Getting Linux into Small Machines

L.C. Benschop

June 26, 2002

Contents

1 Introduction

This document serves as an instruction to build a minimal Linux system for use on rescue diskettes, installation diskettes and embedded projects.

The Bootdisk Howto from the Linux Documentation Project also describes how to build a bootable diskette, but a diskette created in this way has a few shortcomings:

- It carries a lot of ballast such as `inittab`, `getty` and `login`. For a simple to use rescue disk, these programs and their associated configuration files are not necessary.
- It contains full featured GNU shell utilities that occupy a lot of space.
- It contains the real GNU C libraries, which are very large indeed.

Therefore a boot disk created in this way contains not many useful programs and it needs a comparatively large RAM disk to load. If you have 8MB of RAM, this works, but you can forget this on a 4MB or even 2MB machine.

There are two excellent software packages that are specially designed for saving precious memory and disk space when you are on a tight budget:

- Busybox[1] is a combination of a Unix shell and many common Unix utility programs in a single executable. The size of Busybox is about half that of the typical `bash` shell and at that it already contains a few dozen Unix commands, ranging from `ls` to `gzip`. These are trimmed down versions of course, but adequate in most cases.
- uClibc[2] is a scaled down Unix library. The GNU C library is huge, has every feature on the planet and adheres to every conceivable standard on every conceivable platform. uClibc is adequate for many applications and uses much less space.

While the use of Busybox on rescue disks is common, uClibc is rarely used. There are toolkits to create Busybox based diskettes, but IMHO there is no good instructional document that describes how to create a bootable Linux diskette with both Busybox and uClibc. The `BootE` diskette[3] is a bootable Linux diskette with

[1]http://www.busybox.net
[2]http://www.uclibc.org
[3]http://www.everywhere.dk

`Busybox` and `uClibc`. There are no instructions on the site how to customize that disk to your needs.

In order to achieve huge space savings, we need to recompile everything from source.

1.1 The Mission

Suppose we have an old 386 machine in the basement that comes with only 4MB of RAM. We want to show our friends that this old beast can still run Linux.

We must be able to perform the following tasks:

- Partition a hard disk.
- Make a swap partition on the hard disk an use it.
- Create an ext2 file system on the hard disk and make it usable as the root file system.
- Mount diskettes and a hard disk.
- Edit files.

So basically we have everything to install a Linux root file system on the hard disk.

1.2 What needs to be on the diskette?

In order to get a Linux system up and running we need the following items:

- A boot loader. This program is loaded by the PC BIOS and this makes it possible to load another program, such as the Linux kernel/
- the Linux kernel. This is the heart of the operating system.
- a root file system. This is the file system that is mounted when the kernel is started. The first program that runs (typically `/sbin/init` has to be in the root file system. The root file system can exist on a diskette, it can be loaded in RAM at boot time or it can exist on the hard disk.

The root file system has to contain the following items:

- Binaries that we want to run.
- Startup scripts and other configuration files.

- Shared libraries.
- Device files.
- Mount points.

1.3 The Host and Target System

The host system is the computer on which we build the bootable diskette. It is assumed to be a fairly modern PC with a modern installation of Linux. We will assume that it is a Pentium with at least 32MB of RAM.

Also we assume that it contains a modern Linux system that contains the following software:

- Linux kernel 2.4
- recent gcc (2.95 or later)
- Loop devices (a kernel feature that allows you to mount a file system on a file instead of a block device).
- Bzip2
- a recent LILO

You will need lots of hard disk space. Around 300MB would be enough. The unpacked Linux kernel source tree alone takes around 150MB these days. Paradoxically enough the end result will fit on a single 1.44MB diskette.

The target system is an old 386 PC with 4MB of RAM or more and zero, one or two hard disks.

2 First Preparation

First create a directory where we will build the whole project. I chose the name `myboot`. I will use this name throughout the document. Type the following command in your home directory:

```
mkdir myboot
```

Obtain the following source packages and collect them into the directory `myboot`:

- The Linux kernel. In our example we take version 2.4.18, which was the latest version at the time of writing. It may be desirable to use a kernel from the 2.2 series instead as it requires less memory. Even a kernel from the 2.0 series may do the job, it's still maintained, but don't ask me for how long. Download it from the main kernel site[4]
- The small C library `uClibc`[5].
- The shell and utilities `Busybox`[6].
- The utility programs in `util-linux`. These can be found at the main kernel site in the kernel archive, the `utils` subdirectory.
- The package `e2fsprogs`[7] with programs to create and repair file systems. .

Some of these packages may already be present in your Linux distribution. For all packages except `uClibc` and `Busybox` this is very likely. But of course there may exist more recent versions.

After you have downloaded all the sources, your `myboot` directory may look like this. Of course you may have different (more recent) versions of all programs:

```
total 27776
-rw-rw-r--    1 lennartb lennartb   613464 Jun  9 11:44 busybox-0.60.3.tar.bz2
-rw-rw-r--    1 lennartb lennartb  1376698 Jun 12 19:56 e2fsprogs-1.27.tar.gz
-rw-rw-r--    1 lennartb lennartb 24161675 Jun  8 19:28 linux-2.4.18.tar.bz2
-rw-rw-r--    1 lennartb lennartb  1170790 Jun  8 19:00 uClibc-snapshot.tar.bz2
-rw-rw-r--    1 lennartb lennartb  1065574 Jun  9 16:01 util-linux-2.11r.tar.bz2
```

So let's unpack what we've got. Type the following commands when inside the `myboot` directory:

```
bunzip2 -c busybox-0.60.3.tar.bz2 | tar xvf -
gunzip -c e2fsprogs-1.27.tar.gz | tar xvf -
bunzip2 -c linux-2.4.18.tar.bz2 | tar xvf -
bunzip2 -c uClibc-snapshot.tar.bz2 | tar xvf -
bunzip2 -c util-linux-2.11r.tar.bz2 | tar xvf -
```

After this you should have the sources of the five packages, each in its own subdirectory.

[4]http://www.kernel.org
[5]http://www.uclibc.org
[6]http://www.busybox.net
[7]http://e2fsprogs.sourceforge.net

Note: the `myboot` directory is assumed to be created under a user's home directory. In several places in this text we will use an absolute path to this directory and on my system this is `/home/lennartb/myboot`. Other users should replace this with their own home directory.

Note: in this document you see many sequences of shell commands. Of course you can put them into shell scripts, so you need not retype them when you try to build a modified boot disk.

3 Building uClibc

The trickiest part to get right is probably the C library, especially because we want to use shared libraries. The space savings are tremendous and once this is done right, you can fit many more utilities on a diskette. The directory where the shared libraries exist on the target system (where they will be used) is different from the directory where they exist on the host system. Without special tricks, the binaries that are compiled with `uClibc` won't run on the host system.

First create the following subdirectories under the `myboot` directory.

- `uclibc-dev` is the directory that contains everything you need to compile programs with `uClibc`. Int contains the include files for the `uClibc` library and special versions of `gcc` and similar programs. In fact it is a kind of cross-compiler, albeit for the same processor architecture.
- `rootfs` is the directory where everything goes that will be on your bootable diskette.

Next `cd` into the `myboot/uClibc` directory. There run the following command:

```
ln -s extra/Configs/Config.i386 Config
```

Next edit the `Config` file as follows:

- Your native compiler will probably compile for a 386. If not, you can edit the line with CROSS=
- Change the line with KERNEL_SOURCE to

  ```
  KERNEL_SOURCE=/home/lennartb/myboot/linux
  ```

  It is important that this directory matches the kernel that will eventually be used on the boot diskette.

- Check the configuration options. They look reasonable at the moment, except that you have to set `LFS = true`. Even though large file system support seems unnecessary on old 386 machines with hard disks well under 2GB, some programs will complain if it is not there.
- Below the BIG FAT WARNING: change the line to

  ```
  SHARED_LIB_LOADER_PATH=/lib.
  ```

- Change the line with DEVEL_PREFIX to

  ```
  DEVEL_PREFIX=/home/lennartb/myboot/uclibc-dev
  ```

Run the following commands to make and install the library. Note that we do *not* install the library as root as we do not install it in a system-wide directory.

```
make
make install
make PREFIX=/home/lennartb/myboot/rootfs install_target
```

The first command compiles the libraries, the second command installs the development code into the uclibc-dev directory and the last command installs the shared libraries into the rootfs directory. These will end up on the root file system of the bootable diskette.

Compiling with uClibc can be as simple as putting the `uclibc-dev` directory first in your path and just running `make`. Note that you cannot run the programs you have just made on the host system.

4 Building Busybox

First `cd` into the `myboot/busybox-0.60.3` subdirectory.

Edit the file `Conf.h` as follows:

- Add or remove support for programs you do or do not want. For each program there is a `#define BB_XXX` line that can be commented out or not. Uncomment the BB_VI line, as you would probably need an editor. Leave the network related programs commented out if you do not enable network support, otherwise uncomment them. In our example we will not use network support. It's basically your choice what to put in or not. Though it may be possible to start Linux with an interactive shell instead of init, we will leave the init program in.

- Uncomment the `BB_FEATURE_USE_TERMIOS` line.

Edit the `Makefile` as follows:

- Append `-m386` to the `CFLAGS_EXTRA` line.
- Uncomment the `CC=` line below the comment about uClibc and change it to `CC=/home/lennartb/myboot/uclibc-dev/bin/gcc`.
- You could enable LFS support, as you already have selected it in the `uClibc` library as well.

Now build the program.

```
make
make PREFIX=/home/lennartb/myboot/rootfs install
```

Because you have linked with the dynamic `uClibc` library and these are not installed in the host system's `/lib` directory, the program cannot run. There is a trick to work around it: by using the chroot command, you can run a program whose root directory is the specified directory. Become root and type the following command:

```
/usr/sbin/chroot /home/lennartb/myboot/rootfs /bin/sh
```

The shell that you are now in is the shell inside the `/home/lennartb/myboot/rootfs` directory. This shell thinks that this `rootfs` directory is in fact the root directory: even the shared libraries of `uClibc` in the `/lib` directory will be found. Type the command `ls /` and it will be clear. Exit the `chroot` subshell with Control-D and everything will be back to normal.

Now you have most common Unix utilities including an editor and a shell and you've spent only 576kB of disk space!

5 Other Essential Binaries

While Busybox offers us many essential Unix utilities, we still miss a few essential programs for our mission. We cannot partition a hard disk and we cannot create or repair ext2 file systems. Busybox can be made to include `mkfs` and `fsck` for Minix file systems, but not for the much more common Ext2 file system. Both util-linx and e2fsprogs complain if you had not built `uClibc` with large file support.

First start another shell and type the following command:

```
export PATH=/home/lennartb/myboot/uclibc-dev/bin:$PATH
```

From now on, the `uClibc` version of `gcc` will be used instead of the normal version.

For now we need util-linux only for the `fdisk` utility. The build procedure is as follows:

- `cd` into the util-linux source directory.
- Edit the `MCONFIG` file as follows:
  - Change the CPU line near the top of the file to `CPU=i386`.
  - Add another line to the large CFLAGS statement near the bottom of the file:

    ```
            -D__NO_CTYPE \
    ```

    This line must look like the other lines in the same statement, including the backslash with a space before it and nothing after it. [8]
- Run make.
- Make stops with an error while trying to build `mount`. We already have a mount in `busybox`, so we leave it that way.
- `cd` into the fdisk directory.
- Run make.
- Move the file `fdisk` to the directory `/home/lennartb/myboot/roofts/sbin`.

Now we will build e2fsprogs as follows:

- Create a directory named `build` under the e2fsprogs source directory and `cd` to it.
- Configure and build the programs: [9]

  ```
  ./configure
  make BUILD_CC=/usr/bin/gcc
  ```

[8]This option causes real functions to be used for `tolower()` and friends, instead of macros. The macros evaluate their arguments twice, where `fdisk` uses a call to an input function as argument to `tolower()`.

[9]The BUILD_CC option specifies that we want to use the normal `gcc` for building a certain program that must be run on the host system. Otherwise it would be linked with `uClibc` and would not run.

- Strip and move e2fsck and mke2fs to the `rootfs` directory.

  ```
  strip e2fsck/e2fsck.shared
  mv e2fsck/e2fsck.shared \
    /home/lennartb/myboot/rootfs/sbin/e2fsck
  strip mke2fs
  mv misc/mke2fs /home/lennartb/myboot/rootfs/sbin
  ```

This is the time to build any other programs you will need. Link them with `uClibc` and move them to the one of the binary subdirectories in the `myboot/rootfs` directory. If linking with `uClibc` does not work, try to link statically using the ordinary `gcc`.

6 Populating the Root File System

The binaries and libraries are already installed in the `rootfs` directory. Now we will complete the root file system. First create the remaining directories in rootfs.

```
cd /home/lennartb/myboot/rootfs
mkdir dev tmp etc proc mnt etc/init.d
```

Add the device nodes. We will only add the necessary device nodes: two floppy disks, two hard disks with 8 partitions each and four terminals. Further we need some memory related devices and a ram disk. Become root and cd to the `dev` subdirectory in the `myboot/rootfs` file system.

```
mknod fd0 b 2 0
mknod fd1 b 2 1
mknod hda b 3 0
mknod hda1 b 3 1
mknod hda2 b 3 2
mknod hda3 b 3 3
mknod hda4 b 3 4
mknod hda5 b 3 5
mknod hda6 b 3 6
mknod hda7 b 3 7
mknod hda8 b 3 8
mknod hdb b 3 64
mknod hdb1 b 3 65
mknod hdb2 b 3 66
mknod hdb3 b 3 67
```

```
mknod hdb4 b 3 68
mknod hdb5 b 3 69
mknod hdb6 b 3 70
mknod hdb7 b 3 71
mknod hdb8 b 3 72
mknod tty c 5 0
mknod console c 5 1
mknod tty1 c 4 1
mknod tty2 c 4 1
mknod tty3 c 4 1
mknod tty4 c 4 1
mknod ram b 1 1
mknod mem c 1 1
mknod kmem c 1 2
mknod null c 1 3
mknod zero c 1 5
```

Add files in the `/etc` subdirectory. The `init` program from `busybox` works without a login procedure, so the `passwd` and `group` files are not really needed. You could of course create single line versions for the root user and group. Even the `inittab` file is not essential and `busybox` provides a reasonable default. You are free to copy the `scripts/busybox` file from the source directory and customize it. The only file I added in `/etc` was `init.d/rcS`.

```
#!/binsh
mount -t proc none /proc
```

Make all files in the root file system owned by root:

```
chown -R 0:0 /home/lennartb/myboot/rootfs
```

Now we have a complete root file system in a directory. We still need a kernel and a way to boot. Further we need to transfer the file system to a floppy disk.

7 Building a Kernel

Now it is time to build a kernel. For the target system we will build a kernel that is different from the host system kernel. W build it under the `myboot` directory. First `cd` to the `myboot/linux` subdirectory.

The most important job is configuring the kernel. Run the following command:

```
make menuconfig
```

Instead of `menuconfig` you can use `config` (not recommended!) or `xconfig`. This will give a usable kernel for the target system.

- Processor type menu: processor family must be 386, enable math emulation, switch off everything else. Most 386 systems have no 387 coprocessor, so they do need math emulation.
- General setup menu: switch off networking support, PCI support, system V IPC and sysctl support. Support ELF binaries, other formats can be disabled.
- Code maturity, Module support, Memory Technology, Parallel port, Plug and play, Multi-device, Telephony, SCSI, I2O, Amateur radio, ISDN, Old CDROM, Input core, Multimedia, Sound, USB and kernel hacking submenus: disable everything.
- Block device submenu: support floppy, RAM disk and initial RAM disk.
- ATA/IDE/MFM/RLL submenu: support, keep everything under the ATA/IDE… block devices submenu the default.
- Character devices submenu: Support virtual terminal, console on virtual terminal, Unix 98 PTY, disable everything else.
- File systems. Keep second extended, proc and dev PTS enabled. If you want to mount DOS diskettes, enable fat, msdos and maybe vfat. If you want to mount a CDROM, enable ISO9660.
- Console drivers. Keep VGA text console enabled.
- Exit an say Yes to save changes.

Of course you must adapt the configuration to the target system you are using. If your target system has PCI, it would be better to enable it. In that case, you probably have a 486DX or a Pentium, so the math emulation may go. If you have SCSI on your target system, you should of course enable support for it and for the host adapted you are using. If you have SCSI and no IDE devices installed, you can disable ATA/IDE/MFM/RLL support.

Now we only need to build the kernel:

```
make clean
make dep
make zImage
```

The kernel described here should be around 400kB and it should work with `make zImage`. Use `make bzImage` instead if you build a kernel with more features, e.g. networking support.

8 Making a Bootable Diskette

We will describe three methods to boot Linux from a diskette.

- Booting the kernel directly from a diskette and mount a root file system on a different diskette.
- Booting the kernel with LILO and mount a root file system on a diskette (possibly the same diskette).
- Booting the kernel with LILO and add a RAM disk for the root file system.

Instead of LILO we could use another boot loader, such as SYSLINUX or GRUB. This is considered to be outside the scope of this text.

For the boot diskettes we have to format 1.44MB floppy disks or take formatted disks that may be erased. In Linux there are two programs to format a diskette. One program is `fdformat`. A diskette is formatted as follows:

```
fdformat /dev/fd0H1440
```

Some distributions have `superformat` instead. This is used as follows:

```
superformat --hd /dev/fd0
```

We assume 3.5" HD diskettes and we only format them in the standard way (1440kB), so we do not try to get a few more sectors per track or a few more tracks.

All commands in this section should be done as root.

8.1 Booting Linux directly from a diskette

We do not have to use a complicated boot loader such a LILO in order to boot a Linux system. The Linux kernel has its own primitive boot loader. When the kernel is transferred directly to a diskette with the `dd` command, the kernel can boot itself. This way we cannot supply a command line and it works only with diskettes. We also cannot use the `initrd` feature, but there is a different way of loading a RAM disk, which we will not discuss (it is discussed in the Bootdisk HOWTO).

First we have to create a file system on the root diskette and transfer the files to it. Type the following commands:

```
mke2fs /dev/fd0
mkdir /home/lennartb/myboot/rootfs/mnt
mount /dev/fd0 /home/lennartb/myboot/mnt
```

```
cp -a /home/lennartb/myboot/rootfs/* \
   /home/lennartb/myboot/mnt
umouunt /dev/fd0
```

The root file system will be mounted read-only. If this is not desired, add the following line to the file `etc/init.d/rcS` while the floppy is mounted:

```
mount -o remount /dev/fd0 /
```

Next we create the boot disk. With a different diskette in the drive type the following command:

```
dd if=/home/lennartb/myboot/linux/arch/boot/zImage \
   of=/dev/fd0
rdev /dev/fd0 /dev/fd0
```

The `rdev` command selects the device on which the root file system should be mounted by the kernel after booting. We could use the following command to make the diskette mount the root file system on the first hard disk partition.

```
rdev /dev/fd0 /dev/hda1
```

8.2 LILO bootable disk

First make the root diskette as described in the previous section. There are two ways to proceed:

- Mount the root diskette and install the kernel and LILO on it. This way we have a self-contained diskette that boots and has the root file system. It works for the kernel and the utilities as described in this article, but you would quickly run out of disk space when adding more features to the kernel or more utilities to the root file system.

- `mke2fs` a different diskette for the kernel and LILO. This way we still have two diskettes and more space on each.

The LILO diskette is either the root diskette to which we will add LILO or the separate diskette on which we will install LILO.

Mount the LILO diskette and add the kernel and boot loader to it:

```
mount /dev/fd0 /home/lennartb/myboot/mnt/
mkdir /home/lennartb/myboot/mnt/boot
cp /home/lennartb/myboot/linux/arch/i386/boot/zImage \
   /home/lennartb/myboot/mnt/boot
cp /boot/lilo/boot.b /home/lennartb/myboot/mnt/boot
```

Create the file `/home/lennartb/myboot/lilo.conf` as follows:

```
boot=/dev/fd0
install=/home/lennartb/myboot/mnt/boot/boot.b
map=/home/lennartb/myboot/mnt/boot/map
delay=100
compact
image=/home/lennartb/myboot/mnt/boot/zImage
  label=linux
  root=/dev/fd0
```

All relevant files mentioned in `lilo.conf` are on the mounted diskette. The `delay` option will wait 10 seconds, so we have the opportunity to type a command line in LILO. The `compact` option makes loading much faster. Try it without and compare.

Run LILO. This will add a `map` file to the `/boot` directory on the diskette and it will add a boot sector to the diskette. The boot sector will load the loader in boot.b, this will load the map file and the map file contains a list of sectors of the kernel, so this can be loaded.

```
lilo -C /home/lennartb/myboot/lilo.conf
```

Finally unmount the LILO diskette.

```
unount /dev/fd0
```

8.3 RAM disk

A RAM disk has the following advantages.

- Once the system is booted, programs will load faster.
- Once the system is booted, the diskette can be removed from the drive. Therefore the floppy drive can be used to load other programs or to restore a hard disk partition from backup diskettes.
- The RAM disk image can be stored compressed on the diskette. A full 1.44MB or even 2MB file system can be stored compressed on a diskette together with a kernel. A compressed 4MB or 8MB file system is also possible as long as it is not completely filled. After booting the system, you can copy programs from other diskettes into the RAM disk.

The RAM disk has the following disadvantages:

- It requires more RAM to run. The RAM disk configuration described in this system will boot on a system with 4MB of RAM, but this is pretty much the limit. More kernel features or more utilities would make this RAM disk unusable on a 4MB system.
- It requires more steps to create or to adapt the root file system.
- Changes made to the root file system are not permanent. This is either an advantage (security) or a disadvantage (configuration changes are not possible).

First we have to create an image file for the RAM disk. We limit its size to 1000K, so it still runs on a 4MB machine. Create a file system on the image file. The `-N 200` option is necessary to create enough inodes on the file system, as there are a lot of symbolic links.

```
dd if=/dev/zero of=/home/lennartb/myboot/root.img \
   bs=1k count=1000
mke2fs -F -N 200 /home/lennartb/myboot/root.img
```

Next mount the image file using a loop device and copy all files of the root file system to it. The loop option to mount makes it possible to mount a regular file as if it were a block device.

```
mount -o loop /home/lennartb/myboot/root.img \
   /home/lennartb/myboot/mnt
cp -a /home/lennartb/myboot/rootfs/* \
   /home/lennartb/myboot/mnt
umount /home/lennartb/myboot/mnt
```

Compress the root file system image. This image file will be copied to a diskette.

```
gzip -9 /home/lennartb/myboot/root.img
```

Create a new LILO diskette, mount it and add the kernel, root file system image and boot loader to it:

```
mke2fs /dev/fd0
mount /dev/fd0 /home/lennartb/myboot/mnt/
mkdir /home/lennartb/myboot/mnt/boot
cp /home/lennartb/myboot/linux/arch/i386/boot/zImage \
   /home/lennartb/myboot/mnt/boot
cp /home/lennartb/myboot/root.img.gz \
   /home/lennartb/myboot/mnt/boot
cp /boot/lilo/boot.b /home/lennartb/myboot/mnt/boot
```

Create the file `/home/lennartb/myboot/lilo-initrd.conf` as follows:

```
boot=/dev/fd0
install=/home/lennartb/myboot/mnt/boot/boot.b
map=/home/lennartb/myboot/mnt/boot/map
delay=100
compact
image=/home/lennartb/myboot/mnt/boot/zImage
  label=linux
  root=/dev/ram
  initrd=/home/lennartb/myboot/mnt/boot/root.img.gz
image=/home/lennartb/myboot/mnt/boot/zImage
  label=noram
  root=/dev/fd0
```

The configuration file is almost the same, except for the `initrd` option. This causes the boot loader to load a RAM disk image into memory. The Linux kernel will decompress it to a RAM disk device and mount this as the root file system [10]. Further we added a `noram` option, so it is also possible to use this LILO diskette without a RAM disk.

Run LILO:

```
lilo -C /home/lennartb/myboot/lilo-initrd.conf
```

Finally unmount the LILO diskette:

```
unount /dev/fd0
```

9 Using the Boot Disks

First boot Linux by putting the appropriate Linux diskette into the floppy drive.

- If you boot from a diskette without LILO, you should see a “Loading” message right away. After some disk activity you should see the message “Uncompressing Linux” and a little later you should see the kernel messages. Next it prompts for the root diskette. Remove the boot disk and insert the root diskette and press enter.

[10]The `initrd` feature makes it possible to mount the root file system twice, once on the RAM disk loaded by the boot loader and once on another device (most likely a disk partition). The RAM disk can contain an initialization routine to load the required modules to mount the root disk. This two stage root file system feature is not used here.

- If you boot from a diskette with LILO, you should see the LILO message. If you press the space bar, you should see the boot prompt from LILO, if you don't you should see the message "Loading linux" after 10 seconds. After some disk activity you should see the "Uncompressing Linux" message, some kernel messages and next the kernel will prompt for a root diskette. If the LILO diskette contains the root file system, you just press enter, otherwise you have to swap disks first.

- If you boot from a diskette with LILO and a RAM disk, everything starts the same as in the previous case. Instead of prompting you for a boot disk, the kernel should display a message that a compressed RAM disk image was found. It automatically proceeds after decompressing.

As soon as the root file system is mounted, you should see a message from Busybox. A few seconds later you are invited to press enter. After you press enter, you get a root shell prompt, which should be very familiar. With the function keys ALT-F1, ALT-F2, ALT-F3 and ALT-F4 you can switch to four virtual terminals and you should be able to get a shell prompt on all four of them.

WARNING: If your root file system is on a floppy disk (you are not using a RAM disk), do not remove this diskette from the drive when Linux is running. If you use a RAM disk or if you moved all of your root file system to the hard disk and booted with the option `root=/dev/hd??`, then it is OK to remove the diskette as soon as Linux has started.

Type the `reboot` command in order to shut the system down. If you are using a root file system on a diskette, you should wait until the PC has rebooted before removing the diskette.

What we can do is installing the linux root file system on the hard disk. First we have to (re)partition the hard disk. Type the following command:

```
fdisk /dev/hda
```

While inside `fdisk` use the `p` command to show the current partition table. Maybe you find an old DOS partition that you still want to backup. If so, quit `fdisk` now using the `q` command. If you can part with that old DOS partition, you can delete partitions with the `d` command and create new partitions with the `n` command. Finally you can write the modified partition table with the `w` command. Using `fdisk` you should be able to create the following partitions:

- A swap partition (set type to 0x82 with the `t` command) of around 16MB. Suppose this is partition 2 (`/dev/hda2`).

- A Linux partition occupying the rest of the disk. Suppose this is partition 1 (`/dev/hda1`).

After partitioning the hard disk, create a swap partition with the `mkswap` command, create an ext2 file system and copy all data to it. Type the following commands. Of course we assume that the Linux partition is `/dev/hda1` and the swap partition is `/dev/hda2`. Otherwise you must supply appropriate device names.

```
mkswap /dev/hda2
swapon /dev/hda2
mke2fs /dev/hda1
mount /dev/hda1 /mnt
mkdir /mnt/proc /mnt/mnt
cp -a /bin /sbin /usr /etc /lib /dev /tmp  /mnt
```

Edit the file `/mnt/etc/init.d/rcS`. If you haven't learned `vi` by now, tough luck.

```
#!/bin/sh
mount -a
swapon -a
```

Edit the file `/mnt/etc/fstab` as follows:

```
/dev/hda1 /     ext2 defaults 0 0
none      /proc proc defaults 0 0
/dev/hda2 swap  swap defaults 0 0
```

Now we can shut the system down.

```
umouunt /dev/hda1
reboot
```

Reboot the system with the boot diskette in the drive.

As soon as you see the word LILO, press the SHIFT key. Now you see a boot prompt. If you use a boot diskette without a RAM disk, type:

```
linux root=/dev/hda1
```

If you use a boot diskette with a RAM disk, type this at the boot prompt:

```
noram root=/dev/hda1
```

If you have a boot diskette without LILO, you have to go to the host system and change the root device using the `rdev` command:

```
rdev /dev/fd0 /dev/hda1
```

After this, reboot the target system with the modified diskette.

In any case, your system should now mount the root file system on your hard disk and your floppy drive will be free to mount other diskettes, so you can copy more files to the hard disk.

10 And Further

We showed you how to create a boot diskette with one set of features. Now you know how to do this, you should be able to customize it to your needs. You can turn it into a full featured rescue disk, an installation disk for your brand new Linux distribution, a demonstration disk or an embedded project, such as a router or print server.

First try to add kernel module support. Modules come in very handy for devices that are seldom used or are only available on some target systems. Serial ports, network adapters and SCSI features may be candidates for compiling as modules, as well as additional file systems. In order to use modules you have to do the following:

- Reconfigure the kernel with module support enabled. Configure certain features to be compiled as modules.
- Rebuild the kernel.
- Run the additional make step `make modules`.
- Copy the kernel to the boot disk (and run LILO if necessary).
- Copy the modules to the root file system or a different diskette.
- Rebuild busybox with the `insmod` and `rmmod` commands enabled. Move this to the root file system as well.
- If you use a RAM disk, you should recreate a RAM disk image, compress it and rerun LILO on your boot disk . If you hadn't made a shell script to perform all these tasks, you should do by now.

One additional feature that you might try (it only exists in 2.4 kernels) is the `devfs` file system. Instead of a `/dev` directory with hundreds of useless device node you mount a pseudo file system on the `/dev` directory, not unlike the `/proc` file system. There the device nodes appear for only the devices that exist.

After module support you may want to add network support. Once you have added an Ethernet adapter to your old 386, you can connect it to your LAN and you do not need diskettes so often. With a little bit of luck, this still runs on a 4MB machine, but forget about using a RAM disk.

- Rebuild the kernel with networking enabled. Of course you have to rebuild the modules as well.
- Rebuild Busybox with network commands enabled (ifconfig, telnet, ping etc.)

- Copy all relevant files to their respective places.
- Recreate the RAM disk image and rerun LILO if appropriate.

You can add additional programs and libraries to the target system.

- Many programs need `curses`. The `ncurses` library is reported to be usable with `uClibc`. You must recompile it in order to achieve this. You will probably need a stripped down `termcap` file on your target system as well.
- With `ncurses` and the standard libraries of `uClibc`, you should really be able to compile a lot of programs that do not require X.
- Some make files try to run the programs you have just built. This can be a problem. It should be possible to copy the `uClibc` shared libraries to the `/lib` directory of the host system, as their names are different from the `glibc` libraries. The programs linked against `glibc` will still work and the `uClibc` programs will work too. I just haven't tried this, so don't do it on a life-critical system or if you haven't made a backup.
- Even `gcc` and `X` are reported to work with `uClibc`, so give them a try! A target system with `uClibc` as the only C library should therefore be able to run a C development environment and X. Building this is certainly not for beginners.

The RAM disk version of the boot disk runs on a 4MB machine, the version without RAM disk should run with 3MB or RAM, but nothing so far has run on a machine with just 2MB of RAM. In the good old days, people ran Linux routinely on such machines and it should still be possible with modern (if not the newest) software. Some hints:

- Do not use a RAM disk.
- Remove more features from the kernel. RAM disk support can be removed, use the disk-only driver instead of the more functional IDE driver, remove file systems such as FAT and ISO9660. If you do have a 387 in that 2MB box, get rid of math emulation.
- Use an older kernel, 2.2 or even 2.0. Note: if you use 2.0, remember to use the `-O none` option on mke2fs if you make a file system for the target system. Kernel 2.0 does not understand some features added by later kernels.

- Remove more functions from busybox. This should not help much as busybox will be demand loaded. Parts that are not demanded, will not be loaded. It may help to a certain degree. Features line color `ls` and command line editing can be removed and a more primitive shell can be utilized.
- Remove init and start `/bin/sh` directly at startup.
- If this lets you do do `fdisk`, `mkswap` and `swapon`, you can get swapping enabled on the target system. From there you should be fine.

11 Other Useful Resources

This article should have shown you how to create useful bootable diskettes and minimal Linux systems with just the programs you need. You cannot be without the following resources:

- Linux from scratch[11]. If you got the taste of creating your own Linux system, one program at a time, one file at a time, compiling everything from source, then this is the next big thing. This site contains a detailed instruction how to compile a complete Linux system completely from source, including the libraries, C compiler and utilities. The target system is created on the hard disk and not on a diskette.
- Freshmeat[12]. This site contains an index of almost all programs available for Linux. Most open source programs can be found there.
- Linuxdoc[13]. This site contains all Linux related documentation you ever wanted.
- My homepage[14] contains the online version of this document.

[11]http://www.linuxfromscratch.org
[12] http://www.freshmeat.net
[13]http://www.linuxdoc.org
[14]http://www.xs4all.nl/~lennartb