IBM Haifa Labs

Look, Ma, No BIOS!

A Hands-Off Introduction To LinuxBIOS

Oleg Goldshmidt
IBM Haifa Research Laboratory

July 2006

BIOS Milestones

- Early 1970ies: Gary Kildall's 50 mile commute
  - Intel's 8080, Interp/80, PL/M
  - Intel's Intellec-8 (microprocessor + memory + terminal port – storage)
  - IBM's (8 inch, $500) floppy disk
  - John Torode – floppy controller, Gary Kildall – CP/M (Control Program/Monitor)
- 1976: Intel loses interest, Imsai (and a 100 other companies) need an OS
  - Kildall splits CP/M into hardware-independent part and hardware-specific BIOS
- Around 1980: IBM in Boca Raton make the PC (Acorn) – in 1 year
  - Broke every IBM rule by making an open system, published all specs
  - Published – and copyrighted! - BIOS code
- 1981: Rod Canion, Jim Harris, and Bill Murto at Houston's House of Pies
  - Business plans on placemat: Mexican restaurant, build HDs, car key finder
  - Winner: reverse engineer BIOS to make a 100% IBM-compatible PC
  - Investment: 15 senior programmers, several months, $1MM
- Around 1982: Phoenix sells IBM-compatible BIOS chips for $25

How BIOS Works

- From the cold state a special hardware circuit sends a signal to the RESET pin of the CPU
- Some registers (cs, eip) are set to fixed values, and the code found at physical address 0xfffffff0 is executed
  - The address is mapped by hardware to BIOS ROM chip
- Linux does not use BIOS routines, BIOS must be executed in real mode
  - Only real mode addresses (seg*16+off) are available in the cold state
  - No GDT, LDT, or paging are needed, must be initialized in real mode
- Major BIOS tasks
  - POST
  - Hardware initialization (e.g. PCI configuration)
  - Search for an OS to boot (configurable)
  - Copies the first sector of the boot device to RAM (at 0x00007c00), jumps to it, and executes
- Invokes the bootloader

How Bootloader Works

- Floppy
  - The instructions in the first sector are loaded into RAM and executed
  - Bootloader copies itself from 0x00007c00 to 0x00090000
  - Sets up real mode stack
  - Invokes a BIOS procedure to load the setup() code to 0x00090200
  - Invokes a BIOS procedure to load the rest of the image from 0x00100000
  - Jumps to setup() code
- Hard Disk
  - The first sector (MBR) contains the partition table and a small program that loads the first sector of the partition with the chosen OS
  - Some OS (e.g. Win98) identify the boot partition with an “active” flag
    - Only the OS whose image is on the active partition may be loaded
  - Sophisticated programs such as LILO or GRUB may allow runtime choice of image
  - LILO’s first stage is loaded to 0x00007c00, moves itself to 0x0009a000, sets up real mode stack, loads second stage to 0x0009b000
  - User chooses kernel, the rest is similar to floppy
  - Execution jumps to setup()

How setup() Works

- The setup() function is loaded at offset 0x200 of the kernel image file
- Initializes hardware devices and sets up the kernel execution environment
  - But Linux sometimes relies on BIOS for initialization
- Major operations
  - Invokes a BIOS procedure to determine the amount of RAM
  - Initializes keyboard
  - Initializes the graphics adapter
  - Reinitializes the disk controller
  - Checks for buses, bus mice, APM, etc
  - If needed, moves the kernel image to make space for decompression
  - Sets up the Interrupt Descriptor Table (IDT) and a Global Descriptor Table (GDT)
  - Remaps the interrupts (BIOS maps hardware interrupts to the CPU exception range)
  - Switches to protected mode
  - Jumps to startup_32(), which starts the kernel proper

BIOS Today: A Pain In Sensitive Parts Of Anatomy

- Clusters (started at Cluster Research Lab, Advanced Computing Lab, LANL)
  - Nodes depend on vendor-supplied BIOS for booting
  - BIOS normally relies on inherently unreliable legacy devices (floppy, HD) to boot the OS
  - BIOS cannot deal with non-standard, experimental hardware
    Need full control of the boot process, to the point that jobs in the queue might indicate which kernel to run, where to find root FS, etc.

- Maintenance is a nightmare
  - BIOSes are buggy and cannot be fixed (by user)
  - The development toolchain is highly specialized (read: obsolete), difficult to come by, can be quite expensive
  - Try wandering around a few hundred node cluster with a monitor and a keyboard to change one BIOS setting

- Get rid of the legacy BIOS altogether!

Requirements and Decisions (As Specified by LANL)

- Starting point: the current status of netboot on PCs
  - netboot has been available on Suns for more than 15 years
  - on PCs, BIOS and PROM must be in 16-bit mode - 8086 (6MHz, 25 years old CPU) emulation, all sorts of weird stuff such as near and far pointers, etc
  - standards: NIC boot model has to conform to NDIS2 (16-bit Windows model), thanks to Microsoft and Intel
- Requirements
  - load something onto CPU that can load boot parameters over network, find out what to do, and load an OS kernel
  - open source – nothing proprietary
  - portable: no (or minimal) assembly, support a variety of NICs, motherboards
  - avoid reinventing the wheel – try to use code that supports lots of hardware etc
  - support standard protocols – NFS, bootp, etc
- Write netboot from scratch or use a minimal Linux kernel?
  - custom netboot will duplicate OS (Sun netboot – no support for AFS, msdos, etc)
  - no real space savings (netboot – 128K, minimal Linux – 300K)

Operation, Benefits and Disadvantages of LinuxBIOS

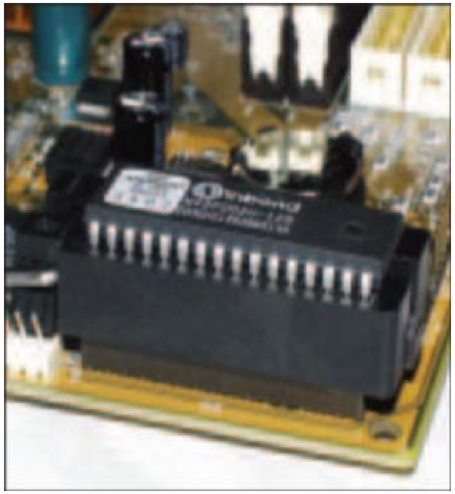
- Gunzips Linux out of flash (NVRAM)
  - No moving parts but the fan
  - Minimal amount of hardware initialization, Linux does the rest
  - Kernel boot times (not system start) as fast as 3 seconds
- Using a real OS rather than simple netboot or BIOS is more flexible
  - Linux can boot over Ethernet, Myrinet, Quadrics, SCI
  - Cluster nodes can be as simple as CPU + memory + network
- Boot diagnostic and maintenance over serial port
- Not all motherboards are supported
- Hardware manipulation may be involved

LinuxBIOS Do-It-Yourself HOWTO

- Check whether the motherboard is supported
  - The BIOS chip must be removable from its socket
- Hardware needed:
  - Disk-on-Chip (DoC) memory device to be plugged in instead of the BIOS chip (8 MByte instead of 2 Mbit)
  - 32-pin ZIF (Zero Insertion Force) socket to make swapping chips easy and safe
  - "development" and "target" machines (can be the same), BIOS chip in ZIF on target
- Install Linux on target, including DoC support (as module)
- Configure and build LinuxBIOS, a patched kernel (working, not latest), and MTD utilities ("erase", "flash_on"). NB: source code changes may be needed.
- With power on, remove the BIOS chip from ZIF and insert the DoC
- Burn LinuxBIOS into DoC using the "burn_mtd" utility
- Reboot (through power cycle) while saying the appropriate prayers to root ("Root, G-d, what is difference?" – Illiad Fraser)

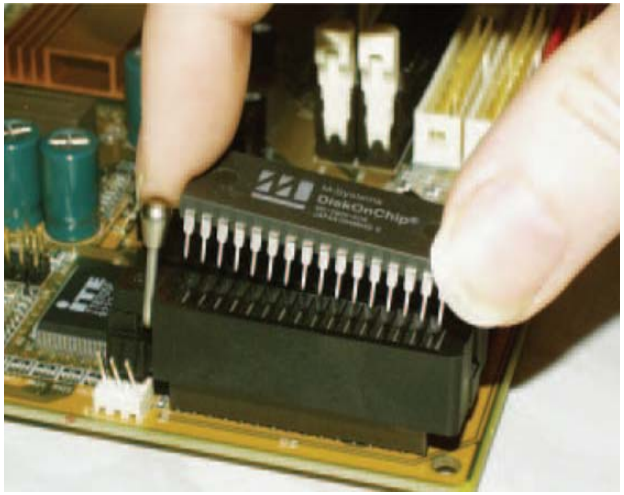
Swapping The BIOS Chip

- The ZIF socket plugged into the motherboard, with the original BIOS chip inserted (both images are from the Linux Magazine article on LinuxBIOS, see “Further Information” below for reference)



- Inserting the DoC into the ZIF socket. WARNING: BE ADVISED THAT THIS STEP CAN HURT OR KILL YOU!

  If you haven't done this, or are not trained, or have a history of getting hurt by hardware, DON'T DO IT.



What Can Go Wrong

- any time you stick your hand into an open machine while the power is on, you're risking life and limb (ESD), and

  YOU CAN MESS UP THE HARDWARE:

- incorrect insertion of the flash (shown)
  - reverses power and ground
  - the DoC gets **very** hot (see the molten sticker)
- incorrect jumper settings
- aggressive and/or inappropriate use of metal objects such as screwdrivers
- miscellaneous miswirings and mishandlings

  (the image is from the LinuxBIOS website, http://www.linuxbios.org)

After LinuxBIOS Boots: Troubleshooting etc.

- If you see a penguin instead of the normal BIOS logo – your prayers have been heard…
- No HD, keyboard, ethernet, root FS? Not to worry – can be sorted out. The main thing is to have the kernel running.
- No penguin? You have not said the right prayers:
  - Connect a null modem RS-232 cable to the first serial port on target
  - Connect the cable to the development machine and start a 115200 baud, 8 bit, no parity serial terminal emulator
  - Reboot and debug
- Create a root FS on the DoC
  - Enable a few more MTD options (to format, write the fs) in kernel, rebuild, and reboot
  - Create a partition (~7MB) in the DoC, and format the fs
  - Change the device that LinuxBIOS expects to find the root fs on
  - What to put into the root fs? See Tom's Root Boot or another tiny distribution.

How It Works

- 32-bit mode from the start: load GDT and enable memory protection
  - execute LGDT instruction and provide a pointer to a table descriptor
  - no problem with having the table in NVRAM - takes about 10 instructions
- Basic chipset initialization
  - assembly code needed to turn DRAM on
  - the rest can be done in C
- Linux assumes hardware is initialized by BIOS. LinuxBIOS cannot assume this
  - e.g., if IDE controller is not initialized then Linux assumes that BIOS disabled it
  - one-line change to make the driver enable IDE by default
- 5 major components:
  - protected mode setup
  - DRAM setup
  - transition to C
  - mainboard fixup
  - Kernel unzip and jump to kernel

Protected Mode Setup: How To Run a Pentium

- Puts segmentation, paging, TLB h/w in a sane state, turn on segmentation, not paging - 17 instructions:
  - 1st instruction, executed in 16-bit mode: jump to BIOS startup – a standard part of x86 reset
  - 5 instructions: disable interrupts, clear TLB, set code and data segments to known values
  - 1 instruction: load a pointer to GDT (to manage addressing in segmented mode)
  - 4 instructions: turn on memory protection
  - 6 instructions: do remaining segment register setup for protected mode
- At this point we
  - are running in protected mode
  - can address 4GB of memory
  - are running a Pentium, not a 8086

DRAM, Transition to C, Mainboard Fixup, Starting Kernel

- DRAM setup
  - non-portable, tricky, tough to figure out
- Transition to C
  - set up the stack and call a function to do the rest
- Mainboard fixup
  - turn on cacheing (MTRR) so that the kernel unzips in reasonable time
    - Otherwise it can take a minute or more
  - make all the FLASH available (rather than 64K or 128K)
    - requires some register manipulation, different for each chipset
  - minimal power management capabilities
  - do stuff Linux cannot (or will not) do:
    - turn clock interrupts on
    - a bit of PCI initialization – set Base Address Registers
- Inflate and run the kernel (snarfed from the kernel itself)
  - handle parameters, command line
  - make gunzip work in ROM environment (declare initialized arrays const)
  - jump to startup_32(), not to setup() as LILO does

OK, How Do We Boot a Real Kernel from LinuxBIOS? LOBOS

- LOBOS (Linux OS Boots OS) – a system call that allows Linux to boot another OS without leaving the 32-bit protected mode or using the BIOS
- Overlaying the kernel (in kernel mode)
  - read a file into memory not occupied by the running kernel
  - move critical structures (page tables, boot arguments, root partition location, log buffer, etc) into a safe place
  - turn off interrupts (point of no return – check all errors)
  - switch memory to the new page tables
  - copy the final bootstrap code (that copies the kernel to 0x100000 and jumps to it) to a safe place where it will not be overwritten by the new kernel
  - jump to the final bootstrap code that will do its thing

LOBOS Implementation

- 5 major pieces in about 300 lines
  - entry for the new system call in arch/i386/kernel/entry.S
  - some additions to arch/i386/kernel/head.S to make space for the critical kernel data that will be copied
    - a few additional pages at the beginning of the kernel virtual address space, not used in normal kernel operation, hence safe
  - the code to read in the new file in kernel/sys.c
    - sys_lobos(char* file) looks up the file and calls read_exec()
  - the code to switch off interrupts, move the critical data, and switch over to the new page tables, in arch/i386/kernel/process.c
    - os_restart(), some assembly required…
  - the code to copy the new kernel to the right place and jump to it, in kernel/sys.c
- can be called from userspace, command line
- faster than BIOS (a lot of waiting in BIOS is for DOS 1.0 support tasks)

Related Work: bootimg, Two Kernel Monte (TKM)

- bootimg
  - allows a userspace program to read a file in and boot a new image via a system call
  - turns VM (paging) off (but not i386-style segmentation)
  - the user buffer (in VM) has to be copied into kernel memory accessible without VM
  - relies on kernel components being in physically contiguous memory
  - 1100 LOC (600 architecture-dependent), some structures (GDT) need to be maintained in sync with kernel
  - can be used with LinuxBIOS
  - thorough permissions checking, ramdisk support
- Two Kernel Monte (TKM)
  - turns off both VM and i386 segmentation
  - builds and internal virtual-to-physical page map (so it can still get to kernel)
  - relies on BIOS to reset hardware (e.g. video card) after reboot
  - cannot be used with LinuxBIOS (because of reliance on the BIOS)
- Both require an external program to boot a new image (LOBOS doesn't)

Status

- Mainboards, chipsets
  - Too many to list, quite a few vendors, but not guaranteed that yours will work
  - Many are marked as "unstable"
  - Better chance with Intel than with others
- Architectures
  - Alpha, K8, K7, PowerPC, P4, PIII, PII, Cyrix (VIA), Geode (now AMD) and SC520 (AMD).
  - PPC (some: Motorola Sandpoint is reported as working, IBM 970 port in progress)
- OS
  - Linux (but of course)
  - OpenBSD
  - Win2K (LILO and GRUB support with the help of BOCHS)
- Deployment
  - the biggest, baddest, fastest clusters built by LinuxNetworx for LANL
  - Pentium and K8

LinuxBIOS in Real Life (Have You Even Been to Los Alamos?)

- Already there:
  - The Superdense Server project (IBM Austin)
    - http://www.research.ibm.com/journal/rd/475/felter.pdf
- Potentially:
  - iSCSI/iBOOT
  - Experimental hardware projects
- Difficulties:
  - Convince everybody there is a commercial advantage
  - Convince Marketing that they can convince potential customers…
  - Convince Legal that open source (GPL) will not put you in jail or out of business
  - Deal with existing and emerging de-facto standards (e.g., EFI)
- Create a market:
  - LANL RFPed 2 clusters requiring LinuxBIOS at $19MM – vendors lined up

Summary

- A promising direction in general – let's get rid of the BIOS!
- For prototype development:
  - tinkering with hardware may cause pain in various parts of anatomy, or sudden death
  - not clear if target hardware or prototype hardware is supported
    - quite a bit of hacking may be required – not necessarily a bad thing
  - open source - a major advantage for hacking
  - limited OS support, though LILO/GRUB are reported to work, so there is hope
- For product development:
  - need a limited range of hardware to work
  - tinkering with hardware is not a problem
  - quite a few advantages in deployment
    - fast boot
    - flexible
    - concurrent remote access, logging, etc
  - biggest question – hardware and OS support

Further Information

- Main website:
  - http://www.linuxbios.org
- “Putting Linux on Your Motherboard” by Antony Stone, Linux Magazine, March 2003, p. 76.
  - http://www.linuxbios.org/papers/linux-magazine/LinuxBIOS.pdf
- “LinuxBIOS at Four” by Ron Minnich, Linux Journal, February 2004.
  - http://www.linuxjournal.com/print.php?sid=7170
- “LOBOS (Linux OS Boots Linux OS): Booting a Kernel in 32-bit Mode” by Ron Minnich, the 4th Annual Linux Showcase and Conference, October 2000
  - http://www.linuxbios.org/papers/als00/lobos.pdf
- Other papers:
  - http://www.linuxbios.org/papers
- Mailing list
  - http://www.clustermatic.org/mailman/listinfo/linuxbios
- BIOS history: “Accidental Empires” by Robert X. Cringely, HarperBusiness