Creating boot CDs

QUICK RECOVERY

BERNHARD BABLOK

Bootable Linux CDs are highly practical in case of emergency. But producing one yourself does require some knowledge about the boot procedure and the tool presented here.

Rescue diskettes are as common as sand on the beach. But they all have a whole series of drawbacks. They are always too small, slow and error-prone. And with fairly modern PCs, they are no longer needed now that it is possible to boot direct from CD. So what could be more obvious than to make your own bootable Linux CD?

Those who are saying, but such a thing already exists, are of course right (see Info box). But often these CDs are lacking one very special, absolutely vital program. There are other reasons for making your own CD. Such CDs are ideal for presentations, training courses, kiosk systems. Or to proudly show off to your best mate your own brand new KDE installation on his computer.

This article presents a procedure which is easy to use and with whose help, at very little effort, a functioning Linux installation can be transferred onto a bootable CD. The first paragraphs below, though, bring you some theory about the boot procedure itself, but are of interest anyway regardless of the topic. On this basis, there follows a description of how to use the build system to create bootable CDs.

All beginnings are easy – the boot procedure

After switching on the computer, it looks in the places defined in the Bios for executable code. Normally these are the diskette, the CD-ROM drive and the first hard disk. This executable code is very simple, since at this time there are no operating system resources available, in particular no file system. Its task is to load and to start the operating system kernel.

This simple code is also contained in the Linux kernel, hence you can copy the kernel directly onto a diskette (*dd if=bzImage of=/dev/fd0*) and start it from there. The kernel then initialises all the subsystems and starts the program file */sbin/init* on the root partition (to be precise, the following files are sought in this order: */sbin/init*, */etc/init*, */bin/init* and */bin/sh*).

The root partition is defined when the kernel is compiled (in the top level makefile) and has by default the same value as the current root partition on which the compilation is running. This value can be modified later by means of *rdev(8)* utilities. Anyone interested in the details of the boot procedure must definitely take a look in the file */usr/src/linux /init/ main.c*.

The program */sbin/init* is the primary process of a running Linux system (it has the process ID *1*). It reads its configuration file */etc/inittab* and starts, depending on the inputs, the corresponding scripts and gettys (or the Xdm for graphical log-ins).

The drawback to the procedure described is its lack of flexibility. The root partition is fixed, plus no additional parameters can be assigned to the kernel. This means that in practice almost exclusively, a two-stage procedure is used. Instead of starting the kernel directly, the Bios loads a bootloader. This then loads the kernel and transfers to it the arguments – either from a configuration file or from a command line. The commonest bootloaders (Lilo, Chos, Grub and others) can do even more. These are boot managers, with which different operating systems and/ or kernels can also be loaded.

Where are the files – the initial ramdisk

Even with the bootloader, one question remains unanswered. On a completely new system, there is no formatted root partition, so nor is there a file system with */sbin/init* and */etc/inittab*. The kernel that has just been successfully started would thus come to a stop with a kernel panic. The solution to this problem is an initial ramdisk. This is a Linux file system, which is loaded into the memory either by the kernel itself (classic ramdisk) or by the bootloader (initial ramdisk: *initrd*). The typical emergency diskette thus contains exactly two components: a kernel and a zipped file containing a complete file system.

If one is using a bootloader, two arguments are necessary for the kernel: *root=/dev/ram* and *initrd=path to file*. Without a bootloader one has to patch the kernel (again with the aid of *rdev*), in order to define the start address of the ramdisk. The last procedure, though, has now become fairly uncommon, since in this case both the kernel as well as the ramdisk has to be copied onto a blank diskette to the right offsets.

The boot procedure with this is slightly modified. Firstly, the bootloader loads the kernel and the initial ramdisk. The kernel unpacks it to a normal ramdisk and mounts it as root file system. Next – if present – the file */linuxrc* is executed.

When this program has finished, the correct partition is mounted, as described above, as root-partition and then */sbin/init* is called up. First the initial ramdisk is either unmounted from the file system using *umount* (and the memory space is released) or – if the */initrd* directory exists – remounted to */initrd*.

Bootscreen of Bernhard's bootable Linux CD.

Stocktaking with Linuxrc

The pivot of any initial installation is the program Linuxrc. It may be a shell script, but usually, in the big distributions, is a very time-consuming C program and is responsible for the partitioning, selection and installation of the packages.

For a bootable CD-ROM, Linuxrc must do three main things: depending on the existing hardware, load the right modules, find a CD-ROM drive with the boot CD and convince the kernel that the corresponding device is the right root partition. But the latter is very simple. Linuxrc has only to write the device number (which consists of major and minor numbers) of the root partition in */proc/sys/kernel/real-root-dev*.

For everything to work, the kernel must be configured and compiled with both ramdisk and initrd support. The default size of ramdisks has changed in one of the most recent kernels and is now only 4MB. This can be modified both during the kernel configuration and also via a kernel boot parameter during run time.

Creating the initial ramdisk

There are various ways to create an initial ramdisk. The necessary steps are listed in the printed listing. Firstly, a RAM device is pre-filled with zeroes, then a file system is made. After that, the RAM device is mounted completely as normal and all the necessary data is copied into the mounted directory. The content of the whole device is then copied by means of *dd* and *gunzip* compressed into a file. Depending on whether only one start diskette is to be made or a complete rescue system, the content of the disk is very simple or correspondingly comprehensive.

In the case of a rescue system the size should also be optimised such that apart from the kernel,

Listing 1: Creating a ramdisk

```
1: dd if=/dev/zero of=/dev/ram bs=1k count=2048
2: mke2fs -vm0 /dev/ram 2048
3: mount /dev/ram /mnt
4: cp -a foo/* /mnt
5: dd if=/dev/ram bs=1k count=2048 | gzip -v9
 > ramdisk.gz
```

every byte is put to good use. A well-known trick here is to write a program which acts differently depending on with which program name it is called up. If it is called up as *cat*, it acts like *cat* and so on. The individual program commands are then nothing but hard links to this program.

This saves a lot of space, because the start-up code, which every program needs, now only needs entering once. The drawback to this is that one cannot simply delete a few programs to make space for a tool of one's own on the ramdisk.

The devfs file system

Devfs is, like Proc, a virtual file system, which can be mounted by the kernel at the same time as the root file system. The great disadvantage of Devfs is that most programs cannot cope with it. So SuSE supplies a Devfs kernel patch (not yet even working properly) for the 7.0, but Yast cannot cope with a running Devfs system.

But Devfs is to be an option from kernel 2.4 on, so there will certainly be some changes here. The distributors must be assuming that systems will run with Devfs. The boot scripts of Red Hat 7 are already prepared for this.

The principle of Devfs is simple. Instead of identifying the devices by means of major and minor numbers, as is currently the case, each driver (similar to when loading the corresponding module) logs on explicitly and is then assigned its name. Contrary to today's systems, in which one can easily reach over 2000 virtual devices under /dev, with Devfs it is only the actual devices which appear there.

The advantages are obvious: a clear, structured /dev directory with meaningful names (who knows what /dev/hdj13 really means), no more administration of major and minor numbers (these are necessary so that several modules don't get tangled up) and support of hot-pluggable devices.

The greatest flame wars in kernel history were probably those to do with Devfs and for a long time it was only available as an unofficial patch. Opponents claimed, in particular, that the kernel gets bigger due to the additional administration of the devices. All the more surprising was the fact that in the course of the last developer series (kernel 2.3.x) Linus Torvalds did include Devfs in the official kernel, although with the label "experimental".

Devfs solves a problem in a generic way, which subsystems such as USB also have to solve. And high-end devices with PCI devices which can be swapped on the fly also demand a solution which allows devices to log on and off.

To run a system neatly using Devfs, it would have to support all drivers in use. But this is generally not yet the case, which is why there is a Devfs daemon, which, when accessing classic device names, converts these into the Devfs names. There will probably be a consolidation of the whole set of problems in the 2.5 kernel series, since it makes little sense to support dynamic devices at several places in the kernel.

A CD-ROM as root directory

A CD-ROM as root directory obviously has the advantage of size, but the major disadvantage, compared with a ramdisk, that it is read-only. Unfortunately a running Linux system requires write access to many different directories, sometimes as early as the start phase:

- */var*: Here for example important files are made or perpetuated under */var/run* and */var/log*.
- */etc*: In */etc/mtab*, all mounts are stored.
- */dev*: This is where pipes are created.
- */tmp*: Many programs create files or sockets here.
- */home*: This is a hotchpotch of all possible configuration files.

One possible solution would be to mount a ramdisk among each of these directories, make a file system and copy the contents of the CD into it. But one quickly realises that this will cause problems. So the kernel should mount the CD as root partition, but under */dev* there are still no devices, so they should first be created in a ramdisk from the contents of the CD. The process is similar with the directory */etc*. The program */sbin/init* reads the */etc/inittab*, but here too there are still no directories and files, since only the first script started by */sbin/init* can make the files and directories.

A ramdisk for the /var

Even if it does not work like this, this approach is not completely wrong and highly practicable for */var*, */tmp* and */home*. So as not to create three ramdisks and end up with a bit of waste, though, */tmp* is replaced by a symbolic link to */var/tmp*, and similarly */home* by a link to */var/home*. The creation of the ramdisk, the mounting under */var* and the playing in of a complete directory hierarchy (from a Tar archive) is done here at as early a stage as possible, after */sbin/init* has passed control to the first boot script (in SuSE for example it is the script */sbin /init.d/boot*).

The /proc/mounts trick

For */etc* though, we do need a different solution. Here one can use the trick of replacing the file */etc/mtab* with a symbolic link to */proc/mounts*. The last file may not contain all the information, like *mtab*, but still enough to be able to work normally. If you output both files with *cat*, you will see hardly any difference.

As the result of this trick, */etc* can stay on the CD. If write access is also needed for additional files, these could be replaced by symbolic links to files under */var*, such as *ln -s /var/etc/foo/etc/foo*.

The /dev problem

As the last remaining directory we have to create */dev*. As yet, there is no completely satisfactory solution to this. One highly efficient option is to use

the Devfs file system. What this is all about is explained in more detail in the box of the same name.

Since the time and effort spent getting a system to run, at least for the first time, with Devfs, is fairly considerable, an alternative is used in this project. In terms of memory consumption, it is certainly not ideal but on the other hand can be used without any manipulation of the installation. It exploits the fact that the initial ramdisk, as described above, is remounted on the directory */initrd*, if this directory exists. This occurs as the last action, before the new root system is mounted. If one now replaces */dev* on the CD with a symbolic link to */initrd/dev*, one has all the devices which were already available on the initial ramdisk.

The situation thus created is fairly pathological. Mounting the CD makes use of a device from the initial ramdisk. This in turn is mounted on a directory on the CD. The effect is that there is no option, during the system power down, of performing a clean dismantling of the file systems. And because Linux bars mounted CD-ROMs, you can only get to the CD again after switching off.

The bootable CD

After this excursion into the shallow end of booting, all that remains is to put together the pieces of the puzzle to make a bootable CD. What further simplifies the matter is the fact that a bootable CD to the El-Torrito standard does nothing more than emulate a diskette. So to this end, one creates a diskette with bootloader, kernel and initial ramdisk (which essentially contains only the special Linuxrc described above), copies the diskette into a file (such as *dd if=/dev/fd0 of=bootdsk.img*) and tells the burn program which file is the diskette emulation.

Under Linux, though, the last line is not quite correct. The actual burn program Cdrecord does not in fact create any CD file systems (ISO9660 file systems), as the program Mkisofs is responsible for that. It creates the file system and at the same time copies all files into it which one wants on the CD. The result is a file with a maximum of 650MB, which is transferred by Cdrecord via a CD burner onto a medium (details on this can be found in last month's CD writer test).

With a Bios which is error-free the selection of the bootloader does not come into it, since the CD now booting is emulating a booting diskette. Stupidly, though, not every Bios is error-free, with the result that the Lilo may be loaded by the CD, but it wants to use Bios commands to load the kernel from a real instead of the emulated diskette. This is why the use of Syslinux has taken over as bootloader for bootable CDs. This loader needs an (obviously immortal) DOS file system on the diskette and as a result does not find the kernel directly via the Bios.

An easier life

This points the way to a bootable Linux CD. All you need do is replace a few directories and files with symbolic links, write a little Linuxrc program, create a boot diskette and burn the whole thing onto CD. Unfortunately, a system with all the remounted directories is hard to maintain. In particular, bending */dev* can have some unpleasant side effects, if you want to boot the system which served as model again from the disk.

But since almost all steps towards a bootable CD are independent of the distribution used, it seems a good idea to create a makefile for automation purposes. A makefile, in the execution of the idea, actually turned into an entire hierarchy, though the principle remains the same. For this you will need a computer with enough space for two Linux systems: an active system for the work and a second system to serve as model for the CD.

The model system is installed and configured completely normally. Of course, one must hold back a bit, because even 650MB soon fills up. The model system is then worked on from the second partition. This splits the procedure into two parts. In the first step, only those modifications are performed which are not destructive in the sense that the system can no longer be booted. So for example, moving */home* to */var/home* is no problem at all. On the other hand potentially destructive operations are performed almost on the fly during the creation of the CD.

Anyone wanting to play around with this concept a bit can download the files from *http://www.bablokb.de/bblcd/*. Extensive documentation comes with them. The system may not be perfect yet, but the basic functions are already in place. There are also professional systems (for example Webpads), which work with bootable Linux CDs. An update of the system is no problem even for amateurs with this, as a simple change of CD handles the system update.

If you have a weakness for cool gadgets, you can also get hold of blank CDs in visiting card format. These are really expensive for just under 20MB of available space, but for a personal Linux rescue system in your trouser pocket, it's worth it. ■

The author

Bernhard Bablok works for AGIS mbH as a systems programmer in the systems management division. When he is not listening to music, cycling or walking, he is involved with topics concerning object orientation. He can be contacted at coffee-shop@bablokb.de.

Info

Bernhard's bootable Linux CD: http://www.bablokb.de/bblcd/
H.P.Anvin: The most overfeatured rescue disk ever created: http://www.kernel.org/pub/dist/superrescue/
Homepage of Gibraltar, a firewall system which can be started from a CD: http://gibraltar.vianova.at/
The bootable visiting card of Innominate: http://www.innominate.de/level2.phtml?parent=101-
A CD-based rescue system: http://rescuecd.sourceforge.net/
The classic. The most you can fit on a diskette: http://www.toms.net/rb/home.html

■