

REPORT ON

LINUXBIOS

SEMINAR PRESENTED
BY

DEBAJIT ADHIKARY

2004

TABLE OF CONTENTS

1	INTRODUCTION	1
1.1	What is LinuxBIOS?	1
1.2	Why do we need LinuxBIOS?	1
	— Problems with the PC BIOS	2
1.3	A Comparison: LinuxBIOS vs. PC BIOS	3
1.4	Advantages	4
2	STRUCTURE OF LINUXBIOS	5
3	POST-STARTUP SCENARIOS	7
3.1	Network Boot	7
3.2	Standard Boot	8
3.3	Diskless Node	8
3.4	Maintenance	9
3.5	Netboot over MyriNet	9
3.6	Netboot over Scalable Coherent Interface	10
3.7	Netboot over IP Multicast	10
4	LOBOS	11
4.1	Introduction	11
4.2	Netboot Overview	11
4.3	Our Requirements for Netboot	12
4.4	The New Netboot	13
4.5	How the Netboot Works	14
4.6	The LOBOS Command	16
4.7	Performance and Usability	16
4.8	Conclusion	16

5 BOOTING WINDOWS 2000 **18**

5.1	Introduction	18
5.2	Bochs	22
5.3	BIOS Dependencies in Modern Operating Systems	24
5.4	The Bootstrap Process	25
	— LinuxBIOS	25
	— ADLO	26
	— Bochs BIOS	27

CONCLUSION **28**

REFERENCES **29**

CHAPTER 1

INTRODUCTION

1.1 WHAT IS LINUXBIOS?

LinuxBIOS is a free-software project that replaces the normal BIOS code on the motherboard with the Linux kernel itself, so that a machine boots instantly into Linux within seconds of turning it on. The BIOS boot and setup is completely eliminated and replaced by a Linux kernel, which is then started, and from there on the boot proceeds as normal. Because we are using Linux the boot mechanism can be very flexible and powerful. More on that will be discussed below.

1.2 WHY DO WE NEED LINUXBIOS?

Coming to the question of why we need something like this, let us take a look at the BIOS itself.

The BIOS, or the Basic Input/Output System, is responsible for various system functions such as

- (1) Initializing the hardware
- (2) Loading the operating system
- (3) Providing a simple I/O interface to the operating system and applications

However today, operating systems have their own drivers and the BIOS is used only to load the OS.

1.2.1 PROBLEMS WITH THE PC BIOS

LinuxBIOS was designed to overcome the inherent problems of the BIOS.

- ❶ The problem with the BIOS is not only that it is **closed-source**, the problem is that it is performing a function we no longer need (**supporting DOS**), rather than functions we do need.
- ❷ The **limited nature** of the BIOS was originally imposed in the days of **8KB legacy EPROMS**. New mainboard NVRAMs can accommodate a real operating system directly on the mainboard.
- ❸ Finally, we need to have the option of doing things the BIOS writers can not imagine, such as booting over Scaleable Coherent Interface or managing the BIOS via SSH connections.
- ❹ All x86 family processors boot up into an 8086 emulation mode, running with 16-bit addresses, operators, and operands. In other words, 1000 Mhz. Pentiums on startup are emulating a 16-bit, 6 Mhz, near 25-year old processor. LinuxBIOS makes a transition to 32-bit mode immediately.
- ❺ Requirements placed upon the firmware by modern operating system kernels are minimal. The Linux kernel directly drives the hardware and does not use the BIOS. Since the Linux kernel does not use the BIOS, most of the hardware initialization is unnecessary. Modern operating systems require only basic system initialization services.
- ❻ Finally, BIOS maintenance is a nightmare. All Pentium BIOS's are **maintained via DOS programs**, and configuration settings for most

Pentium BIOS's and all Alpha BIOS's are via keyboard and display (and, in some truly deranged cases, a mouse). It is completely impractical to walk up to 1024 racked PC nodes and boot DOS on each one in turn to change one BIOS setting. Yet this impractical method is the only one supported by vendors.

- ⑦ This situation is completely unacceptable for clusters of 64, 128, or more nodes. The only practical way to maintain a BIOS in a cluster is via the network, under control of an operating system: BIOS configuration and upgrade should be possible from user programs running under an operating system. BIOS parameters should be available via /proc.

1.3 A COMPARISON: LINUXBIOS vs. PC BIOS

No.	LINUXBIOS	PC BIOS
1.	Small <32kB	Large >256kB
2.	Fast	Slow
3.	Customizable	Rigid
4.	Mostly written in C	Much assembly language
5.	Clean and portable	Complex
6.	No license fees	Per unit license fees
7.	Less manufacturer support	Better manufacturer support
8.	Non-standard	De facto standard

As a result, we now have a completely free software replacement for the BIOS that supports (without modification) either LILO or GRUB as bootloaders, and Linux, OpenBSD, and Windows 2000 as operating systems, among others. LinuxBIOS has sparked a lot of interest in the consumer electronics industry, it would only a matter of time before the 16-bit, closed-source model of BIOS's is

abandoned in favor of Open Source BIOS's with extended capabilities that modern operating systems need.

1.4 ADVANTAGES

- ❶ Using a real operating system to boot another operating system provides much greater flexibility than using a simple netboot program or the BIOS.
- ❷ Because Linux is the boot mechanism, it can boot over standard Ethernet or over other interconnects such as Myrinet, Quadrics, or SCI. It can use SSH connections to load the kernel, or it can use the InterMezzo caching file system or traditional NFS.
- ❸ Cluster nodes can be as simple as they need to be—perhaps as simple as a CPU and memory, no disk, no floppy, and no file system. The nodes will be much less autonomous thus making them easier to maintain. Also updates to the LinuxBIOS can be performed over the network.

The open source stackable BIOS, eliminates the proprietary BIOS from numerous mother-boards, and as a result, a number of new projects can be started within the BIOS space. For example

- ❶ Low level cryptographic support can now be easily added for strong preboot authentication
- ❷ Secure remote console support
- ❸ Secure configuration management

are just some of the possible new efforts that can be started now, that a general purpose open source BIOS exists

CHAPTER 2

STRUCTURE OF LINUXBIOS

The structure of the LinuxBIOS is driven by our desire to avoid writing new code. We want to build the absolute minimum amount of code needed to get Linux up, and then let Linux do the rest of the work. Linux has shown its ability to handle quirky hardware; it is doubtful we can do a better job than it can.

One initial concern for any PC BIOS is what mode to run the processor in. All x86 family processors boot up into an 8086 emulation mode, running with 16-bit addresses, operators, and operands. In other words, 1000MHz. Pentiums on startup are emulating a 16-bit, 6MHz, near 25-year-old processor. BIOS's even now have to take into account such unpleasant details as near and far pointers (see the PXE standard for some examples).

LinuxBIOS makes a transition to 32-bit mode immediately. The transition is actually a fairly simple process: the processor has to load a segment descriptor table (the so-called "global descriptor table"—the GDT) and enable memory protection. To load the GDT the processor must execute an LGDT instruction, and supply a pointer to a table descriptor in addressable memory. Fortunately there is no problem with having this table in NVRAM, so moving to protected mode in the first few instructions is no problem. In fact the sequence takes about 10 instructions.

Once the processor is in 32-bit mode it must do basic chipset initialization. While one might expect that chipset initialization would require reams of assembly code, in actuality assembly code is only needed to turn DRAM on. Once DRAM is on, C code can be used. The advantage of using C, besides the obvious

improvement in productivity, is that non-Pentium mainboards have a lot in common with Pentium mainboards, and in some cases even use the same chipsets. LinuxBIOS code can be portable between these different mainboards.

Another potential problem is that LinuxBIOS boots and runs on hardware that is completely uninitialized. LinuxBIOS can not make any assumptions about the state of any hardware; the hardware is in an indeterminate state. Linux, on the other hand, assumes that all hardware is initialized by the BIOS. While most Linux hardware initialization still works with uninitialized hardware, we are also finding that we have had to make minor changes to the kernel. For example, Linux assumes that if an IDE controller is not enabled, it is because the BIOS disabled it. On uninitialized hardware, however, the IDE controller is not enabled because there is no BIOS to enable it in the first place. Thus, 'IDE controller not enabled' has a diametrically opposite meaning in BIOS and non-BIOS environments. We have chosen to make the one-line change to the Linux IDE driver to enable IDE controllers when they are found. The change is controlled by an `#ifdef LINUXBIOS`.

In the end, the structure we arrived at is very simple. There are five major components:

- ❶ Protected mode setup
- ❷ DRAM setup
- ❸ Transition to C
- ❹ Mainboard fix-up
- ❺ Kernel unzip
- ❻ Jump to kernel.

CHAPTER 3

POST-STARTUP SCENARIOS

Let me now describe some possible uses of LinuxBIOS once it is booted. These uses include a network boot; standard boot; a diskless node boot; and maintenance activities. There are also some unique startup modes that have not been tried to date for clusters. In each case, the Linux kernel can boot another Linux kernel using the LOBOS system call, which will be discussed later.

3.1 NETWORK BOOT

Once the BIOS has booted Linux from NVRAM, the kernel take can a number of actions. One would be to establish an SSH connection to a remote DHCP configuration daemon. Using SSH represents advance over the UDP-based approach used by, e.g., bootp and PXE. The SSH-based connection can be much more secure. The SSH connection also benefits from the more stable behavior of TCP under load, as compared to a UDP-based approach. We have seen cases where 32 nodes try to netboot off of one server and only 20 succeed. PXE would have severe problems in this case as the PXE standard suggests that a node only send four packets, then give up.

The DHCP daemon can tell the node its identity and direct it as to which kernel to boot. The DHCP server can even send a kernel image over the SSH connection, and the kernel can boot it using LOBOS. In order for the kernel to use SSH, we need basic functions that are currently buried in the various SSH client programs. To address this problem we are building a library, based on the OpenSSH source, which will allow both kernels and user programs to create connections to

SSH daemons and establish encrypted TCP connections. We may also see if CIPE is useful for this purpose.

3.2 STANDARD BOOT

The DHCP daemon could send the kernel its parameters and then tell it to continue booting. In this case, the NVRAM kernel is the kernel of choice; no further kernel loading is needed. Or the kernel could boot another kernel off a local disk or other file system resource such as CDROM. This standard boot is almost identical to what is done on cluster nodes now save for two crucial differences:

- (1) The boot sequence is entirely under the control of a remote node. The boot will be about as fast, but there is much better control.
- (2) All existing boot sequences for cluster nodes rely on devices with moving parts, either floppy disks, CD-ROM's, or hard disks. Our LinuxBIOS-based boot sequence relies on devices with no moving parts, namely the NVRAM on the mainboard. If the node is told to use a hard drive and the hard drive has failed, the node can report the failure to the control node. No more guessing when a cluster node won't come up!

3.3 DISKLESS NODE

The DHCP daemon could direct the kernel to mount file systems via NFS (no problem since this is a full-featured kernel), AFS, Coda, InterMezzo, or any other network file system. The kernel could then proceed or boot a different kernel via the network file system being used. The kernel can also use many different transports for the network file system, such as MyriNet, GigaNET, and SCI. Peter

Braam enabled InterMezzo to cache an entire root file system, so one option is to cache the root file system to a RAM disk. Experiments have shown that a capable root file system can be contained in a 256MB RAM disk. Since InterMezzo supports fetch-on-demand, initially only about 50MB of files need to be loaded.

3.4 MAINTENANCE

The DHCP daemon could also direct the kernel to make an SSH port available for remote maintenance. The node would thus not even start /sbin/init, and would instead wait for instructions from a remote maintenance control program. Instructions could include changing LinuxBIOS parameters or even writing a new test kernel to NVRAM. The kernel could even load a new root file system or repartition the local disk. This maintenance model represents a major advance over what we have now. Using Linux to write to the NVRAM is tricky. If the write fails for any reason there needs to be a way to recover. For now, we are depending on the BIOS recovery NVRAM. In future, and as the on-board NVRAM grows in size, we expect to have two kernels in the NVRAM, a recovery kernel and a normal boot kernel. The bootstrap code will decide which one to run. If the recovery kernel is damaged in some way the bootstrap code can run the recovery kernel. Another possibility is to have the kernel open a port and communicate using HTTP commands. We could do BIOS maintenance with a web browser.

3.5 NETBOOT OVER MYRINET

All PC netboot standards extant (including PXE) rely on a netboot ROM running in 16-bit mode for network packet I/O. Needless to say this requirement greatly reduces the number of interfaces we can use. Since LinuxBIOS boots a true

Linux kernel, we can use high performance network interfaces such as MyriNet for the netboot process. Loading a new disk image over MyriNet would similarly be much faster than Ethernet-based disk image loading.

3.6 NETBOOT OVER SCI

This model is perhaps the most interesting. SCI (Scalable Coherent Interface) is a memory-based network, and moving data requires only a bcopy. When the kernel comes up it could configure an SCI interface. Once the kernel has the interface configured it could use SCI (via a fetch-and-add operation which only takes 6 microseconds) to notify a remote server. The remote server can bcopy a kernel image to the local node, and the local node can use LOBOS to boot this image. Or, more interesting, the remote server can bcopy a whole ramdisk image in, and the local kernel can use that image. SCI bandwidth on 32-bit PCI is about 80MB/second, so moving over a RAMDISK image could be done in a few seconds. SCI has the further advantage that configuring the interface only requires 128 bytes of configuration data, and in fact the interface can be configured from a remote controller node. LinuxBIOS doesn't have to load microcode into the interface, as it does for MyriNet.

3.7 NETBOOT USING IP MULTICAST

If the whole cluster is booting, we can use IP Multicast to distribute kernel and disk images. Most current tools that support this mode (e.g. Ghost) run under DOS. In our case, we have a full Linux kernel at our disposal and can use IP Multicast for most data, and open TCP sockets as needed to recover lost packets.

CHAPTER 4

LOBOS

4.1 INTRODUCTION

LOBOS (Linux Os Boots OS) is a system call that allows a running Linux kernel to boot a new kernel, without leaving 32-bit protected mode and, in particular, without using the BIOS in any way. This capability in turn allows Linux to be used as a network bootstrap program and even as a BIOS, both of which are used in LinuxBIOS. Here we discuss how LOBOS works, how we use it, and how LOBOS makes Linux usable as a BIOS, replacing the proprietary PC BIOS's we have today.

4.2 NETBOOT OVERVIEW

Netbooting has been around in the workstation world for many years, with perhaps the most capable systems being offered by Sun Microsystems. On a Sun system (or, nowadays, any system that runs OpenBoot firmware such as a Power Macintosh), one can simply type 'boot net' and the PROM-based bootstrap code is able to:

- ❶ Initialize the network interface
- ❷ Send out broadcast or point-to-point IP packets to locate a TFTP server
- ❸ Load a secondary bootstrap from the TFTP server

The secondary bootstrap in turn is capable of mounting NFS partitions, disk partitions, and so on to locate and load the actual kernel to boot. Net booting on

Suns has been used for almost 15 years. The protocols are open and there are many open source TFTP servers that can support Sun clients for netboot.

In the PC world the situation is not nearly as good. Even today, few PC BIOS's are capable of supporting a netboot option. Even if the BIOS understands netboot, the user often has to procure a PROM for the network card, which of course only works on that one card, and only if the card vendor has provided PROM software. Both the BIOS and the network card PROM are 16-bit 8086 code. As a result, 8086 mode operation is more important than ever. We would like to see 8086 emulation gradually grow less important and eventually disappear, but the netboot standards being promulgated by Intel and Microsoft are leading us the other way. A further problem is the nature of the standards for netboot. The network card boot model has to conform to a standard interface (NDIS2, a 16-bit Windows model) designed by Microsoft. Intel is working out the BIOS API as well as the network protocols. As a result of these two trends, PC netboot is going to be 16-bit code cleaving to a network card API-defined by Windows, using an Intel-defined BIOS API and Intel-defined protocols. Much of this code is proprietary, and using the BIOS for netboot will require us to continue relying on an 8086 assembler. We end up more dependent on 16-bit code running on an emulation of a 20-year-old processor, all of which is proprietary. This is not progress.

4.3 OUR REQUIREMENTS FOR NETBOOT

Given this undesirable situation we decided to give the problem another look, taking nothing for granted. Our goals are simple: we want to load something onto the CPU that in turn can load boot parameters over the network interface, find out what to do, and then load a kernel. Whatever it is has to be Open Source – we are no longer interested in burning proprietary binaries into PROM's. We have a few other goals:

- ❶ We don't like assembly code. Also, we have no desire to put a lot of effort into x86 assembly and then repeat our effort with, e.g., Alpha assembly. Therefore, any code we write will be C or better, unless it is impossible to escape assembly.
- ❷ We don't like code that only works for a particular Ethernet card. There are a number of packages for netboot available but their usefulness is strictly limited to a small number of cards. We want to support any network card that Linux supports.
- ❸ We don't see any point in reinventing the wheel. If there is code available that supports lots of network cards, file systems, disk types, and boot protocols, why start from scratch?
- ❹ We don't want to count on the features of any one motherboard. If a motherboard supports netboot, that's no real help, since we don't expect to use that motherboard forever.
- ❺ We want standard protocols, such as NFS, BOOTP, and so on.

4.4 THE NEW NETBOOT

It was realized that the requirements for our netboot could be met in one of two ways: we could write a new netboot program from scratch, or we could build a netboot using a minimal Linux kernel. Although there is an apparent advantage to writing our own program from scratch, experience shows that it is not a real advantage. The Sun netboot code has to support many of the same capabilities as a full-blown operating system: it has to be able to do NFS mounts, mount disk partitions, and so on. At the same time, there are many types of file systems it

can not use, such as msdos or AFS. Finally, there is no huge savings in space: the network bootstrap is 128 Kbytes. A minimal Linux kernel is 300K. Given the current cost of storage, the difference is insignificant. We decided to go with a minimal Linux kernel for our bootstrap.

4.5 HOW THE NEW NETBOOT WORKS

The new netboot works as follows. The netboot code is actually a tiny Linux kernel. It doesn't have much – basically disk, file system, network and NFS code. In the current version it does not even need to be able to run user-mode programs – it never exits kernel mode. All it has to do is the following:

- (1) Boot (eventually from NVRAM, for now from floppy, CDROM, or hard drive)
 - ❶ Contact BOOTP server and get parameters for this machine
 - ❷ Mount a remote file system via NFS or AFS; or mount the disk or floppy or CDROM.
 - ❸ Overlay the currently running kernel with the new file.

Items 1-3 exist in current Linux. The only thing missing is the ability to overlay the kernel with a new kernel. In a sense we need exec for the kernel. The steps required to support this operation are:

- ❶ In kernel mode, open the file and read it into memory. This step is done in kernel mode so that we need not depend on starting /etc/init and having a user program read the file in. In other words, a kernel can boot a new kernel without even starting any user mode programs. The file must be read into memory but not into any area of memory occupied by the

- existing kernel – the existing kernel has to keep running, so overlaying the current kernel code as the file is read in is out of the question. Overlaying the running kernel is the last step.
- ② Move critical kernel structures into a safe place. These structures must be moved out of the way when the new kernel is copied over the running kernel. So far these structures include Virtual Memory (VM) support structures such as page tables and, on the Pentium, the Global Descriptor Table (GDT); and the parameters used by the kernel when it boots to locate the root partition, as well as any arguments passed to the kernel from the boot command line. These structures will soon also include the log buffer, so that kernel printk messages are not lost on reboot.
 - ③ Turn off interrupts. This is the point of no return, so any error checking should have been done by this point.
 - ④ Switch the VM hardware over to the new page tables (and GDT, on the Pentium).
 - ⑤ Copy the final bootstrap code to a safe place where it will not be overlaid by the new kernel code. The final bootstrap code is simple: it performs a copy of the kernel to the standard location (0x100000), overlaying the currently running kernel.
 - ⑥ Jump to the final bootstrap code. The final bootstrap code copies the new kernel into the right place and jumps to it. We call this “kernel exec” LOBOS, for Linux Os Boots OS. In the next section we discuss its operation in more detail.

4.6 THE LOBOS COMMAND

Following the model of fastboot, we have created a command called lobos. Lobos puts a binary, uncompressed kernel image in /tmp, and creates a file called /lobos. We have modified the reboot script so that if the /lobos file exists, the bootfile program is invoked with the uncompressed kernel image as the argument. To reboot any kernel, the user can type the full kernel path, or simply the intermediate part of the name, e.g. the command 'lobos linux-2.2.13' will reboot the the kernel /usr/src/linux-2.2.13/vmlinix.

4.7 PERFORMANCE AND USABILITY.

Booting a kernel via LOBOS is much faster and easier than the standard BIOS-based boot. There is no long wait common with BIOS boots. The unnecessary memory test and zero is a thing of the past, as is the wait for the many unnecessary tasks that exist only to support DOS 1.0. We now have a log buffer that survives reboots and that has proven to be a major plus. We much prefer this style of booting to the 16-bit BIOS-based style used on PCs to date.

4.8 CONCLUSION

LOBOS is a system call that allows a running kernel to boot another kernel. Once a kernel is running it has no need to use the BIOS to boot other kernels. This new capability allows us to use Linux kernels as a network bootstrap, as opposed to using a special network bootstrap program. It is also very easy to boot new kernels: we simply type in 'lobos <kernel-name>' and the new kernel is up and running in less than a minute. We don't really need LILO any more.

LOBOS also makes it possible to replace the BIOS with a Linux-based BIOS. The benefits to our work are clear: the BIOS is the last great barrier to truly Open Source-based clusters. The BIOS also represents a major stumbling block to managing large clusters, due to its primitive structure and limited capabilities, as well as to its 16-bit unprotected-mode origins. We feel that LOBOS represents a first step to freeing Linux users from the BIOS and all its constraints.

CHAPTER 5

BOOTING WINDOWS 2000

5.1 INTRODUCTION

LinuxBIOS is a GPLed project started at the Cluster Research Lab, a division of the Advanced Computing Laboratory (ACL), in Los Alamos National Labs. A computer cluster is a group of computers connected to work together as parallel computer. The Cluster Research Lab performs research and development in operating systems and cluster design in order to improve the way clusters are built, managed, and used. LinuxBIOS was created to fill a need in the Computer Research Lab for an open source BIOS. Currently, x86 motherboards ship with BIOS's supplied by vendors such as Phoenix Technologies and American Mega-trends. But these BIOS's had shortcomings which made them impractical for cluster use. The ACL team found that existing BIOS's do a poor job of setting up a PC for modern OS's. They found:

- ❶ BIOS's which configure memory in a suboptimal way. For example, some BIOS's were discovered to use memory capable of CAS2 speeds at a much slower CAS3 setting.
- ❷ Incorrect configuration of PCI address spaces that open up security holes when memory-mapped cards (e.g. MyriNet) are used.
- ❸ Suboptimal assignment of interrupt requests. Some BIOS's were found to share IRQ's between multiple devices when such sharing was not

required. This sharing of IRQ's added latency upon interrupts software implicitly included the Linux kernel.

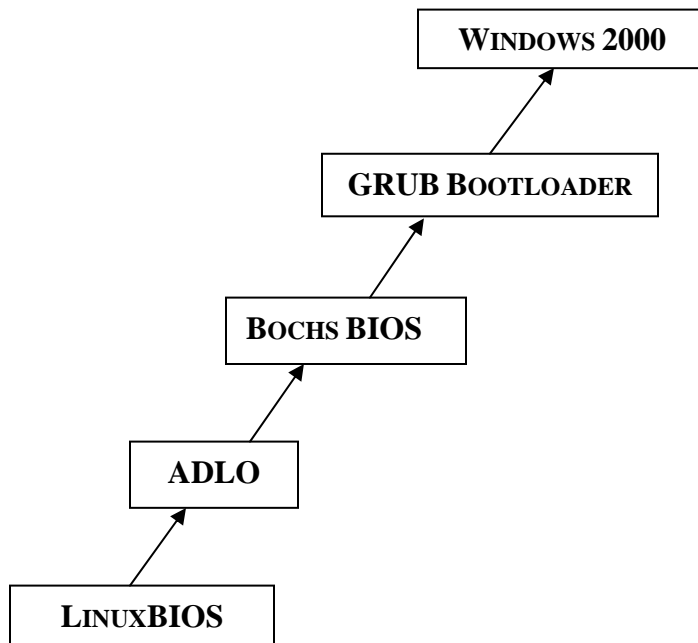


FIGURE 1: STACKABLE BIOS OVERVIEW

While the original motivation for LinuxBIOS was focused solely on improving the management of large computing clusters, numerous developers viewed LinuxBIOS as a means to provide additional features in the BIOS such as security, and support for additional operating systems beyond Linux. Here, we describe one of those efforts— an open source stackable BIOS.

The elements of the stackable BIOS are shown in Figure1. The first element is LinuxBIOS. LinuxBIOS performs all of the steps necessary to initialize the hardware on the motherboard. The next step is ADLO, the **Adhesive Loader**. This is stackable BIOS software specifically written to serve as the “glue” between the LinuxBIOS and the next element, the Bochs BIOS. Bochs is a highly portable open source x86 PC emulator which includes emulation of the x86 CPU

and common I/O de-vices. Bochs also includes a custom BIOS which provides the legacy BIOS functionality needed to boot modern operating systems such as OpenBSD, and Windows 2000 using the GNU Grub bootloader. The resulting BIOS which comes from the combination of LinuxBIOS, ADLO, and the Bochs BIOS has proven to be quite slim and fast.

LinuxBIOS, ADLO, Bochs BIOS, and even the Video BIOS (typically 64KB) can take up less than 175KB in size (Figure 2). Because alignment data pads out many of the images, a size of less than 100KB can be achieved. In the remainder of this paper, we present the details of how elements of two relatively mature open source projects (LinuxBIOS and Bochs) were combined with additional GPL'd software written by the authors to completely eliminate the need for a proprietary legacy BIOS.

ADLO & BOCHS BIOS & VIDEO BIOS	0x00
FREE SPACE	0x21000
LINUXBIOS	0x36E00
ASSEMBLY STARTUP CODE	0x3FE00
	0x40000

FIGURE 2: TYPICAL LINUXBIOS + ADLO + BOCHSBIOS EEPROM MAP FOR 512K

There are also structural problems with BIOS's that derive from the requirements for supporting DOS. One of these is the requirement that the BIOS zero memory. This requirement makes post-mortem debugging virtually impossible, as a reset results in erasing the memory image of the previously running operating system.

Commercial BIOS's also have problems accommodating custom built hardware, and can take considerable time to boot. Finally, when bugs or errors crop up in a commercial BIOS, it is almost impossible to fix them. These short-comings made it obvious that an open source BIOS such as LinuxBIOS is needed, especially for the fast paced innovations which can make the next generation of clusters successful. To accomplish its task, LinuxBIOS is placed in the computer's EEPROM chip (Electrically Erasable Programmable Read Only Memory) where normally a vendor supplied BIOS would reside. LinuxBIOS completely replaces the vendor BIOS; no fragment of the vendor BIOS is left once LinuxBIOS is installed. The LinuxBIOS binary itself is small, typically only 36 kilobytes in size depending on configuration choices. Then, a Linux kernel is placed in the EEPROM chip to act as a bootloader (Figure 3).

The Linux kernel is used for this task because it supports a wide array of hardware that can be used to acquire the binaries necessary for further booting operating systems. The Linux kernel is well suited for this task because it is, for the most part, written to avoid a need for legacy BIOS services. This lack of dependencies reduces the number of services LinuxBIOS must provide and thus results in a small and compact BIOS.

With the complexity and latency of commercial BIOS's removed, the LinuxBIOS developers discovered they were able to accomplish the entire bootstrap process in a matter of seconds. Pretty soon, the LinuxBIOS developers found there was demand not only from those building clusters, but also from a computer enthusiast community, and even more notably, motherboard vendors who were hopeful of a future where they wouldn't have to pay a royalty to BIOS vendors.

This created several problems for the LinuxBIOS developers if they wished to create an open source BIOS for a wider audience.

- ❶ Most motherboards ship with 256Kbyte EEPROM parts. Even under extreme compression, the Linux kernel could not fit in such a small flash.
- ❷ LinuxBIOS was kept small and fast by not supporting legacy PC BIOS services. The Linux Kernel used with LinuxBIOS had to be slightly modified to avoid using these legacy services. Unfortunately, many of the other operating systems that motherboard vendors and computer enthusiasts want to run require these services. It is not practical to modify the BIOS interface of each of these operating systems. This is especially true for operating systems which are closed source.

LINUX KERNEL	0x00
LINUXBIOS	0x76E00
ASSEMBLY STARTUP CODE	0x7FE00
	0x80000

FIGURE 2: TYPICAL LINUXBIOS EEPROM MAP USING A LINUX KERNEL FOR A 512K PART

5.2 BOCHS

We were able to solve both of these problems by taking advantage of another open source project, Bochs. Bochs is an LGPLed project, sponsored by MandrakeSoft, which serves as a highly portable x86 emulator written in C++.

Bochs is a true, complete emulator in that it fully emulates an x86 CPU, common I/O devices such as floppy and hard drives, and an x86 BIOS. The component of Bochs which we found useful in our plans to add PC BIOS services to LinuxBIOS was its BIOS. The Bochs BIOS had many attributes which made it especially appealing for our needs.

- ❶ None of the hardware emulated by Bochs is implemented in their BIOS layer. The BIOS was designed to interact with the hardware through true device calls and aimed to conform to up to date standards. This was advantageous to us in that few modifications were needed for the Bochs BIOS to utilize the real hardware on our target platform.
- ❷ Since the hardware is emulated, it does not need to be “turned on” like on a real x86 platform. This means Bochs does not implement the process of initializing such things as CPU cache, IDE controllers, etc. These services are already provided by LinuxBIOS for the motherboards it supports. Further, these attributes meant that the modifications we found necessary to make to Bochs were prime candidates for inclusion back into the Bochs source.

All the changes we needed to make to Bochs were merely more correct implementations of the BIOS interrupts it provided. These modifications did not break the Bochs BIOS' compatibility with their emulated hardware, and allowed it to work more correctly with the real hardware on our platform as fewer assumptions were made. This means that further maintenance of the BIOS layer of our solution can stay within the Bochs source and LinuxBIOS can continue to leverage itself effortlessly off of the further hard work of the Bochs PC emulator developers.

5.3**BIOS DEPENDENCIES IN
MODERN OPERATING SYSTEMS**

The BIOS services required to boot some modern operating systems are plentiful. Most are trivial and hardly worth mentioning, such as probing for keyboard presence or the timer interrupt. But there are four main services which proved critical to booting both operating systems like Windows 2000 and the Linux Kernel 2.4 series. These services comprised

- ❶ The video BIOS functions
- ❷ Hard drive services
- ❸ Memory sizing
- ❹ Providing a PCI table.

The following section outlines the steps taken to make ADLO and Bochs BIOS functionality closer to that of standard commercial BIOS's and distinguish this functionality from that of its base LinuxBIOS, see Table 2.

BIOS FEATURE NEEDED	STANDARD 2.4.X LINUX KERNEL	LINUX KERNEL 2.4.X MODIFIED FOR LINUXBIOS	WINDOWS 2000	WINDOWS XP	FREEBSD
Int13 Handling	Yes	No	Yes	Yes	Yes
Int15 ax=E820	Yes	No	Yes	Yes	Yes
PnP BIOS	No	No	No	No	No
PCI Table	Yes	Yes	Yes	Yes	Yes
Video BIOS	No	No	Yes	Yes	Yes
ACPI	No	No	No	No	No
APM	No	No	No	No	No

TABLE 1: OPERATING SYSTEM DEPENDENCES ON BIOS INTERRUPT FUNCTIONALITY

FEATURE	STANDARD PC BIOS	LINUXBIOS USING LINUX KERNEL	LINUXBIOS USING ETHERBOOT	LINUXBIOS USING ADLO & BOCHS BIOS
Int13 Handling	Yes	No	No	Yes
Int15 ax=E820	Yes	No	No	Yes
PnP BIOS	Yes	No	No	No
PCI Table	Yes	Yes	Yes	Yes
Video BIOS	Yes	Yes	Yes	Yes
ACPI	Yes	No	No	No
APM	Yes	No	No	No

TABLE 2: FEATURES PROVIDED BY A VARIETY OF BIOS CONFIGURATIONS

5.4 THE BOOTSTRAP PROCESS

Though LinuxBIOS and the Bochs BIOS are an open source take on the PC BIOS, neither intended to be a drop in replacement for commercial BIOS's. By stacking and executing the LinuxBIOS, ADLO, and Bochs BIOS components in order, the foundation is in place for an open source BIOS to finally be a viable alternative to commercial BIOS's on many x86 PCs. A description of the tasks performed by each component in this advantageous configuration follows.

5.4.1 LINUXBIOS

The guiding philosophy of LinuxBIOS is simple: "Let the Linux kernel do it." In other words, if Linux can perform some task such as device initialization, there is no need for any other software to do that work. We have found that in most cases Linux will redo work that the BIOS did anyway; or, still worse, Linux has to redo work that the BIOS did since in so many cases the BIOS gets it wrong. The

BIOS should only do the bare minimum of work that Linux can't do. By doing the bare minimum, LinuxBIOS becomes fast and small.

In the procedure of booting up the computer to a usable state, LinuxBIOS performs many tasks. First among these tasks is to set up and initialize memory; and set up the serial consoles so debugging information is available. Once the DRAM is initialized, LinuxBIOS can continue in C code. This is a welcome feature when faced with thousands of lines of x86 assembly. From this point, LinuxBIOS can set up Memory Type Range Registers (MTRRs) on the CPU(s), making the cache of the CPU(s) available and thus speeding up execution considerably. LinuxBIOS is also responsible for creating an IRQ routing table, and initializing individual hardware components on the motherboard. These often include an IDE controller, keyboard, southbridge, etc. Again, only the bare minimum initialization is done; Linux does the rest.

When LinuxBIOS is finished initializing hardware, it then looks in the contents of the ROM to find a next stage in the boot process. With the traditional approach of LinuxBIOS, this would be a Linux Kernel. This configuration is what provides such phenomenal records such as 3 seconds from power-on to a bash prompt. There are however other extensions to LinuxBIOS which make full use of its modular design. A popular choice among these would be to load Etherboot, which gives the bootstrap process the ability to retrieve the next stage in the bootstrap process (either another component to further enhance the boot strap process, or the final operating system itself).

5.4.2 ADLO

Indeed, instead of executing a Linux Kernel at this stage, we want to load the Bochs BIOS to provide standard commercial PC BIOS interrupt support. In order to do this effectively, we built a small wrapper program around the Bochs BIOS to

transfer valuable information from LinuxBIOS to the Bochs BIOS without having to make modifications to the Bochs BIOS. We have named this small wrapper ADLO, the “ADhesive LOader” ADLO allows us to leverage the capabilities of the Bochs BIOS without making modifications to it. ADLO is responsible for making sure the ROMs that make up the Bochs BIOS and the VGA BIOS are stored at the expected addresses. It also performs the task of copying the Bochs BIOS from its original location into “Shadow RAM.” In addition, LinuxBIOS stores some tables (such as a memory map and the IRQ routing table) in a format designed to be practical across architectures, but not conforming to the format in which they’re normally stored on a commercial PC BIOS. ADLO converts these tables back to a format easily understood by the Bochs BIOS as it is implemented presently. In the same vein, the Bochs BIOS was written for the Bochs emulation environment, and was written to emulate an AMI BIOS. To accomplish this goal, configuration of the Bochs BIOS is done by storing and retrieving configuration data in the CMOS as a real BIOS would do. ADLO is responsible for storing some of this data in the CMOS before executing the Bochs BIOS for parameters such as primary boot device and floppy size.

5.4.3 BOCHS BIOS

The primary job of the Bochs BIOS is to set up the Interrupt Vector Table, and supply an entry point for each of its BIOS services. The interrupt vector table is stored from memory location 0000:0000h up to 0000:03FCh and contains a maximum of 256 vectors, each 4 bytes wide. For instance, if we wanted to save the interrupt vector for our newly implemented Interrupt 13, the vector would be saved as the 19th entry (13h = 19) in the interrupt vector table. Since each interrupt vector is 4 bytes wide, the vector for Int13 will be stored at address 0x4C (19 * 4). The interrupt handler for Interrupt 13 is placed at offset 0xE3FE within the BIOS image. The BIOS image is placed at segment 0xF000 in

memory. Therefore, the four bytes at this offset can merely contain the segment (0xF000) as the high four bytes and the offset (0xE3FE) as the low four bytes.

After the Bochs BIOS has established its interrupt vector table, we rely on a good foundation of BIOS interrupt services to insure that everything continues to run smoothly.

5.5 CONCLUSION

LinuxBIOS is a small BIOS that completely replaces the standard BIOS with a simple bootstrap that loads Linux from NVRAM and starts it up. LinuxBIOS makes new types of configuration, maintenance, and startup models possible that were not practical to date. LinuxBIOS is widely used in clusters today and is considered invaluablely essential for the construction of maintainable clusters. Also, vendor-support for LinuxBIOS is growing by the day, and it is possibly only a matter of time before the 16-bit, closed-source model of BIOS's is abandoned in favour of free software BIOS's with extended capabilities that modern operating systems need.

While open source operating systems on the PC have flourished, the same can not be said for the firmware or BIOS. The reasons for this are many, but the lack of an open source and general purpose BIOS has limited innovation in an important space of the personal computer.

REFERENCES

- ① **LinusBIOS Home Page**
<http://www.linuxbios.org>
 - ② Ron Minnich, James Hendricks, Dale Webster
The Linux BIOS.
The 4th Annual Linux Showcase and Conference, Atlanta, October 2000.
<http://www.acl.lanl.gov/linuxbios/papers/als00/linuxbios.pdf>
 - ③ Ron Minnich
LOBOS: Booting a Kernel in 32-Bit Mode
The 4th Annual Linux Showcase and Conference, Atlanta, October 2000.
<http://www.acl.lanl.gov/linuxbios/papers/als00/lobos.pdf>
 - ④ Adam Sulmicki
A Study of BIOS Interrupts as used by Microsoft Windows 2000.
<http://www.missl.cs.umd.edu/Projects/sebos/winint/index1.html>
 - ⑤ Adam Sulmicki
A Study of BIOS Interrupts as used by Microsoft Windows XP
<http://www.missl.cs.umd.edu/Projects/sebos/winint/index2.html>
 - ⑥ **The LinuxBios Status Guide**
<http://www.linuxbios.org/status/index.html>
 - ⑦ **The Etherboot Project.**
<http://www.etherboot.org>
-