Linux Guru Guide - Running Linux on Real Hardware

Jan-Benedict Glaw

- -4pc - -4pc

Linux Guru Guide - Running Linux on Real Hardware

Jan-Benedict Glaw

- -4pc - -4pc

- -4pc - -4pc

Table of Contents

- -4pc - -4pc

List of Tables

- -4pc - -4pc

Chapter 1. Introduction

This book is for you if you want to learn how to correctly debug some programs. This includes programs you've got sources for, as well as programs that you don't have source code for, like proprietary programs. Also, you should read this book if you want to run linux on a Non-PC compatible architecture. It describes how to use the hardware, how to interface to it's PROM/console functions, how to start the bootloader and finally how to load the Linux kernel. There is, however, no attempt to tell you how a given distro works.

There was a simple motivation to write this guide. I'm using Linux for nearly ten years now and learned a lot. There were things I needed to know, so I started to figure out. Other pieces of information were gained during everyday's work. But after all, it was hard work. Sometimes, it was just a matter of finding appropriate descriptions on the web, sometimes it wasn't that easy because nobody cared to describe how things work. Especially using rarely used computer's firmware consoles and the process of booting a Linux kernel on those machines is not all that easy to find, just because there's not a that large userbase and because these machines are just rare.

On the other hand, this rareness and lack of documentation is what many people keeps away from running Linux on Non-PC computers. Lack of people results in more bugs on those platforms, resulting in even less people that want to use those...

- -4pc - -4pc

Chapter 2. Generic Linux Knowledge

Startup of a dynamically linked ELF executable

1. The kernel loads some bytes of the binary to start and tries to determine it's file type. Therefor, it calls a number of binary handlers (which try to detect their specific binary signature). Two are commonly called:

   First, there's the script handler. It matches if the binary starts with "#!", the so-called "shell bang". The script handler then copies the remaining rest of the very first (shell bang) line and starts whatever is mentioned there, plus the script's file name.

   Second, there is the ELF binary handler. If it finds the ELF signature, it calls the "interpreter" mentioned inside the ELF binary.

2. The ELF interpreter (it's only called an interpreter, it isn't in real life) is most of the time `/lib/ld-linux.so.2`, the dynamic loader of GNU's glibc. It's always the "library" with absolute path that is printed by "ldd `/path/to/binary`". Procedure is the same as for scripts: the "interpreter" is called with the binary to start as it's command line argument. That is, you can manually call the dynamic ELF interpreter by hand: /lib/ld-linux.so.2 /bin/ls

   The interpreter's task is to finally link the program. Remember, it's dynamically linked. That said, it needs to load the binary into proper memory locations and do the same for all needed libraries. Proper addresses are partially coded into the binary, partially they're just calculated.

3. For speed (and resource) reasons, ld-linux.so.2 doesn't just load all the binary into RAM, but uses a trick. It calls mmap() to do some kind of projection of the binary. This way, the kernel "knows" where the file's parts need to be inside RAM, but may decide to load them as they're needed.

4. After all the executable's needed parts (the executable ("text") section, the data sections, constructors and destructors, ...), ld-linux.so.2 jumps to the libc's entry point within the program. libc then starts all constructors (if there are any) and finally calls the main() function, which is the programs starting function from the programmer's point of view.

http://linuxassembly.org/articles/startup.html [http://linuxassembly.org/articles/startup.html]

Using Serial Console

A serial console will save you from lots of trouble. It is basically a redirection of all kernel messages from screen (where you'd expect them to show up) to a serial port. The tricky part is that a serial port needs some configuration and setup, which of course needs to be finished beforehand.

To start serial console output, append "console=ttyS0" (or something like that; serial port's name depends on your hardware) to the kernel's command line. If you don't set any other options, 9600 baud, 8 data bits, no parity and one stop bit is assumed. There's no handshake enabled.

Using Tape Drives

- -4pc - -4pc
Even real gurus do mistakes at some time, so they better prepare backups before files are lost in space. As HDDs get cheaper, more and more admins prefer to buy a number of large disks and simply copy all data to those. There's a real benefit: if you make them available read-only to your users, they may just copy a file from two days ago off your store without asking you for recovering a backup.

Nonetheless, tape drives are available with high capacities today, so they're still used all over the world. In turn, you'd better know how to use them. So here are some "do"'s and "don't"'s about tape cardriges and tape drives.

1. Tape drives need to be cleaned regularly. To clean them, you're expected to insert a cleaner cardrige from time to time. Some drives will signal a LED when cleaning is neccessary, some don't. Consult your drive's documentation about how often (or after how many hours of operation) cleaning is needed.

2. Linux presents you with two types of char devices for each tape drive: `/dev/st0` and `/dev/nst0`. "st" is for "SCSI Tape" (because there are virtually only SCSI tapes available which won't break every two weeks...), "0" is first tape drive. If you access the `st` device nodes, the tape will be rewinded to it's start after the device node is closed by any accessing application. The "n" in `nst` is for "non-rewinding" access, so after use, access to these nodes won't rewind the tape.

UMA - Unified Memory Architecture

Usual question: what can be made cheaper on your computer? Well, we can build a graphics controller without RAM. Let's just use main memory. We cut off a piece (some 1 to 64 MB, you can pre-select that via BIOS) and use it for the graphics controller (which is now called a "Chipset Graphics Controller", or CGC for short). Of course, main memory bandwith may be a bit slower then, compared to regular systems...

But how to really use it now? If you reserve a lot of RAM (by choosing a large amount somewhere in the BIOS), you also loose a lot of RAM, which is possibly never ever used for actually displaying graphics. If you reserve too little RAM, you can't use high-resolution displays with 16, 24 or eve 32 bpp colour depth. There's a simple way out: Let the operating system do the job. Just configure an UMA-capable graphic driver for XFree86 (eg. "i810" for for Intel 8xx CGCs) and load the kernel `agpgart` driver. (This driver actually supports different chipsets, which are all compile-time options. So better make sure it actually contains support code for your chipset!) In this combination, XFree86 will ask the agpgart kernel driver to provide a piece of RAM large enough to hold all the image data for your configured display resolution.

Using GDB via a Serial Link to Debug a Running Kernel

Needs to be written...

- -4pc - -4pc

Chapter 3. The GNU toolchain

Toolchain Anatomy

GNU binutils

Binutils are a set of low-level programs like assembler, linker, disassembler, ... It’s their task to create and manipulate binary executable files.

GNU Compiler Collection (formerly GNU C Compiler)

GCC is the "main" actor when compiling programs. It’s the compiler who translates your C, C++, ..., program source code into architecture-specific assembler instructions, which are then build up to an executable by GNU binutils.

GNU glibc

Glibc is a large blob of glue code, which tries to give it’s best to hide kernel specific functions from you. You simply use always-the-same functions like read() etc., and glibc translates it to something the kernel can execute.

This way, you can do portable programming by only using libc calls, not (specific) kernel calls.

Getting a Full Toolchain

If there’s already a Linux distribution available for your machine, just use the distribution’s supplied tools.

However, if you need to start from scratch (possibly starting with porting the GNU toolchain to your architecture), then follow this long’n’hard way...

Needs to be written...

- -4pc - -4pc

Chapter 4. Debugging for Starters

ltrace

ltrace allows you to trace calls into dynamic libraries. So if you start a program under ltrace's control, you'll mostly see this application's libc library calls. It may, of course, also use other libraries, which calls you'll also see.

Tracing an application with ltrace will help you if you've got trouble that look like problems with the resolver or if you suspect off-by-one string handling errors.

strace

strace doesn't trace library calls, as ltrace does, but system calls. System calls are done whenever a program needs to do either I/O, signal handling, interprocess communication and other more basic things like that. (Commonly, quite some of these calls are made by indirectly calling a library function, so you will see most of the system calls show up as library calls, too.)

gdb - The GNU DeBugger

gdb is a full featured debugger, which will help you to really trace a program step by step.

Conclusions

So, if you've got to debug a small's program semantics, then you're best off using `ltrace`. If you need to debug some misbehaviour in medium-sized applications, then have a try with `strace`. All other variants will either require GDB, or are just not debugable if you aren't into that program's source code.

Debugging heavy graphical applications (things like Mozilla or OpenOffice) is a real pain in the a*s. They'll do tons of useless I/O (eg. writing things to the X11 server or getting X11 events), but this massive amount of data can simply hide the small number of interesting calls...

Network all around

You'll now learn how to properly do networking. There are Good Tools ™ you can use, and Bad Tools ™ you should try to avoid. Good Tools are those that either aid you in debugging network problems or those that are secure. Bad Tools are tools that are either insecure, or known to be wrongly used.

Setting Up Networks

Add text to cover ifconfig, route, ppp (also PPP over serial cross-cable) and radvd for IPv6 networking.

Debugging Networks

Add text about tcpdump, ettercap, ethereal, nc (also known as netcat) and telnet (but only for connecting and testing SMTP/HTTP/..., not for logging in, because it's one of the Bad Tools).

Logging In Into Remote Machines And Having Fun

Describe ssh for interactive use (with password), non-interactive use (by providing a command to execute), key authentification and local/remote port forwarding.

- -4pc - -4pc

Chapter 5. Using Different Firmware Types

Alpha's SRM firmware

SRM device names

Device names start with a two bytes long type ID (floppy, SCSI block device, network card, controller, ...). Third byte is channel number (esp. interesting for scsi), 4th byte is device ID within it's channel. After that, supplemental information (like partition number) shows up.

Table 5.1. SRM devicename overview

device name	description
dka0	dk = block device, a = first (SCSI?) channel, 0 = SCSI-ID=0
dva0	dv = floppy disk, a = first floppy disk controller, 0 = first floppy drive
ewa0	ew = network card, a = first network card bus, 0 = first network card.

Alpha's AlphaBIOS

AlphaBIOS looks a bit like Win311, using graphical windows and menus.

Sun's OpenBOOT PROM

Needs to be written...

Apple's OpenFirmware

Some information about using OpenFirmware can be found here:

- Apple Technote TN1061 [http://developer.apple.com/technotes/tn/tn1061.html]
- a short OpenFirmware introduction [http://www.netneurotic.net/mac/openfirmware.html]

- -4pc - -4pc

VAXen’s firmware

http://mail-index.netbsd.org/port-vax/1996/02/19/0006.html

PA-RISC’s firmware

Needs to be written...

SGI’s ARC Console

http://lists.debian.org/debian-mips/2004/debian-mips-200404/msg00056.html [http://lists.debian.org/debian-mips/2004/debian-mips-200404/msg00056.html]

- -4pc - -4pc

Chapter 6. Boot Loaders

Normally, a computer’s firmware doesn’t directly load an operating system. Instead, it starts a boot loader, which in turn needs to load the kernel image (and possibly other parts) and starts it afterwards.

LILO - the LInux LOader

Needs to be written...

Grub

Needs to be written...

PALO - the PA-RISC LOader

Needs to be written...

milo.exe - an Alpha loader

milo is used on AlphaBIOS Alphas.

aboot - another Alpha loader

aboot is used on Alphas which offer SRM firmware.

amiload - an Amiga bootloader

amiload loads a Linux kernel from a running AmigaOS.

YaMON is used on MIPS boards

Needs to be written...

dvhtool - SGI MIPS boot loader

Needs to be written...

SILO - Sparc Image LOader

Needs to be written...

BootX - Bootloader for OldWorld Apple Macintosh (and Clones)

Bootloader that needs an installed MacOS < 10.

- -4pc - -4pc

It either runs at system startup as an Extension or later as an Application. More information can be found here: http://penguinppc.org/projects/bootx/ [http://penguinppc.org/projects/bootx/]

Quik - Bootloader for OldWorld Apple Macintosh (and Clones)

Should theoretically work without an installed MacOS, but seems quite tricky to setup.

yaBoot - Bootloader for NewWorld Apple Macintosh (and Clones)

Does not need an installed MacOS, but an extra partition of the type Apple_Bootstrap.

For more details have a look into the yaBoot-Howto: http://penguinppc.org/projects/yaboot/doc/yaboot-howto.shtml [http://penguinppc.org/projects/yaboot/doc/yaboot-howto.shtml]

- -4pc - -4pc

Chapter 7. Network Boot Protocols

Booting over the network has several advantages. First, you can have a centralized archive of boot images, second you can test things in a simple manner without risking locally saved data, third it will allow you to use NFS-Root, which is especially interesting if you work on machines whose Linux port is in an early state...

If your computer's firmware uses a proprietary boot protocol, there's not all that much to setup in the first run. However, if it uses the long-standing bootp/dhcp + tftp variant, you need to boot in two steps: first get some IP configuration, second start downloading the OS image.

Remote IP configuration

Quite a lot of non-ia32 machines do have firmware support for using the IP protocol stack. Therefor, they requite to have an useable IP configuration. Often, this can be done manually, but most machines also accept to get their IP configuration from a remote server.

BOOTP

BOOTP is a simple protocol defined by RFCs XXX and XXX and allows a server to give a specific IP address to a client (those are normally identified by their MAC address).

DHCP - Dynamic Host Configuration Protocol

DHCP is basically the same as BOOTP (they're even binary compatible), but with a lot of extensions.

RARP - Reverse ARP

A client machine may also issue a reverse ARP query to the network. Normal ARP queries try to figure out a MAC address for a given IP address. RARP does the opposite: it asks for an IP address for a given (own) MAC address.

Directed Ping

A directed ping is mostly used to configure small, embedded devices. The trick is to enter a target's MAC address (statically) into some computer's ARP cache. Then, you use this computer to ping the machine (to be configured) once. The source machine can send the ping packet because it already knows the client's MAC address. Once the client receives a non-broadcast ping, it can read it's own IP address off the ping packet.

Delivering the Boot Image to a Client Computer

TFTP

TFTP is by far the most common way to deliver a boot image in a IP-based network (possibly after using bootp or DHCP).

RBoot

RBoot (short for Remote Boot) is mostly used by early HP PA-RISC machines. Later firmware releases may also use bootp and TFTP, but I'm unsure on this...

- -4pc - -4pc

MOP - Maintainence and Operator's Protocol

MOP is one of the oldest boot protocols and used on early VAXstations as well as DECstations. It's not only about booting. You can also remotely control VAXen's and DECstation's consoles with this protocol suite, but matching client applications seem to be missing under Linux...

Configuring a General-Purpose Boot Server

Configuring dhcpd for BOOTP and DHCP

For BOOTP and DHCP support, I'd recommend to install ISC's dhcpd. It will handle both protocols.

Configuring rarpd

rarpd responds to reverse ARP requests. It uses `/etc/ethers` (or NIS lookup, but that's not commonly configured) to get an IP address or hostname. If a hostname is presented in `/etc/ethers`, the hostname is looked up regulary to get an IP address.

`/etc/ethers` uses a quite simple format. One entry per line, each consisting of an MAC address (with ':' separated, hex digits in upper case) and (separated by space or tab) IP-Address or hostname.

But before it responds (after having found a match in `/etc/ethers`), it also checks if it can find a boot image for the IP address found. This IP address is used in hex format, uppercase with no interpunctuation (like `C0A80A0F` for 192.168.10.15). rarpd tries to find this file in the `/tftpboot/` directory and only responds if it is found there. A symlink is okay, too, even a stale one...

Configuring rbootd

rbootd typically reads a file (`/etc/rbootd.conf`) containing pairs of ethernet addresses (upper/lower case is ignored, colons to separate octets, leading zero of an octet isn't needed to be noted) and a filename. This is the file which will be served (if you don't specify a file, chances are that you may specify the filename at the workstation's command prompt).

All those boot image's filenames are relative to /var/lib/rbootd, so this directory contains all boot images.

Configuring mopd

There are several mopd variants available. I suggest you to use the one you can find at XXX. This one can read the ELF object format so you don't need to convert your freshly linked kernel image into proprietary MOP image format (I'm not even sure if there are conversion programs available at all...).

Generally, mopd tries to delive files from the `/tftpboot/mop/` directory. It doesn't use the client's supplied machine type ID, but the client's MAC address, plus the suffix "`.SYS`". The MAX address is expected to be with leading zeros and hex digits in lowercase. It also accepts symlinks, so keep the images' real names readable and supply proper symbolic links for all your MOP-booting clients.

- -4pc - -4pc

Chapter 8. Linux on Real Hardware

This chapter is mostly about running Linux on non-ia32 (non-cheap and horribly broken, that is) hardware.

In the following discussion, I basically recommend the following, simple partition layout. You may want to create more partitions, but since most of the older computers aren't equipped with all that large hard disk drives, it's only a waste of needed space to have more partitions:

First, a small boot partition (mounted on /boot) to hold the kernel image(s). Note that you not only may need that for older ia32-style computers. Even other machines (Sparc32 systems come to mind) need it!

Next partition is a swap partition, size should depend on actual RAM size and the foreseen job for the box.

The third and last partition is meant to be the largest partition, containing the root filesyetem (thus, mounted on /).

Linux on Alpha Machines

Alphas do basically come along with two totally different firmware types. Most machines actually can switch between both. First, it's AlphaBIOS, which either looks like an old Windows 3.1, or like a simple ncurses-style menu-driven window-system. If you have this firmware running, you need to use PC-BIOS partition layout and you'll need a separate, small partition holding the milo boot loader as well as the kernel image.

If you've got an Alpha using SRM firmware (you'll have textmode with blue background and ">>> " prompt), you need to use BSD slices as partition layout. Also, aboot is the boot loader of choice.

Linux on VAX Machines

For VAX computers, there isn't yet a real boot loader available. Simply load the kernel image through MOP or `dd` or `cat` a properly compiled kernel image to the HDD's start, as you may be used to when creating bootable floppy disks.

Linux on HP PA-RISC Machines

Older PA-RISC machines use rboot for booting over the network. Newer machines may also support bootp/dhcp/tftp, but I haven't had access to today's PA-RISCs...

Additionally to that, PA-RISCs also boot off CD-ROM drives (and eventually even from SCSI tapes). This is particularly useful if your PA-RISC only got exotic network interfaces that you can't easily serve connectivity for (like FDDI).

Linux on MIPS Machines

Older machines only boot ECOFF kernel images, newer firmware versions support ELF images.

Linux on PowerPC (PPC) Machines

PowerPCs are a quite large family of computers. They start with small 32bit laptops and older Macintosh computers and find their end in IBM's fastest RS/6000 ("RISC SYSTEM") boxes.

Linux on PPC based Apple Macintosh computers with PCI bus

- -4pc - -4pc
Apple uses OpenFirmware.

The so called OldWorld Macs have a part of their OS in this ROM.

All the systems with translucent colored plastic cases (iMac and later) and most of the PowerBooks produced 1999 and later are so called NewWorld Macs.

To find out more about OpenFirmware follow the links in the firmware section of this document.

Apple Macintosh with OldWorld OpenFirmware

Macs with an OldWorld Open Firmware ROM include a part of MacOS in this ROM. This Firmware is not upgradeable by any Software.

The easiest way to boot Linux on these boxes is a small bootloader called BootX which is started from a running MacOS < 10.

There's no bootloader I know of, that boots Linux from MacOS X like BootX does from MacOS < 10. BootX also will not work in the classic environment of MacOS X.

Another way is using quik which seems quite tricky to setup and rendered installed systems (nearly) useless after some unsuccessfull trials installing it.

Apple Macintosh with NewWorld OpenFirmware

Macs with a NewWorld OpenFirmware cannot use BootX or quik.

On these machines yaBoot is the needed bootloader. It cannot only boot Linux, but also MacOS (X), directly. Furthermore it is able to boot from CD-ROM and even network.

For more details about configuring yaBoot have a look into the yaBoot HowTo [http://penguinppc.org/projects/yaboot/doc/yaboot-howto.shtml].

PPC64 (RS/6000) Machines

RS/6000 don't have a real bootloader yet. You just keep some free space in front of your first partition and `cat` or `dd` the kernel image directly to the HDD's first sectors. Also, kernel command line options are directly compiled into the kernel image (but can be changed with a spartanic in-kernel command line editor).

However, http://penguinppc.org/projects/yaboot/ [http://penguinppc.org/projects/yaboot/] states that it should also work on RS/6000 machines. I've not seen that yet, but it might work, though.

Linux on S/390 Mainframes

Needs to be written...

Linux on m68k Machines

Needs to be written...

Linux on Sparc Machines

Needs to be written...