USENIX Association

Proceedings of the 4th Annual Linux Showcase & Conference, Atlanta

Atlanta, Georgia, USA
October 10–14, 2000

LOBOS: (Linux OS Boots OS) Booting a kernel in 32-bit mode

Ron Minnich

August 14, 2000

Advanced Computing Lab, Los Alamos National Labs, Los Alamos, New Mexico

Abstract

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 we are working on now. In this paper we discuss how LOBOS works, how we use it, and how LOBOS makes Linux usable as a BIOS, replacing the proprietary PC BIOSes we have today. [1]. LOBOS has been used by two other groups as a reference implementation for their Linux-boots-Linux system calls. One of these other implementations, bootimg, may become a part of the 2.4 kernel.

1 Introduction

At the ACL we have built Linux clusters of 64 nodes, and most recently have built a larger cluster of 128 nodes. While we currently use the 'magic floppy' approach for loading and reloading cluster nodes, we know that this approach will not scale to even 256 nodes – it takes far too much time and effort to put floppies into 256 nodes and make sure they boot properly. We have also found that we need to have absolute control at boot-time of what the node does, even if we are not reloading or initializing the node. We might have half the cluster running a different version of Linux with a different root file system at different times. We might even let jobs in a queueing system indicate which kernel they needed to run, and part of the work of starting a job on a set of cluster nodes might be booting those nodes with the proper kernel.

To support our needs we have decided we will use a netboot-style initialization for both normal operation as well as loading and reloading the cluster nodes. Each time the node is booted we can control which kernel to run, how to get the kernel (over the network or on the disk), and what root partition to use, either local or via NFS, even though in many cases the operating system is booted from the local disk, and the root flie system is chosen from one partition of the local disk.

In this paper we will describe our approach to netboot, which is to use Linux as the bootstrap instead of a special bootstrap program. We first provide an overview of how netboot has been done in the past, how it is being done now in the Windows/PC world, and the problems with the current PC approaches. We close with a disussion of how we might extend our work and use Linux as the BIOS, and hence save a few steps and a lot of time in the booting process.

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:

1. Initialize the network interface
2. Send out broadcast or point-to-point IP packets to locate a tftp server
3. Load a secondary bootstrap from the tftp server

[1] It's a mere coincidence that many other things in New Mexico are called LOBOS too.

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 BIOSes 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 APIdefined 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.

2.1 Our requirements for netboot

Given this undesirable situtation 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 PROMs.

We have a few other goals:

1. 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.

2. 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.

3. 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?

4. 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.

5. We want standard protocols, such as NFS, bootp, and so on.

3 The New Netboot

We 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.

3.1 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, filesystem, 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)

2. Contact BOOTP server and get parameters for this machine

3. Mount a remote file system via NFS or AFS; or mount the disk or floppy or CDROM.

4. 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:

1. 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 overlayed 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.

```
        .long SYMBOL_NAME(sys_ni_syscall)               /*
streams1 */
        .long SYMBOL_NAME(sys_ni_syscall)               /*
streams2 */
        .long SYMBOL_NAME(sys_vfork)            /* 190 */
        .long SYMBOL_NAME(sys_lobos)          /* 191 */
```

Figure 1: Additional system call entry for lobos at 191 in the 2.2.13 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.

3.2 LOBOS implementation

The LOBOS implementation consists of five major pieces, resulting in the addition of 300 or so lines to the kernel. A context diff to apply these changes to a 2.2.13 kernel is available at www.acl.lanl.gov/˜rminnich. The basic pieces are as follows:

1. Entry for the *lobos* system call in arch/i386/kernel/entry.S

2. Some additions to the arch/i386/kernel/head.S to make room for the ’safe areas’ for the GDT, page tables, kernel startup parameters, and other information

3. The code to read in the new file, in kernel/sys.c

4. The code to turn off interrupts, move the processor page tables and GDT, and switch over to the new page tables and GDT, in arch/i386/kernel/process.c

5. The code to copy the new kernel to the right place and jump to it, in kernel/sys.c

We will go over each of these in turn.

3.2.1 System Call Entry Point

The system call entry point is simply an additional line to arch/i386/kernel/entry.S, as shown in 1

```
/* here begins the support for a kernel rebooting a kernel. Not all this stuff
 * is used yet. Also, at some point, the logbuffer goes here so that logs are
 * preserved across reboots
 */
ENTRY(reboot_gdt)
.org 0x7000
ENTRY(reboot_pgdir)
.org 0x8000
ENTRY(reboot_code)
/* leave padding for later use, i.e. a log buffer that survives reboot*/
.org 0x10000

.globl SYMBOL_NAME(reboot_gdt)
.globl SYMBOL_NAME(reboot_pgdir)
.globl SYMBOL_NAME(reboot_code)
/* end reboot stuff */
```

Figure 2: How the safe areas are declared in head.S

3.2.2 Safe Areas

The safe areas consist of a few additional pages at the beginning of the kernel virtual address space. The lobos bootstrap code knows not to touch these pages, and they are not used in normal kernel operation. Hence this memory represents a safe place to put data that will not be changed by either lobos or the kernel. Currently the GDT, reboot code, and kernel parameters are saved here. The code for the safe areas is shown in 2. The reboot_pgdir area is not currently used.

3.2.3 Reading in the file

The real meat of this system call is the work done to read in a file and set the kernel up for reboot. This work occurs in a few places. The first is the sys_lobos system call, which we show in 3. This function is called with a name. It first gets a copy of the file name via getname, then performs a lookup on the file.

The function has to get access to a file, which is done via the *lookup_dentry* call. We double check to make sure there is a real inode associated with the dentry, although this level of checking is probably unnecessary. The size of the file is contained in the inode structure. We allocate that amount of memory and, if the allocation succeeds, call the kernel *read_exec* function to actually read the file into

```
/* get a dentry via lookup, then use the open_private function to open
 * it, then use read_exec to read it.
 */
asmlinkage int sys_lobos(char *file)
{
  char *name;
  struct dentry *d;
  name = getname(file);
  printk("sys_bootfile: file ptr is %p\n", file);
  if (! name)
    return -EFAULT;
  printk("the name is %s\n", name);
  d = lookup_dentry(name, 0, 1 /* read  only */);
  if (d)
    {
      void *v;
      int result;
      int good = 1;
      size_t count;
      printk("good open, dentry is %p\n", d);
      if (! d->d_inode)
        good = 0;
      if (! good) printk("NO INODE!\n");
      if (good) {
          count = d->d_inode->i_size;
          printk("the size is %d\n", count);
          printk("let's try to mallo that much\n");
          v = vmalloc(count);
          if (v) {
              result = read_exec(d, 0, v, count, 1);
              printk("read result is %d\n", result);
              if (result == count)
                run_boot_file(v, count);
            }
          else printk("alloc failed\n");
        }
    }
  else
    printk("open failed, d is null\n");
  return -EINVAL;
}
```

Figure 3: Top level of the sys_lobos system call

memory. Although *read_exec* is intended for reading in executable files, it also serves perfectly for our purposes.

At this point much of the work is done. The final steps are handled by the function *run_boot_file*, which is called with a pointer to the kernel area and a size. This function is shown in 4.

This function copies the final bootstrap, *do_boot_file*, to the safe memory location. It calls *os_restart* to set up the virtual memory structures (GDT and page tables on the Pentium), and finally calls the final bootstrap code to the do actual final step of copying the new kernel over the current kernel. If anything fails, the current behaviour is to hang forever, although obviously the correct long-term behavior is to reset the machine.

3.2.4 Setting up the page tables and GDT

This work is done by the *os_restart* function. This function has to change the state of the virtual memory hardware and by its very nature represents the most machine-dependent code in LOBOS (the assembly code presented above for reserving space and system call table entries could just as easily be C code, and is in many kernels).

The main goal of this function is to move the GDT and page tables out of the way, and to do it in a way that allows the VM hardware to function until the new kernel takes over and loads the hardware with the new kernel's GDT and page tables. Currently, the GDT is put in the safe area, and the page tables are put in an area allocated in high memory. We use the allocated memory for the page tables as they can vary in size for different types of processor. Pentium-compatible processors that support 4 MByte page table entries only need one page to address 4 Gbytes of memory; processors that only support 4 Kbyte page table entries need much more space.

The steps here are as follows: store the current gdt into curgdt, so we can find out where it is. Get the pointer to the safe gdt, and copy the first page of the current gdt to it. We only need a very small part of the GDT, but for now we just grab the whole first page. Next we allocate a new page table and copy the current page table to it. Note that for now this code only works for machines with 4 MB page table entries. Next we switch to the new GDT (the sgdt instruction); and finally we switch to the new page tables. At this point the kernel can be safely overwritten by the final bootstrap. The only assembly code in this function is for very low-level hardware support.

```
void run_boot_file(void *kernel, size_t count)
{
  extern void os_restart(int);
  extern char saved_bootparams[4096];
  extern void *reboot_code;
  int result;
  unsigned long *test = kernel;
  void *setup = 0, *kernelstart, *bootsector = 0;
  size_t funcsize = ((unsigned long) end_boot_file) -
    ((unsigned long) do_boot_file);
  void *v;
  typedef int (*z)(void *v, size_t count, void *setup, void
*kernelstart,
                void *bootsector, int testonly);
  z bf;
  cli();
  kernelstart = __va(0x100000);
  v = &reboot_code;
  /* copy it out */
  memcpy(v, do_boot_file, funcsize);
  os_restart(0);
  /* copy out saved_bootparams ..*/
  printk("copying out ssaved_bootparams\n");
  memcpy(__va(0x90000), saved_bootparams, 4096);
  /* now call it */
  printk("allocated %d bytes, now call %p\n", funcsize, v);
  bf = v;
  result = (*bf)(kernel, count, setup, kernelstart, bootsec-
tor, 0);
  printk("RETURNED FROM do_boot_file: HANGING
FOREVER\n");
  while(1);
}
```

Figure 4: The run_boot_file function.

```
void os_restart(int notused)
{
  void *newgdt = 0;
  extern char *reboot_gdt;
  pgd_t *newpagedir = 0;
  unsigned long cp;
  void *gdtbase;
  int gdtsize;
  unsigned long l;
  unsigned long curpagetable;
  unsigned long x;
  int i;
  printk("os_restart ...\n");
  curgdt[0] = curgdt[1] = 0;
  __asm__ __volatile__ ("sgdt %0" : "=m" (curgdt));
  newgdt = & reboot_gdt;
  gdtsize = 4095;
  memcpy(newgdt, gdtbase, gdtsize + 1);
  /* build the new page dir that is out of the way ... */
  newpagedir = get_pgd_slow();
  if (! newpagedir) {
        printk("newpagedir allocate failed\n");
        return;
  }
  memcpy(newpagedir ,
 swapper_pg_dir, sizeof(swapper_pg_dir))
  l = (unsigned long) newgdt;
  curgdt[1] = l >> 16;
  curgdt[0] = ((l & 0xffff) << 16) | gdtsize;
  cli();
  __asm__ __volatile__ ("lgdt curgdt");
  __asm__ __volatile__ ("ljmp $0x10,
   $blahblah\nblahblah:nop\n");
  __asm__ __volatile__(
                  "movl $0x18,%eax\n"
                  "movl %eax,%ds\n"
                  "movl %eax,%es\n"
                  "movl %eax,%fs\n"
                  "movl %eax,%gs\n"
                  "movl %eax,%ss\n"
                  );
  SET_PAGE_DIR(current,newpagedir);
  return;
}
```

Figure 5: os_restart code

```
void
do_boot_file(void *v, size_t count, void *kernelstart, int
testonly)
{
  int i;
  void (*f)(void) = kernelstart;
  extern char *reboot_gdt, *get_options;
  volatile unsigned char *src = (char *) v;
  volatile unsigned char *dst = (char *) kernelstart;
  unsigned long *l;

  for(i = 0; i < count; i++, src++, dst++)
    {
      if ((dst >= &reboot_gdt) && (dst < &get_options)) {
        continue;
      }
      if (testonly) {
      }
     else {
             *dst = *src;
        }
    }
  if (testonly)
    return;
  f();
}
```

Figure 6: Final bootstrap, do_boot_file

3.2.5 Final Bootstrap

The final bootstrap copies the new kernel over the old one, skipping the safe areas.

This is a rather simple function, in essence a memcpy. The one difference is that it does not overly the region between *restart_gdt* and *get_options* (the safe area) with the new kernel. Once it has done the copy it calls the new kernel.

3.3 Calling LOBOS from user mode

The program that uses the system call is shown in 7. The program is quite minimal. It takes the name of the file and calls the system call with that name as a parameter.

```
#include <stdio.h>
#include <errno.h>
#include <syscall.h>

#define __NR_bootfile 191

_syscall1(int, bootfile, char *, name);

int
main(int argc, char *argv[])
{
  char *name =  "test";
  if (argc > 1)
    name = argv[1];
  printf("name is %p\n", name);
  bootfile(name);
}
```

Figure 7: The bootfile program

3.4 The LOBOS command

Following the model of fastboot(1), 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/vmlinux.

4 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.

5 Related work

There are two other systems which allow Linux to boot Linux: bootimg, from Werner Almesberger; and Two Kernel Monte, from Scyld Computing. They differ in philosophy from LOBOS in a few significant ways.

5.1 Bootimg

Bootimg allows a user-mode program to reboot the kernel with a new image. The user program has to read the file into memory, and then calls the bootimg system call. The kernel code is responsible for parsing the header of the file and unzipping the code.

Bootimg turns virtual memory (i.e. paging) off at some point, but leaves i386-style segments on. Turning VM off complicates a number of issues. Since the user buffer is in virtual memory, bootimg must first copy it in to physically contiguous kernel memory that can be addressed with VM off. Also, in the future kernel components may not all be in phsyically contiguous memory; we certainly do not want to count on it. Finally, on systems such as Alpha, turning off VM is tantamount to turning off all protection. Given that even the lowest-level NVRAM software on the Alpha runs with VM enabled, we are worried about any approach that involves turning VM off.

Bootimg constitutes about 1100 lines of code, of which at least 600 are architecture-dependent. There are only 40 or so lines of assembly. One issue is that bootimg does define a number of structures (such as GDTs) that need to maintained in synch with the kernel.

Bootimg can be used with the LinuxBIOS.

5.2 Two Kernel Monte

Two kernel monte (TKM) takes a very different approach to the problem. TKM at some point turns off BOTH VM (paging) AND i386-style segmentation. In order to avoid copies and the requirement for a large area of physically contiguous memory, TKM builds an internal virtual-to-physical page map so that when VM is off, TKM can still get to the new kernel image. Also, once real mode is off, TKM can call the BIOS to reset hardware that may not work properly after a reboot. TKM can not work with the LinuxBIOS, since it depends on the BIOS for a critical part of the reboot step.

5.3 Summary of the three systems

TKM is probably the most architecture-dependent of the three, and LOBOS is probably the least architecture-dependent. LOBOS is less than half the size of the others, and has only a fraction as much assembly code. Bootimg is the most polished in certain ways: it does the most thorough permission checking and ramdisk support, for example. TKM will probably work with just about any kind of video hardware, since it calls the video bios to reset the video card. TKM will probably never work with the LinuxBIOS.

All three systems deal with the problem of VM in very different ways. LOBOS keeps paging and segmentation turned on, and relies on the presence of the "safe areas" to maintain through the reboot process. LOBOS needs to relocate the GDT and page tables once. Bootimg turns VM off, and relies on the presence of physically contiguous memory in kernel mode to get around the lack of VM. Bootimg relocates the GDT four times during a boot. TKM turns VM off and relies on its own virtual-to-physical map to keep track of memory. TKM reloads the GDT once. We feel most comfortable with keeping VM turned on at all times, especially as we move to the Alpha, where there is not support for segmentation.

TKM and Bootimg require an external program to load the image. LOBOS does not; we supply such a program, but a LOBOS-equipped kernel can, given a file name, boot that file. When the LinuxBIOS boots from NVRAM, it can further boot a different kernel (e.g. at the direction of a DHCP server) without ever having to run a user-mode program.

Only LOBOS allows the kernel log buffer to survive across reboots. We have found this capability very useful, since we no longer need to wait for klogd to clean up before rebooting. We are trying to reach a 3-second reboot time, and the fewer processes we have to wait for when we reboot, the better.

In terms of security, all three systems are no more (and no less) secure than a standard reboot system call.

There is a final question: which of these systems will make it as the "standard" in the Linux kernel? While we prefer the LOBOS implementation, and especially some of the LOBOS design decisions, we believe that the standard system call for 2.4 will be bootimg. At some point we will then need to revisit portability issues, since bootimg depends very heavily (over 30% of the code, as opposed to several tens of lines in LOBOS) on aspects of i386 Linux that do not exist in other architectures.

For our own purposes we will probably continue using LOBOS. The two determinants are the ability to boot a new kernel entirely from the kernel, and the fact that the log buffer is preserved across reboots. LOBOS has also demonstrated portability across a wide range of kernels due to its simplicity.

6 Next Steps

We are working on putting a LOBOS-enabled kernel into the FLASH RAM on our Intel 440GX motherboards. We are using code from the OpenBIOS project to bootstrap our kernel into memory. The kernel we boot serves as a true network bootstrap, in that it comes up and asks a manager node what it should do, which may include simply booting from the disk. We report on the new BIOS work in a companion paper.

Our work on LOBOS has been used by other researchers. Werner Almesberger has developed bootimg, which will probably appear in the 2.4 kernel. Researchers at Scyld Computing had started a project similar to LOBOS but had gotten stuck; they were able to use our work to finish their system, Two Kernel Monte.

7 Conclusions

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.

LOBOS has to date been used as a reference by two other groups to build working system calls with similar capabilities. One of these system calls, bootimg, will probably be a part of the standard 2.4 kernel.