *   in linux/arch/i386/kernel/init_task.c */
#endif
        EAX = 0;
        lldt AX;
        cld;

#ifdef CONFIG_SMP
        if (1!=ready) {         // not first CPU
                initialize_secondary();
                // see linux/arch/i386/kernel/smpboot.c
        } else
#endif
        {
                start_kernel(); // see linux/init/main.c
        }
L6:     goto L6;
}
```

The first CPU (BSP) will call *linux/init/main.c:start_kernel()* and the others (AP) will call *linux/arch/i386/kernel/smpboot.c:initialize_secondary()*. See *start_kernel()* in Section 7 and *initialize_secondary()* in Section 8.4.

init_task_union happens to be the task struct for the first process, "idle" process (pid=0), whose stack grows from the tail of *init_task_union*. The following is the code related to *init_task_union*:

```
ENTRY(stack_start)
        .long init_task_union+8192;
        .long __KERNEL_DS;
```

```
#ifndef INIT_TASK_SIZE
# define INIT_TASK_SIZE 2048*sizeof(long)
#endif

union task_union {
        struct task_struct task;
        unsigned long stack[INIT_TASK_SIZE/sizeof(long)];
};

/* INIT_TASK is used to set up the first task table, touch at
 * your own risk! Base=0, limit=0x1fffff (=2MB) */
union task_union init_task_union
        __attribute__((__section__(".data.init_task"))) =
                { INIT_TASK(init_task_union.task) };
```

init_task_union is for BSP "idle" process. Don't confuse it with "init" process, which will be mentioned in Section 7.2.

6.5. Miscellaneous

```
////////////////////////////////////////////////////////////////////////////////
// default interrupt "handler"
ignore_int() { printk("Unknown interrupt\n"); iret; }

/*
 * The interrupt descriptor table has room for 256 idt's,
 * the global descriptor table is dependent on the number
 * of tasks we can have..
 */
#define IDT_ENTRIES     256
#define GDT_ENTRIES     (__TSS(NR_CPUS))

.globl SYMBOL_NAME(idt)
.globl SYMBOL_NAME(gdt)

        ALIGN
        .word 0
idt_descr:
        .word IDT_ENTRIES*8-1           # idt contains 256 entries
SYMBOL_NAME(idt):
        .long SYMBOL_NAME(idt_table)

        .word 0
gdt_descr:
        .word GDT_ENTRIES*8-1
SYMBOL_NAME(gdt):
        .long SYMBOL_NAME(gdt_table)

/*
 * This is initialized to create an identity-mapping at 0-8M (for bootup
 * purposes) and another mapping of the 0-8M area at virtual address
 * PAGE_OFFSET.
 */
.org 0x1000
ENTRY(swapper_pg_dir)   // "ENTRY" defined in linux/include/linux/linkage.h
        .long 0x00102007
        .long 0x00103007
        .fill BOOT_USER_PGD_PTRS-2,4,0
        /* default: 766 entries */
        .long 0x00102007
```

```
        .long 0x00103007
        /* default: 254 entries */
        .fill BOOT_KERNEL_PGD_PTRS-2,4,0

/*
 * The page tables are initialized to only 8MB here - the final page
 * tables are set up later depending on memory size.
 */
.org 0x2000
ENTRY(pg0)

.org 0x3000
ENTRY(pg1)

/*
 * empty_zero_page must immediately follow the page tables ! (The
 * initialization loop counts until empty_zero_page)
 */
.org 0x4000
ENTRY(empty_zero_page)

/*
 * Real beginning of normal "text" segment
 */
.org 0x5000
ENTRY(stext)
ENTRY(_stext)

////////////////////////////////////////////////////////////////////////////////
/*
 * This starts the data section. Note that the above is all
 * in the text section because it has alignment requirements
 * that we cannot fulfill any other way.
 */
.data

ALIGN
/*
 * This contains typically 140 quadwords, depending on NR_CPUS.
 *
 * NOTE! Make sure the gdt descriptor in head.S matches this if you
 * change anything.
 */
ENTRY(gdt_table)
        .quad 0x0000000000000000        /* NULL descriptor */
        .quad 0x0000000000000000        /* not used */
        .quad 0x00cf9a000000ffff        /* 0x10 kernel 4GB code at 0x00000000 */
        .quad 0x00cf92000000ffff        /* 0x18 kernel 4GB data at 0x00000000 */
        .quad 0x00cffa000000ffff        /* 0x23 user   4GB code at 0x00000000 */
        .quad 0x00cff2000000ffff        /* 0x2b user   4GB data at 0x00000000 */
        .quad 0x0000000000000000        /* not used */
        .quad 0x0000000000000000        /* not used */
        /*
         * The APM segments have byte granularity and their bases
         * and limits are set at run time.
         */
        .quad 0x0040920000000000        /* 0x40 APM set up for bad BIOS's */
        .quad 0x00409a0000000000        /* 0x48 APM CS    code */
        .quad 0x00009a0000000000        /* 0x50 APM CS 16 code (16 bit) */
        .quad 0x0040920000000000        /* 0x58 APM DS    data */
        .fill NR_CPUS*4,8,0             /* space for TSS's and LDT's */
```

Macro ALIGN, before *idt_descr* and *gdt_table*, is for performance consideration.

6.6. Reference

- IA−32 Intel Architecture Software Developer's Manual
- MultiProcessor Specification
- Using as
- GNU Binary Utilities

7. linux/init/main.c

I felt guilty writing this chapter as there are too many documents about it, if not more than enough. *start_kernel()* supporting functions are changed from version to version, as they depend on OS component internals, which are being improved all the time. I may not have the time for frequent document updates, so I decided to keep this chapter as simple as possible.

7.1. start_kernel()

```
///////////////////////////////////////////////////////////////////////////////
asmlinkage void __init start_kernel(void)
{
        char * command_line;
        extern char saved_command_line[];
/*
 * Interrupts are still disabled. Do necessary setups, then enable them
 */
        lock_kernel();
        printk(linux_banner);

        /* Memory Management in Linux, esp. for setup_arch()
         * Linux-2.4.4 MM Initialization */
        setup_arch(&command_line);
        printk("Kernel command line: %s\n", saved_command_line);

        /* linux/Documentation/kernel-parameters.txt
         * The Linux BootPrompt-HowTo */
        parse_options(command_line);

        trap_init() {
#ifdef CONFIG_EISA
                if (isa_readl(0x0FFFD9) == 'E'+('I'<<8)+('S'<<16)+('A'<<24))
                        EISA_bus = 1;
#endif
#ifdef CONFIG_X86_LOCAL_APIC
                init_apic_mappings();
#endif
                set_xxxx_gate(x, &func);    // setup gates
                cpu_init();
        }
        init_IRQ();
        sched_init();
        softirq_init() {
                for (int i=0; i<32: i++)
                        tasklet_init(bh_task_vec+i, bh_action, i);
                open_softirq(TASKLET_SOFTIRQ, tasklet_action, NULL);
                open_softirq(HI_SOFTIRQ, tasklet_hi_action, NULL);
        }
        time_init();

        /*
         * 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.
         */
        console_init();
#ifdef CONFIG_MODULES
        init_modules();
```

```
#endif
        if (prof_shift) {
                unsigned int size;
                /* only text is profiled */
                prof_len = (unsigned long) &_etext - (unsigned long) &_stext;
                prof_len >>= prof_shift;
                size = prof_len * sizeof(unsigned int) + PAGE_SIZE-1;
                prof_buffer = (unsigned int *) alloc_bootmem(size);
        }

        kmem_cache_init();
        sti();

        // BogoMips mini-Howto
        calibrate_delay();

        // linux/Documentation/initrd.txt
#ifdef CONFIG_BLK_DEV_INITRD
        if (initrd_start && !initrd_below_start_ok &&
                        initrd_start < min_low_pfn << PAGE_SHIFT) {
                printk(KERN_CRIT "initrd overwritten (0x%08lx < 0x%08lx) - "
                    "disabling it.\n",initrd_start,min_low_pfn << PAGE_SHIFT);
                initrd_start = 0;
        }
#endif

        mem_init();
        kmem_cache_sizes_init();
        pgtable_cache_init();

        /*
         * For architectures that have highmem, num_mappedpages represents
         * the amount of memory the kernel can use.  For other architectures
         * it's the same as the total pages.  We need both numbers because
         * some subsystems need to initialize based on how much memory the
         * kernel can use.
         */
        if (num_mappedpages == 0)
                num_mappedpages =  num_physpages;

        fork_init(num_mempages);
        proc_caches_init();
        vfs_caches_init(num_physpages);
        buffer_init(num_physpages);
        page_cache_init(num_physpages);
#if defined(CONFIG_ARCH_S390)
        ccwcache_init();
#endif
        signals_init();
#ifdef CONFIG_PROC_FS
        proc_root_init();
#endif
#if defined(CONFIG_SYSVIPC)
        ipc_init();
#endif
        check_bugs();
        printk("POSIX conformance testing by UNIFIX\n");

        /*
         *      We count on the initial thread going ok
         *      Like idlers init is an unlocked kernel thread, which will
         *      make syscalls (and thus be locked).
```

```
         */
        smp_init() {
#ifndef CONFIG_SMP
#     ifdef CONFIG_X86_LOCAL_APIC
                APIC_init_uniprocessor();
#     else
                do { } while (0);
#     endif
#else
                /* Check Section 8.2. */
#endif
        }

        rest_init() {
                // init process, pid = 1
                kernel_thread(init, NULL, CLONE_FS | CLONE_FILES | CLONE_SIGNAL);
                unlock_kernel();
                current->need_resched = 1;
                // idle process, pid = 0
                cpu_idle();     // never return
        }
}
```

start_kernel() calls *rest_init()* to spawn an "init" process and become "idle" process itself.

7.2. init()

"Init" process:

```
////////////////////////////////////////////////////////////////////////////////
static int init(void * unused)
{
        lock_kernel();
        do_basic_setup();

        prepare_namespace();

        /*
         * Ok, we have completed the initial bootup, and
         * we're essentially up and running. Get rid of the
         * initmem segments and start the user-mode stuff..
         */
        free_initmem();
        unlock_kernel();

        if (open("/dev/console", O_RDWR, 0) < 0)        // stdin
                printk("Warning: unable to open an initial console.\n");

        (void) dup(0);                                  // stdout
        (void) dup(0);                                  // stderr

        /*
         * We try each of these until one succeeds.
         *
         * The Bourne shell can be used instead of init if we are
         * trying to recover a really broken machine.
         */

        if (execute_command)
                execve(execute_command,argv_init,envp_init);
```

```
        execve("/sbin/init",argv_init,envp_init);
        execve("/etc/init",argv_init,envp_init);
        execve("/bin/init",argv_init,envp_init);
        execve("/bin/sh",argv_init,envp_init);
        panic("No init found.  Try passing init= option to kernel.");
}
```

Refer to "**man init**" or SysVinit for further information on user−mode "init" process.

7.3. cpu_idle()

"Idle" process:

```
/*
 * The idle thread. There's no useful work to be
 * done, so just try to conserve power and have a
 * low exit latency (ie sit in a loop waiting for
 * somebody to say that they'd like to reschedule)
 */
void cpu_idle (void)
{
        /* endless idle loop with no priority at all */
        init_idle();
        current->nice = 20;
        current->counter = -100;

        while (1) {
                void (*idle)(void) = pm_idle;
                if (!idle)
                        idle = default_idle;
                while (!current->need_resched)
                        idle();
                schedule();
                check_pgt_cache();
        }
}

///////////////////////////////////////////////////////////////////////////////
void __init init_idle(void)
{
        struct schedule_data * sched_data;
        sched_data = &aligned_data[smp_processor_id()].schedule_data;

        if (current != &init_task && task_on_runqueue(current)) {
                printk("UGH! (%d:%d) was on the runqueue, removing.\n",
                        smp_processor_id(), current->pid);
                del_from_runqueue(current);
        }
        sched_data->curr = current;
        sched_data->last_schedule = get_cycles();
        clear_bit(current->processor, &wait_init_idle);
}

///////////////////////////////////////////////////////////////////////////////
void default_idle(void)
{
        if (current_cpu_data.hlt_works_ok && !hlt_counter) {
                __cli();
                if (!current->need_resched)
                        safe_halt();
```

```
                else
                        __sti();
        }
}

/* defined in linux/include/asm-i386/system.h */
#define __cli()                 __asm__ __volatile__("cli": : :"memory")
#define __sti()                 __asm__ __volatile__("sti": : :"memory")

/* used in the idle loop; sti takes one instruction cycle to complete */
#define safe_halt()             __asm__ __volatile__("sti; hlt": : :"memory")
```

CPU will resume code execution with the instruction following "hlt" on the return from an interrupt handler.

7.4. Reference

- Linux Kernel 2.4 Internals
- Kerneldoc
- LDP HOWTO–INDEX
- Linux Device Drivers, 2nd Edition

8. SMP Boot

There are a few SMP related macros, like *CONFIG_SMP, CONFIG_X86_LOCAL_APIC, CONFIG_X86_IO_APIC*, *CONFIG_MULTIQUAD* and *CONFIG_VISWS*. I will ignore code that requires *CONFIG_MULTIQUAD* or *CONFIG_VISWS*, which most people don't care (if not using IBM high−end multiprocessor server or SGI Visual Workstation).

BSP executes *start_kernel() −> smp_init() −> smp_boot_cpus() −> do_boot_cpu() −> wakeup_secondary_via_INIT()* to trigger APs. Check MultiProcessor Specification and IA−32 Manual Vol.3 (Ch.7. Multile−Processor Management, and Ch.8. Advanced Programmable Interrupt Controller) for technical details.

8.1. Before smp_init()

Before calling *smp_init()*, *start_kernel()* did something to setup SMP environment:

```
start_kernel()
|-- setup_arch()
|    |-- parse_cmdline_early();  // SMP looks for "noht" and "acpismp=force"
|    |    `-- /* "noht" disables HyperThreading (2 logical cpus per Xeon) */
|    |        if (!memcmp(from, "noht", 4)) {
|    |            disable_x86_ht = 1;
|    |            set_bit(X86_FEATURE_HT, disabled_x86_caps);
|    |        }
|    |        /* "acpismp=force" forces parsing and use of the ACPI SMP table */
|    |        else if (!memcmp(from, "acpismp=force", 13))
|    |            enable_acpi_smp_table = 1;
|    |-- setup_memory();         // reserve memory for MP configuration table
|    |    |-- reserve_bootmem(PAGE_SIZE, PAGE_SIZE);
|    |    `-- find_smp_config();
|    |        `-- find_intel_smp();
|    |            `-- smp_scan_config();
|    |                |-- set flag smp_found_config
|    |                |-- set MP floating pointer mpf_found
|    |                `-- reserve_bootmem(mpf_found, PAGE_SIZE);
|    |-- if (disable_x86_ht) {   // if HyperThreading feature disabled
|    |        clear_bit(X86_FEATURE_HT, &boot_cpu_data.x86_capability[0]);
|    |        set_bit(X86_FEATURE_HT, disabled_x86_caps);
|    |        enable_acpi_smp_table = 0;
|    |    }
|    |-- if (test_bit(X86_FEATURE_HT, &boot_cpu_data.x86_capability[0]))
|    |        enable_acpi_smp_table = 1;
|    |-- smp_alloc_memory();
|    |    `-- /* reserve AP processor's real-mode code space in low memory */
|    |        trampoline_base = (void *) alloc_bootmem_low_pages(PAGE_SIZE);
|    `-- get_smp_config();     /* get boot-time MP configuration */
|        |-- config_acpi_tables();
|        |    |-- memset(&acpi_boot_ops, 0, sizeof(acpi_boot_ops));
|        |    |-- acpi_boot_ops[ACPI_APIC] = acpi_parse_madt;
|        |    `-- /* Set have_acpi_tables to indicate using
|        |         * MADT in the ACPI tables; Use MPS tables if failed. */
|        |        if (enable_acpi_smp_table && !acpi_tables_init())
|        |            have_acpi_tables = 1;
|        |-- set pic_mode
|        |    /* =1, if the IMCR is present and PIC Mode is implemented;
|        |     * =0, otherwise Virtual Wire Mode is implemented. */
|        |-- save local APIC address in mp_lapic_addr
```

```
|       `-- scan for MP configuration table entries, like
|              MP_PROCESSOR, MP_BUS, MP_IOAPIC, MP_INTSRC and MP_LINTSRC.
|-- trap_init();
|   `-- init_apic_mappings();   // setup PTE for APIC
|       |-- /* If no local APIC can be found then set up a fake all
|       |     * zeroes page to simulate the local APIC and another
|       |     * one for the IO-APIC. */
|       |    if (!smp_found_config && detect_init_APIC()) {
|       |        apic_phys = (unsigned long) alloc_bootmem_pages(PAGE_SIZE);
|       |        apic_phys = __pa(apic_phys);
|       |    } else
|       |        apic_phys = mp_lapic_addr;
|       |-- /* map local APIC address,
|       |     *   mp_lapic_addr (0xfee00000) in most case,
|       |     *   to linear address FIXADDR_TOP (0xffffe000) */
|       |    set_fixmap_nocache(FIX_APIC_BASE, apic_phys);
|       |-- /* Fetch the APIC ID of the BSP in case we have a
|       |     * default configuration (or the MP table is broken). */
|       |    if (boot_cpu_physical_apicid == -1U)
|       |        boot_cpu_physical_apicid = GET_APIC_ID(apic_read(APIC_ID));
|       `-- // map IOAPIC address to uncacheable linear address
|            set_fixmap_nocache(idx, ioapic_phys);
|       // Now we can use linear address to access APIC space.
|-- init_IRQ();
|   |-- init_ISA_irqs();
|   |   |-- /* An initial setup of the virtual wire mode. */
|   |   |    init_bsp_APIC();
|   |   `-- init_8259A(auto_eoi=0);
|   `-- setup SMP/APIC interrupt handlers, esp. IPI.
`-- mem_init();
    `-- /* delay zapping low mapping entries for SMP: zap_low_mappings() */
```

IPI (InterProcessor Interrupt), CPU−to−CPU interrupt through local APIC, is the mechanism used by BSP to trigger APs.

Be aware that "one local APIC per CPU is required" in an MP−compliant system. Processors do not share APIC local units address space (physical address 0xFEE00000 − 0xFEEFFFFF), but will share APIC I/O units (0xFEC00000 − 0xFECFFFFF). Both address spaces are uncacheable.

8.2. smp_init()

BSP calls *start_kernel() −> smp_init() −> smp_boot_cpus()* to setup data structures for each CPU and activate the rest APs.

```
///////////////////////////////////////////////////////////////////////////////
static void __init smp_init(void)
{
        /* Get other processors into their bootup holding patterns. */
        smp_boot_cpus();
        wait_init_idle = cpu_online_map;
        clear_bit(current->processor, &wait_init_idle); /* Don't wait on me! */

        smp_threads_ready=1;
        smp_commence() {
                /* Lets the callins below out of their loop. */
                Dprintk("Setting commenced=1, go go go\n");
                wmb();
                atomic_set(&smp_commenced,1);
```

```
        }

        /* Wait for the other cpus to set up their idle processes */
        printk("Waiting on wait_init_idle (map = 0x%lx)\n", wait_init_idle);
        while (wait_init_idle) {
                cpu_relax();    // i.e. "rep;nop"
                barrier();
        }
        printk("All processors have done init_idle\n");
}

///////////////////////////////////////////////////////////////////////////////
void __init smp_boot_cpus(void)
{
        // ... something not very interesting :-)

        /* Initialize the logical to physical CPU number mapping
         * and the per-CPU profiling router/multiplier */
        prof_counter[0..NR_CPUS-1] = 0;
        prof_old_multiplier[0..NR_CPUS-1] = 0;
        prof_multiplier[0..NR_CPUS-1] = 0;

        init_cpu_to_apicid() {
                physical_apicid_2_cpu[0..MAX_APICID-1] = -1;
                logical_apicid_2_cpu[0..MAX_APICID-1] = -1;
                cpu_2_physical_apicid[0..NR_CPUS-1] = 0;
                cpu_2_logical_apicid[0..NR_CPUS-1] = 0;
        }

        /* Setup boot CPU information */
        smp_store_cpu_info(0); /* Final full version of the data */
        printk("CPU%d: ", 0);
        print_cpu_info(&cpu_data[0]);

        /* We have the boot CPU online for sure. */
        set_bit(0, &cpu_online_map);
        boot_cpu_logical_apicid = logical_smp_processor_id() {
                GET_APIC_LOGICAL_ID(*(unsigned long *)(APIC_BASE+APIC_LDR));
        }
        map_cpu_to_boot_apicid(0, boot_cpu_apicid) {
               physical_apicid_2_cpu[boot_cpu_apicid] = 0;
               cpu_2_physical_apicid[0] = boot_cpu_apicid;
        }

        global_irq_holder = 0;
        current->processor = 0;
        init_idle();    // will clear corresponding bit in wait_init_idle
        smp_tune_scheduling();

        // ... some conditions checked

        connect_bsp_APIC();     // enable APIC mode if used to be PIC mode
        setup_local_APIC();

        if (GET_APIC_ID(apic_read(APIC_ID)) != boot_cpu_physical_apicid)
                BUG();

        /* Scan the CPU present map and fire up the other CPUs
         *   via do_boot_cpu() */
        Dprintk("CPU present map: %lx\n", phys_cpu_present_map);
        for (bit = 0; bit < NR_CPUS; bit++) {
                apicid = cpu_present_to_apicid(bit);
```

```
                /* Don't even attempt to start the boot CPU! */
                if (apicid == boot_cpu_apicid)
                        continue;
                if (!(phys_cpu_present_map & (1 << bit)))
                        continue;
                if ((max_cpus >= 0) && (max_cpus <= cpucount+1))
                        continue;
                do_boot_cpu(apicid);
                /* Make sure we unmap all failed CPUs */
                if ((boot_apicid_to_cpu(apicid) == -1) &&
                                (phys_cpu_present_map & (1 << bit)))
                        printk("CPU #%d not responding - cannot use it.\n",
                                                                apicid);
        }

        // ... SMP BogoMIPS
        // ... B stepping processor warning
        // ... HyperThreading handling

        /* Set up all local APIC timers in the system */
        setup_APIC_clocks();

        /* Synchronize the TSC with the AP */
        if (cpu_has_tsc && cpucount)
                synchronize_tsc_bp();

smp_done:
        zap_low_mappings();
}

///////////////////////////////////////////////////////////////////////////////
static void __init do_boot_cpu (int apicid)
{
        cpu = ++cpucount;

        // 1. prepare "idle process" task struct for next AP

        /* We can't use kernel_thread since we must avoid to
         * reschedule the child. */
        if (fork_by_hand() < 0)
                panic("failed fork for CPU %d", cpu);
        /* We remove it from the pidhash and the runqueue
         * once we got the process: */
        idle = init_task.prev_task;
        if (!idle)
                panic("No idle process for CPU %d", cpu);

        /* we schedule the first task manually */
        idle->processor = cpu;
        idle->cpus_runnable = 1 << cpu; // only on this AP!

        map_cpu_to_boot_apicid(cpu, apicid) {
                physical_apicid_2_cpu[apicid] = cpu;
                cpu_2_physical_apicid[cpu] = apicid;
        }

        idle->thread.eip = (unsigned long) start_secondary;

        del_from_runqueue(idle);
        unhash_process(idle);
        init_tasks[cpu] = idle;
```

```
        // 2. prepare stack and code (CS:IP) for next AP

        /* start_eip had better be page-aligned! */
        start_eip = setup_trampoline() {
                memcpy(trampoline_base, trampoline_data,
                        trampoline_end - trampoline_data);
                /* trampoline_base was reserved in
                 * start_kernel() -> setup_arch() -> smp_alloc_memory(),
                 * and will be shared by all APs (one by one) */
                return virt_to_phys(trampoline_base);
        }

        /* So we see what's up */
        printk("Booting processor %d/%d eip %lx\n", cpu, apicid, start_eip);
        stack_start.esp = (void *) (1024 + PAGE_SIZE + (char *)idle);
        /* this value is used by next AP when it executes
         *   "lss stack_start,%esp" in
         *   linux/arch/i386/kernel/head.S:startup_32(). */

        /* This grunge runs the startup process for
         * the targeted processor. */
        atomic_set(&init_deasserted, 0);
        Dprintk("Setting warm reset code and vector.\n");

        CMOS_WRITE(0xa, 0xf);
        local_flush_tlb();
        Dprintk("1.\n");
        *((volatile unsigned short *) TRAMPOLINE_HIGH) = start_eip >> 4;
        Dprintk("2.\n");
        *((volatile unsigned short *) TRAMPOLINE_LOW) = start_eip & 0xf;
        Dprintk("3.\n");
        // we have setup 0:467 to start_eip (trampoline_base)

        // 3. kick AP to run (AP gets CS:IP from 0:467)

        // Starting actual IPI sequence...
        boot_error = wakeup_secondary_via_INIT(apicid, start_eip);
        if (!boot_error) {      // looks OK
                /* allow APs to start initializing. */
                set_bit(cpu, &cpu_callout_map);

                /* ... Wait 5s total for a response */

                // bit cpu in cpu_callin_map is set by AP in smp_callin()
                if (test_bit(cpu, &cpu_callin_map)) {
                        print_cpu_info(&cpu_data[cpu]);
                } else {
                        boot_error= 1;
                        // marker 0xA5 set by AP in trampoline_data()
                        if (*((volatile unsigned char *)phys_to_virt(8192))
                                        == 0xA5)
                                /* trampoline started but... */
                                printk("Stuck ??\n");
                        else
                                /* trampoline code not run */
                                printk("Not responding.\n");
                }
        }
        if (boot_error) {
                /* Try to put things back the way they were before ... */
                unmap_cpu_to_boot_apicid(cpu, apicid);
                clear_bit(cpu, &cpu_callout_map); /* set in do_boot_cpu() */
```

```
                clear_bit(cpu, &cpu_initialized); /* set in cpu_init() */
                clear_bit(cpu, &cpu_online_map);  /* set in smp_callin() */
                cpucount--;
        }

        /* mark "stuck" area as not stuck */
        *((volatile unsigned long *)phys_to_virt(8192)) = 0;
}
```

Don't confuse *start_secondary()* with *trampoline_data()*. The former is AP "idle" process task struct EIP value, and the latter is the real−mode code that AP runs after BSP kicks it (using *wakeup_secondary_via_INIT()*).

8.3. linux/arch/i386/kernel/trampoline.S

This file contains the 16−bit real−mode AP startup code. BSP reserved memory space *trampoline_base* in *start_kernel() −> setup_arch() −> smp_alloc_memory()*. Before BSP triggers AP, it copies the trampoline code, between *trampoline_data* and *trampoline_end*, to *trampoline_base* (in *do_boot_cpu() −> setup_trampoline()*). BSP sets up 0:467 to point to *trampoline_base*, so that AP will run from here.

```
///////////////////////////////////////////////////////////////////////////////
trampoline_data()
{
r_base:
        wbinvd;         // Needed for NUMA-Q should be harmless for other
        DS = CS;
        BX = 1;         // Flag an SMP trampoline
        cli;

        // write marker for master knows we're running
        trampoline_base = 0xA5A5A5A5;

        lidt idt_48;
        lgdt gdt_48;

        AX = 1;
        lmsw AX;        // protected mode!
        goto flush_instr;
flush_instr:
        goto CS:100000; // see linux/arch/i386/kernel/head.S:startup_32()
}

idt_48:
        .word   0                       # idt limit = 0
        .word   0, 0                    # idt base = 0L

gdt_48:
        .word   0x0800                  # gdt limit = 2048, 256 GDT entries
        .long   gdt_table-__PAGE_OFFSET # gdt base = gdt (first SMP CPU)

.globl SYMBOL_NAME(trampoline_end)
SYMBOL_NAME_LABEL(trampoline_end)
```

Note that BX=1 when AP jumps to `linux/arch/i386/kernel/head.S:startup_32()`, which is different from that of BSP (BX=0). See Section 6.

8.4. initialize_secondary()

Unlike BSP, at the end of *linux/arch/i386/kernel/head.S:startup_32()* in Section 6.4, AP will call *initialize_secondary()* instead of *start_kernel()*.

```
/* Everything has been set up for the secondary
 * CPUs - they just need to reload everything
 * from the task structure
 * This function must not return. */
void __init initialize_secondary(void)
{
        /* We don't actually need to load the full TSS,
         * basically just the stack pointer and the eip. */
        asm volatile(
                "movl %0,%%esp\n\t"
                "jmp *%1"
                :
                :"r" (current->thread.esp),"r" (current->thread.eip));
}
```

As BSP called *do_boot_cpu()* to set *thread.eip* to *start_secondary()*, control of AP is passed to this function. AP uses a new stack frame, which was set up by BSP in *do_boot_cpu() −> fork_by_hand() −> do_fork()*.

8.5. start_secondary()

All APs wait for signal *smp_commenced* from BSP, triggered in Section 8.2 *smp_init() −> smp_commence()*. After getting this signal, they will run "idle" processes.

```
////////////////////////////////////////////////////////////////////////////////
int __init start_secondary(void *unused)
{
        /* Dont put anything before smp_callin(), SMP
         * booting is too fragile that we want to limit the
         * things done here to the most necessary things. */
        cpu_init();
        smp_callin();
        while (!atomic_read(&smp_commenced))
                rep_nop();
        /* low-memory mappings have been cleared, flush them from
         * the local TLBs too. */
        local_flush_tlb();
        return cpu_idle();      // never return, see Section 7.3
}
```

cpu_idle() −> init_idle() will clear corresponding bit in *wait_init_idle*, and finally make BSP finish *smp_init()* and continue with the following function in *start_kernel()* (i.e. *rest_init()*).

8.6. Reference

- MultiProcessor Specification
- IA−32 Intel Architecture Software Developer's Manual
- Linux Kernel 2.4 Internals: Ch.1.7. SMP Bootup on x86
- Linux SMP HOWTO
- ACPI spec

- An Implementation Of Multiprocessor Linux: `linux/Documentation/smp.tex`

A. Kernel Build Example

Here is a kernel build example (in Redhat 9.0). Statements between "/*" and "*/" are in−line comments, not console output.

```
[root@localhost root]# ln −s /usr/src/linux−2.4.20 /usr/src/linux
[root@localhost root]# cd /usr/src/linux
[root@localhost linux]# make xconfig
        /* Create .config
         *   1. "Load Configuration from File" −>
         *        /boot/config−2.4.20−28.9, or whatever you like
         *   2. Modify kernel configuration parameters
         *   3. "Save and Exit" */
[root@localhost linux]# make oldconfig
        /* Re−check .config, optional */
[root@localhost linux]# vi Makefile
        /* Modify EXTRAVERSION in linux/Makefile, optional */
[root@localhost linux]# make dep
        /* Create .depend and more */
[root@localhost linux]# make bzImage
        /* ... Some output omitted */
ld −m elf_i386 −T /usr/src/linux−2.4.20/arch/i386/vmlinux.lds −e stext arch/i386
/kernel/head.o arch/i386/kernel/init_task.o init/main.o init/version.o init/do_m
ounts.o \
        −−start−group \
        arch/i386/kernel/kernel.o arch/i386/mm/mm.o kernel/kernel.o mm/mm.o fs/f
s.o ipc/ipc.o \
         drivers/char/char.o drivers/block/block.o drivers/misc/misc.o drivers/n
et/net.o drivers/media/media.o drivers/char/drm/drm.o drivers/net/fc/fc.o driver
s/net/appletalk/appletalk.o drivers/net/tokenring/tr.o drivers/net/wan/wan.o dri
vers/atm/atm.o drivers/ide/idedriver.o drivers/cdrom/driver.o drivers/pci/driver
.o drivers/net/pcmcia/pcmcia_net.o drivers/net/wireless/wireless_net.o drivers/p
np/pnp.o drivers/video/video.o drivers/net/hamradio/hamradio.o drivers/md/mddev.
o drivers/isdn/vmlinux−obj.o \
        net/network.o \
        /usr/src/linux−2.4.20/arch/i386/lib/lib.a /usr/src/linux−2.4.20/lib/lib.
a /usr/src/linux−2.4.20/arch/i386/lib/lib.a \
        −−end−group \
        −o vmlinux
nm vmlinux | grep −v '\(compiled\)\|\(\.o$\)\|\( [aUw] \)\|\(\.\.ng$\)\|\(LASH[R
L]DI\)' | sort > System.map
make[1]: Entering directory `/usr/src/linux−2.4.20/arch/i386/boot'
gcc −E −D__KERNEL__ −I/usr/src/linux−2.4.20/include −D__BIG_KERNEL__ −traditiona
l −DSVGA_MODE=NORMAL_VGA  bootsect.S −o bbootsect.s
as −o bbootsect.o bbootsect.s
bootsect.S: Assembler messages:
bootsect.S:239: Warning: indirect lcall without `*'
ld −m elf_i386 −Ttext 0x0 −s −−oformat binary bbootsect.o −o bbootsect
gcc −E −D__KERNEL__ −I/usr/src/linux−2.4.20/include −D__BIG_KERNEL__ −D__ASSEMBL
Y__ −traditional −DSVGA_MODE=NORMAL_VGA  setup.S −o bsetup.s
as −o bsetup.o bsetup.s
setup.S: Assembler messages:
setup.S:230: Warning: indirect lcall without `*'
ld −m elf_i386 −Ttext 0x0 −s −−oformat binary −e begtext −o bsetup bsetup.o
make[2]: Entering directory `/usr/src/linux−2.4.20/arch/i386/boot/compressed'
tmppiggy=_tmp_$$piggy; \
rm −f $tmppiggy $tmppiggy.gz $tmppiggy.lnk; \
objcopy −O binary −R .note −R .comment −S /usr/src/linux−2.4.20/vmlinux $tmppigg
y; \
gzip −f −9 < $tmppiggy > $tmppiggy.gz; \
```

```
echo "SECTIONS { .data : { input_len = .; LONG(input_data_end - input_data) inpu
t_data = .; *(.data) input_data_end = .; }}" > $tmppiggy.lnk; \
ld -m elf_i386 -r -o piggy.o -b binary $tmppiggy.gz -b elf32-i386 -T $tmppiggy.l
nk; \
rm -f $tmppiggy $tmppiggy.gz $tmppiggy.lnk
gcc -D__ASSEMBLY__ -D__KERNEL__ -I/usr/src/linux-2.4.20/include -traditional -c
head.S
gcc -D__KERNEL__ -I/usr/src/linux-2.4.20/include -Wall -Wstrict-prototypes -Wno-
trigraphs -O2 -fno-strict-aliasing -fno-common -fomit-frame-pointer -pipe -mpref
erred-stack-boundary=2 -march=i686 -DKBUILD_BASENAME=misc -c misc.c
ld -m elf_i386 -Ttext 0x100000 -e startup_32 -o bvmlinux head.o misc.o piggy.o
make[2]: Leaving directory `/usr/src/linux-2.4.20/arch/i386/boot/compressed'
gcc -Wall -Wstrict-prototypes -O2 -fomit-frame-pointer -o tools/build tools/buil
d.c -I/usr/src/linux-2.4.20/include
objcopy -O binary -R .note -R .comment -S compressed/bvmlinux compressed/bvmlinu
x.out
tools/build -b bbootsect bsetup compressed/bvmlinux.out CURRENT > bzImage
Root device is (3, 67)
Boot sector 512 bytes.
Setup is 4780 bytes.
System is 852 kB
make[1]: Leaving directory `/usr/src/linux-2.4.20/arch/i386/boot'
[root@localhost linux]# make modules modules_install
        /* ... Some output omitted */
cd /lib/modules/2.4.20; \
mkdir -p pcmcia; \
find kernel -path '*/pcmcia/*' -name '*.o' | xargs -i -r ln -sf ../{} pcmcia
if [ -r System.map ]; then /sbin/depmod -ae -F System.map  2.4.20; fi
[root@localhost linux]# cp arch/i386/boot/bzImage /boot/vmlinuz-2.4.20
[root@localhost linux]# cp vmlinux /boot/vmlinux-2.4.20
[root@localhost linux]# cp System.map /boot/System.map-2.4.20
[root@localhost linux]# cp .config /boot/config-2.4.20
[root@localhost linux]# mkinitrd /boot/initrd-2.4.20.img 2.4.20
[root@localhost linux]# vi /boot/grub/grub.conf
        /* Add the following lines to grub.conf:
title Linux (2.4.20)
        kernel /vmlinuz-2.4.20 ro root=LABEL=/
        initrd /initrd-2.4.20.img
         */
```

Refer to Kernelnewbies FAQ: How do I compile a kernel and Kernel Rebuild Procedure for more details.

To build the kernel in Debian, also refer to Debian Installation Manual: Compiling a New Kernel, The Debian GNU/Linux FAQ: Debian and the kernel and Debian Reference: The Linux kernel under Debian. Check "**zless /usr/share/doc/kernel−package/Problems.gz**" if you encounter problems.

B. Internal Linker Script

Without −T (−−script=) option specified, **ld** will use this builtin script to link targets:

```
[root@localhost linux]# ld --verbose
GNU ld version 2.13.90.0.18 20030206
  Supported emulations:
   elf_i386
   i386linux
using internal linker script:
==================================================
/* Script for -z combreloc: combine and sort reloc sections */
OUTPUT_FORMAT("elf32-i386", "elf32-i386",
              "elf32-i386")
OUTPUT_ARCH(i386)
ENTRY(_start)
SEARCH_DIR("/usr/i386-redhat-linux/lib"); SEARCH_DIR("/usr/lib"); SEARCH_DIR("/u
sr/local/lib"); SEARCH_DIR("/lib");
/* Do we need any of these for elf?
   __DYNAMIC = 0;    */
SECTIONS
{
  /* Read-only sections, merged into text segment: */
  . = 0x08048000 + SIZEOF_HEADERS;
  .interp         : { *(.interp) }
  .hash           : { *(.hash) }
  .dynsym         : { *(.dynsym) }
  .dynstr         : { *(.dynstr) }
  .gnu.version    : { *(.gnu.version) }
  .gnu.version_d  : { *(.gnu.version_d) }
  .gnu.version_r  : { *(.gnu.version_r) }
  .rel.dyn        :
    {
      *(.rel.init)
      *(.rel.text .rel.text.* .rel.gnu.linkonce.t.*)
      *(.rel.fini)
      *(.rel.rodata .rel.rodata.* .rel.gnu.linkonce.r.*)
      *(.rel.data .rel.data.* .rel.gnu.linkonce.d.*)
      *(.rel.tdata .rel.tdata.* .rel.gnu.linkonce.td.*)
      *(.rel.tbss .rel.tbss.* .rel.gnu.linkonce.tb.*)
      *(.rel.ctors)
      *(.rel.dtors)
      *(.rel.got)
      *(.rel.bss .rel.bss.* .rel.gnu.linkonce.b.*)
    }
  .rela.dyn       :
    {
      *(.rela.init)
      *(.rela.text .rela.text.* .rela.gnu.linkonce.t.*)
      *(.rela.fini)
      *(.rela.rodata .rela.rodata.* .rela.gnu.linkonce.r.*)
      *(.rela.data .rela.data.* .rela.gnu.linkonce.d.*)
      *(.rela.tdata .rela.tdata.* .rela.gnu.linkonce.td.*)
      *(.rela.tbss .rela.tbss.* .rela.gnu.linkonce.tb.*)
      *(.rela.ctors)
      *(.rela.dtors)
      *(.rela.got)
      *(.rela.bss .rela.bss.* .rela.gnu.linkonce.b.*)
    }
  .rel.plt        : { *(.rel.plt) }
```

```
  .rela.plt       : { *(.rela.plt) }
  .init           :
  {
    KEEP (*(.init))
  } =0x90909090
  .plt            : { *(.plt) }
  .text           :
  {
    *(.text .stub .text.* .gnu.linkonce.t.*)
    /* .gnu.warning sections are handled specially by elf32.em.  */
    *(.gnu.warning)
  } =0x90909090
  .fini           :
  {
    KEEP (*(.fini))
  } =0x90909090
  PROVIDE (__etext = .);
  PROVIDE (_etext = .);
  PROVIDE (etext = .);
  .rodata         : { *(.rodata .rodata.* .gnu.linkonce.r.*) }
  .rodata1        : { *(.rodata1) }
  .eh_frame_hdr : { *(.eh_frame_hdr) }
  .eh_frame       : ONLY_IF_RO { KEEP (*(.eh_frame)) }
  .gcc_except_table   : ONLY_IF_RO { *(.gcc_except_table) }
  /* Adjust the address for the data segment.  We want to adjust up to
     the same address within the page on the next page up.  */
  . = ALIGN (0x1000) - ((0x1000 - .) & (0x1000 - 1)); . = DATA_SEGMENT_ALIGN (0x
1000, 0x1000);
  /* For backward-compatibility with tools that don't support the
     *_array_* sections below, our glibc's crt files contain weak
     definitions of symbols that they reference.  We don't want to use
     them, though, unless they're strictly necessary, because they'd
     bring us empty sections, unlike PROVIDE below, so we drop the
     sections from the crt files here.  */
  /DISCARD/ : {
      */crti.o(.init_array .fini_array .preinit_array)
      */crtn.o(.init_array .fini_array .preinit_array)
  }
  /* Ensure the __preinit_array_start label is properly aligned.  We
     could instead move the label definition inside the section, but
     the linker would then create the section even if it turns out to
     be empty, which isn't pretty.  */
  . = ALIGN(32 / 8);
  PROVIDE (__preinit_array_start = .);
  .preinit_array     : { *(.preinit_array) }
  PROVIDE (__preinit_array_end = .);
  PROVIDE (__init_array_start = .);
  .init_array     : { *(.init_array) }
  PROVIDE (__init_array_end = .);
  PROVIDE (__fini_array_start = .);
  .fini_array     : { *(.fini_array) }
  PROVIDE (__fini_array_end = .);
  .data           :
  {
    *(.data .data.* .gnu.linkonce.d.*)
    SORT(CONSTRUCTORS)
  }
  .data1          : { *(.data1) }
  .tdata          : { *(.tdata .tdata.* .gnu.linkonce.td.*) }
  .tbss           : { *(.tbss .tbss.* .gnu.linkonce.tb.*) *(.tcommon) }
  .eh_frame       : ONLY_IF_RW { KEEP (*(.eh_frame)) }
  .gcc_except_table   : ONLY_IF_RW { *(.gcc_except_table) }
```

```
  .dynamic        : { *(.dynamic) }
  .ctors          :
  {
    /* gcc uses crtbegin.o to find the start of
       the constructors, so we make sure it is
       first.  Because this is a wildcard, it
       doesn't matter if the user does not
       actually link against crtbegin.o; the
       linker won't look for a file to match a
       wildcard.  The wildcard also means that it
       doesn't matter which directory crtbegin.o
       is in.  */
    KEEP (*crtbegin.o(.ctors))
    /* We don't want to include the .ctor section from
       from the crtend.o file until after the sorted ctors.
       The .ctor section from the crtend file contains the
       end of ctors marker and it must be last */
    KEEP (*(EXCLUDE_FILE (*crtend.o ) .ctors))
    KEEP (*(SORT(.ctors.*)))
    KEEP (*(.ctors))
  }
  .dtors          :
  {
    KEEP (*crtbegin.o(.dtors))
    KEEP (*(EXCLUDE_FILE (*crtend.o ) .dtors))
    KEEP (*(SORT(.dtors.*)))
    KEEP (*(.dtors))
  }
  .jcr            : { KEEP (*(.jcr)) }
  .got            : { *(.got.plt) *(.got) }
  _edata = .;
  PROVIDE (edata = .);
  __bss_start = .;
  .bss            :
  {
   *(.dynbss)
   *(.bss .bss.* .gnu.linkonce.b.*)
   *(COMMON)
   /* Align here to ensure that the .bss section occupies space up to
      _end.  Align after .bss to ensure correct alignment even if the
      .bss section disappears because there are no input sections.  */
   . = ALIGN(32 / 8);
  }
  . = ALIGN(32 / 8);
  _end = .;
  PROVIDE (end = .);
  . = DATA_SEGMENT_END (.);
  /* Stabs debugging sections.  */
  .stab          0 : { *(.stab) }
  .stabstr       0 : { *(.stabstr) }
  .stab.excl     0 : { *(.stab.excl) }
  .stab.exclstr  0 : { *(.stab.exclstr) }
  .stab.index    0 : { *(.stab.index) }
  .stab.indexstr 0 : { *(.stab.indexstr) }
  .comment       0 : { *(.comment) }
  /* DWARF debug sections.
     Symbols in the DWARF debugging sections are relative to the beginning
     of the section so we begin them at 0.  */
  /* DWARF 1 */
  .debug          0 : { *(.debug) }
  .line           0 : { *(.line) }
  /* GNU DWARF 1 extensions */
```

```
  .debug_srcinfo  0 : { *(.debug_srcinfo) }
  .debug_sfnames  0 : { *(.debug_sfnames) }
  /* DWARF 1.1 and DWARF 2 */
  .debug_aranges  0 : { *(.debug_aranges) }
  .debug_pubnames 0 : { *(.debug_pubnames) }
  /* DWARF 2 */
  .debug_info     0 : { *(.debug_info .gnu.linkonce.wi.*) }
  .debug_abbrev   0 : { *(.debug_abbrev) }
  .debug_line     0 : { *(.debug_line) }
  .debug_frame    0 : { *(.debug_frame) }
  .debug_str      0 : { *(.debug_str) }
  .debug_loc      0 : { *(.debug_loc) }
  .debug_macinfo  0 : { *(.debug_macinfo) }
  /* SGI/MIPS DWARF 2 extensions */
  .debug_weaknames 0 : { *(.debug_weaknames) }
  .debug_funcnames 0 : { *(.debug_funcnames) }
  .debug_typenames 0 : { *(.debug_typenames) }
  .debug_varnames  0 : { *(.debug_varnames) }
}


==================================================
[root@localhost linux]#
```

C. GRUB and LILO

Both GNU GRUB and LILO understand the real−mode kernel header format and will load the bootsect (one sector), setup code (*setup_sects* sectors) and compressed kernel image (*syssize**16 bytes) into memory. They fill out the loader identifier (*type_of_loader*) and try to pass appropriate parameters and options to the kernel. After they finish their jobs, control is passed to setup code.

C.1. GNU GRUB

The following GNU GRUB program outline is based on grub−0.93.

```
stage2/stage2.c:cmain()
`-- run_menu()
    `-- run_script();
        |-- builtin = find_command(heap);
        |-- kernel_func();              // builtin->func() for command "kernel"
        |   `-- load_image();           // search BOOTSEC_SIGNATURE in boot.c
        |   /* memory from 0x100000 is populated by and in the order of
        |    *   (bvmlinux, bbootsect, bsetup) or (vmlinux, bootsect, setup) */
        |-- initrd_func();              // for command "initrd"
        |   `-- load_initrd();
        `-- boot_func();                // for implicit command "boot"
            `-- linux_boot();           // defined in stage2/asm.S
                or big_linux_boot();    //   not in grub/asmstub.c!

// In stage2/asm.S
linux_boot:
        /* copy kernel */
        move system code from 0x100000 to 0x10000 (linux_text_len bytes);
big_linux_boot:
        /* copy the real mode part */
        EBX = linux_data_real_addr;
        move setup code from linux_data_tmp_addr (0x100000+text_len)
            to linux_data_real_addr (0x9100 bytes);
        /* change %ebx to the segment address */
        linux_setup_seg = (EBX >> 4) + 0x20;
        /* XXX new stack pointer in safe area for calling functions */
        ESP = 0x4000;
        stop_floppy();
        /* final setup for linux boot */
        prot_to_real();
        cli;
        SS:ESP = BX:9000;
        DS = ES = FS = GS = BX;
        /* jump to start, i.e. ljmp linux_setup_seg:0
         * Note that linux_setup_seg is just changed to BX. */
        .byte   0xea
        .word   0
linux_setup_seg:
        .word   0
```

Refer to "**info grub**" for GRUB manual.

One reported GNU GRUB bug should be noted if you are porting grub−0.93 and making changes to *bsetup*.

C.2. LILO

Unlike GRUB, LILO does not check the configuration file when booting system. Tricks happen when **lilo** is invoked from terminal.

The following LILO program outline is based on lilo−22.5.8.

```
lilo.c:main()
|-- cfg_open(config_file);
|-- cfg_parse(cf_options);
|-- bsect_open(boot_dev, map_file, install, delay, timeout);
|   |-- open_bsect(boot_dev);
|   `-- map_create(map_file);
|-- cfg_parse(cf_top)
|   `-- cfg_do_set();
|       `-- do_image();             // walk->action for "image=" section
|           |-- cfg_parse(cf_image) -> cfg_do_set();
|           |-- bsect_common(&descr, 1);
|           |   |-- map_begin_section();
|           |   |-- map_add_sector(fallback_buf);
|           |   `-- map_add_sector(options);
|           |-- boot_image(name, &descr) or boot_device(name, range, &descr);
|           |   |-- int fd = geo_open(&descr, name, O_RDONLY);
|           |   |   read(fd, &buff, SECTOR_SIZE);
|           |   |   map_add(&geo, 0, image_sectors);
|           |   |   map_end_section(&descr->start, setup_sects+2+1);
|           |   |       /* two sectors created in bsect_common(),
|           |   |        *   another one sector for bootsect */
|           |   |   geo_close(&geo);
|           |   `-- fd = geo_open(&descr, initrd, O_RDONLY);
|           |       map_begin_section();
|           |       map_add(&geo, 0, initrd_sectors);
|           |       map_end_section(&descr->initrd,0);
|           |       geo_close(&geo);
|           `-- bsect_done(name, &descr);
`-- bsect_update(backup_file, force_backup, 0); // update boot sector
    |-- make_backup();
    |-- map_begin_section();
    |   map_add_sector(table);
    |   map_write(&param2, keytab, 0, 0);
    |   map_close(&param2, here2);
    |-- // ... perform the relocation of the boot sector
    |-- // ... setup bsect_wr to correct place
    |-- write(fd, bsect_wr, SECTOR_SIZE);
    `-- close(fd);
```

map_add(), map_add_sector() and *map_add_zero()* may call *map_register()* to complete their jobs, while *map_register()* will keep a list for all (CX, DX, AL) triplets (data structure SECTOR_ADDR) used to identify all registered sectors.

LILO runs `first.S` and `second.S` to boot a system. It calls *second.S:doboot()* to load map file, bootsect and setup code. Then it calls *lfile()* to load the system code, calls *launch2() −> launch() −> cl_wait() −> start_setup() −> start_setup2()* and finnaly executes "jmpi 0,SETUPSEG" instruction to run setup code.

Refer to "**man lilo**" and "**man lilo.conf**" for LILO details.

C.3. Reference

- GNU GRUB
- GRUB Tutorial
- LILO (freshmeat.net)
- LDP HOWTO−INDEX: Boot Loaders and Booting the OS

D. FAQ

For things that are to be in appropriate chapters, or should be here. /* TODO: */