Making Linux Installation Disks for Fun and Profit

L.C. Benschop

July 16, 2003

Contents

1 Introduction

In my previous article "Getting Linux into Small Machines" I described how to create a bootable Linux diskette with Busybox (shell with many built in utilities) and uClibc (a small C library). These programs save an enormous amount of memory and disk space, so that they are usable even on small machines.

In this article I want to carry the idea of creating a small boot diskette a step further. Especially I want to achieve the following goals:

- Updating the software to the latest available versions.
- Adding more functionality to the diskette. Especially I want to add the following extra features:
  - Module support, especially for SCSI and Ethernet cards.
  - Network support.
  - Curses support, especially useful for the dialog utility.
- Laying the basis for a more generalized Linux installer.

I will draw many ideas from the Debian[1] installer. Busybox was created for the Debian install diskettes in the first place and the Debian installer made heavy use of the dialog utility.

The ideas described in this article should not only be useful for installation disks, but also for special-purpose stand-alone Linux versions such as routers or print servers.

[1] http://www.debian.org

Albeit reluctantly, I will abandon the idea of targeting a 386 system with only 4MB or RAM. As even a stripped down Linux kernel needs around 3MB of memory to run and I plan to use a RAM disk of at least 2MB, the use of a RAM disk on a 4MB machine is definitely out if the installation is to be considered useful. Systems without a RAM disk would need their root file system on a diskette and that would be limited to 1.44MB (it cannot be compressed).

1.1 The Mission

We want to create an installation diskette that enables us to install a Linux distribution onto a hard disk. The system is considered to meet the following requirements:

- A 486DX CPU or later. This requirement is arbitrary. It allows us to leave FPU emulation out of the kernel and leave out special support for the 386.
- 8MB or RAM.
- A hard disk on either an (E)IDE interface or a SCSI interface.
- Either an Ethernet card or a CD-ROM drive.
  - The Ethernet card must be connected to a (local) network with a static IP address. We could add a DHCP client as well.
  - The CD-ROM must be ATAPI or SCSI.
- A normal diskette drive or the ability to boot from CD-ROM.

The Linux installer would need to pull a huge tarball from somewhere, either from a CD-ROM or from a network connection. We could devise other methods to get the tarball into the machine. We could use another operating system to put the tarball onto a hard disk partition and we could write the tarball onto a large pile of floppies and read them one by one. Especially the latter is considered unwieldy.

At the moment the system does not support PCMCIA or USB devices to install from. For laptops it means that their Ethernet cards would not be accessible. The Debian installer (version 2.1) could do this though.

The aim of this article is to assist in creating a simple boot diskette for your own Linux distribution. This is only the infrastructure for the installation diskette, not a completely working installation diskette.

What needs to be added:

- The Linux distribution itself. In its simplest form this can be a huge tarball (compressed tar archive) that contains all files that should go onto the hard

disk. If you create Linux from Scratch[2], a brand new Linux system is created on a separate hard disk partition on the host system. The contents of this partition can be put in a tarball, which can then be put on a CD-ROM and then it can be installed on a target system using this installation diskette.

Any existing Linux installation (regardless of the distribution it was originally created with) could in principle be archived and restored this way.

- A set of shell scripts (using dialog) that guides the user through the installation process. This is not strictly necessary, but if non-experts are expected to install the distribution, this is of course necessary.

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. We will make use of the GRUB boot loader, especially due to its flexibility.
- 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.

  On our boot diskette we will use an initial RAM disk (initrd), which will be loaded by the boot loader before the kernel starts.

The root file system has to contain the following items:

- Binaries that we want to run.
- Startup scripts and other configuration files.
- The installation scripts themselves.
- Shared libraries.
- Device files.
- Mount points.

[2] http://www.linuxfromscratch.org

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
- Support for the various file systems that we want to use.

You will need lots of hard disk space. Around 400MB would be enough. The unpacked Linux kernel source tree alone takes around 170MB these days. Paradoxically enough the end result will fit on one or two 1.44MB diskettes.

The target system is the system on which the diskette will be booted. It is supposed to be at least a 486 with 8MB of RAM.

2 First Preparation

First create a directory where we will build the whole project. I chose the name `myboot`. Make an environment variable with the name MYBOOT, that contains the name of this directory. It is advised that you assign this variable from a script. I will use this variable throughout the document. Type the following commands in your home directory (assuming you use the bash shell):

```
mkdir myboot
export MYBOOT=~/myboot
```

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

- The Linux kernel. In our example we take version 2.4.21, 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[3].

[3] http://www.kernel.org

- The small C library `uClibc`[4]. This can also be downloaded from the main kernel site, the kernel archive, the libs subdirectory.
- The shell and utilities `Busybox`[5].
- The utility programs in `util-linux`. These can be found at the main kernel site in the kernel archive, the `utils` subdirectory[6].
- The package `e2fsprogs`[7] with programs to create and repair file systems.
- The ncurses[8] library.
- The dialog[9] utility.
- The keyboard utilities in the kbd package. These can be found at the main kernel site in the kernel archive, the `utils` subdirectory. This is essential if you live in a country (such as Belgium or Germany) where they don't use the one true QWERTY layout.
- The GRUB[10] boot loader.

Some of these packages may already be present in your Linux distribution (in source form). 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 38572
-rw-r--r--    1 lennartb users      626209 2003-06-08 12:58 busybox-0.60.5.tar.bz2
-rw-r--r--    1 lennartb users      215806 2003-05-29 19:03 dialog_0.9b-20030308.o
rig.tar.gz
-rw-r--r--    1 lennartb users     2959524 2003-04-22 01:40 e2fsprogs-1.33.tar.gz
-rw-r--r--    1 lennartb users      877112 2003-03-24 17:14 grub-0.92.tar.gz
-rw-r--r--    1 lennartb users      819924 2003-06-08 12:36 kbd-1.08.tar.gz
-rw-r--r--    1 lennartb users    28533733 2003-07-06 13:16 linux-2.4.21.tar.bz2
-rw-r--r--    1 lennartb users     2067718 2003-03-24 17:24 ncurses-5.3.tar.gz
-rw-r--r--    1 lennartb users     1501374 2003-07-13 13:17 uClibc-0.9.20.tar.bz2
-rw-r--r--    1 lennartb users     1839967 2003-06-08 12:52 util-linux-2.11z.tar.gz
```

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

[4] http://www.uclibc.org
[5] http://www.busybox.net
[6] http://www.kernel.org/pub/linux/utils
[7] http://e2fsprogs.sourceforge.net
[8] http://www.gnu.org/directory/GNU/ncurses.html
[9] http://hightek.org/dialog
[10] http://www.gnu.org/directory/GNU/grub.html

```
bunzip2 -c busybox-0.60.5.tar.bz2 |tar xvf -
gunzip -c dialog_0.9b-20030308.orig.tar.gz |tar xvf -
gunzip -c e2fsprogs-1.33.tar.gz |tar xvf -
gunzip -c grub-0.92.tar.gz |tar xvf -
gunzip -c kbd-1.08.tar.gz |tar xvf -
bunzip2 -c linux-2.4.21.tar.bz2b |tar xvf -
gunzip -c ncurses-5.3.tar.gz |tar xvf -
bunzip2 -c uClibc-0.9.20.tar.bz2 |tar xvf -
gunzip -c util-linux-2.11z.tar.gz |tar xvf -
```

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

Note: the `$MYBOOT` shell variable must point to the myboot 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`. It 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-0.9.20` directory. There run the following command:

```
make menuconfig
```

Next set the following configuration options in the menu:

- Target architecture features and options: set target CPU to 486 and leave the rest at the defaults. Do not forget to set the correct kernel source. Fill in `$(MYBOOT)/linux-2.4.21`. MYBOOT must be in normal (round) parentheses.
- General Library settings: disable support for global constructors and destructors and profiling, leave the rest at the defaults.
- Networking support: enable RPC support, it's useful for NFS mounts.
- Library installation options. This it the trickiest part.
  - Set shared Library Loader path (first item) to `/lib`. This is the libraries will be loaded on the target diskette, so any prefix must be left out.
  - Set uClibc environment directory (second item) to `$(MYBOOT)/uclibc-dev`. This is where all files will end up that are needed by the C compiler when compiling and linking programs with `uClibc`.

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=$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-0.60.5` 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. I uncommented the following lines, some of these commands are expected to be useful in install scripts, others are necessary for network access or modules and of course I wanted the `vi` editor: BB_CMP, BB_EXPR, BB_IFCONFIG, BB_INSMOD, BB_PING, BB_RMMOD, BB_ROUTE, BB_VI and BB_WGET.

- Add or remove features that you do or do not want by uncommenting or commenting th corresponding feature line. I uncommented the following: BB_FEATURE_USE_TERMIOS, BB_FEATURE_MOUNT_NFSMOUNT and BB_FEATURE_IFCONFIG_STATUS.

Edit the `Makefile` as follows:

- Uncomment the `CC=` line below the comment about uClibc and change it to `CC=${MYBOOT}/uclibc-dev/bin/gcc`
- You could enable LFS support, as it is already selected in `uClibc` as well.

Now build the program.

```
make
make PREFIX=$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 $MYBOOT/rootfs /bin/sh
```

The shell that you are now in is the shell inside the `$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 614kB of disk space!

5 Building Curses and dialog

If you thought `uClibc` was tricky to build correctly, you will have an even harder time with ncurses. Do it exactly as I tell you and it will work, at least for the current version of the software. Set the `uClibc` directory first in your path. Do this in a subshell, so you can return by leaving that shell.

```
bash
export PATH=$MYBOOT/uclibc-dev:$PATH
```

Now configure and build ncurses:

```
./configure --with-build-cc=/usr/bin/gcc \
  --host=i686-unknown-linux --without-cxx \
  --prefix=/home/lennartb/myboot/uclibc-dev/
make
make install
```

It builds a static library only. It sucks, but at the moment I don't know a better solution.

Next go to the dialog directory in order to build the dialog utility:

```
./configure
make
strip dialog
mv dialog ../rootfs/usr/bin
```

Next comes the ugliest thing of all: ncurses expects its terminfo files in the `$MYBOOT/uclibc-dev` subdirectory, even on the target system! There are proper ways around that, I think, but for now we will simply create that directory within the target system.

```
mkdirhier $MYBOOT/rootfs$MYBOOT/uclibc-dev/share/terminfo/l
cp $MYBOOT/uclibc-dev/share/terminfo/l/linux \
        $MYBOOT/rootfs$MYBOOT/uclibc-dev/share/terminfo/l
```

The first command creates the directory under the rootfs directory and the second command copies the terminfo file into it. The linux file is the only one we need.

We are eager to test the whole thing. First become root and chroot into the rootfs environment.

```
/usr/sbin/chroot $MYBOOT/rootfs /bin/sh
```

Next type the following commands:

```
export TERM=linux
dialog --msgbox "Hello, world" 5 20
```

This looks much nicer if you try it on a text console.

6 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, we cannot select an appropriate keyboard layout 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=$MYBOOT/uclibc-dev/usr/bin:$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` and `cfdisk` utilities. The build procedure is as follows:

- `cd` into the util-linux source directory.
- Run configure and make.
- Make stops with an error while trying to build `swapon`. We already have a swapon in `busybox`, so we leave it that way.
- `cd` into the fdisk directory.
- Run the following commands.

  ```
  cp $MYBOOT/uclibc-dev/include/ncurses/curses.h \
     $MYBOOT/uclibc-dev/include
  make HAVE_NCURSES=yes LIBCURSES=-lncurses
  ```

  All the trouble is to get cfdisk compiled, so we can use it in addition to or instead of the stone age fdisk program.
- Copy the binaries fdisk and cfdisk to the `$MYBOOT/rootfs/sbin` directory.

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: [11]

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

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

```
strip e2fsck/e2fsck.shared
mv e2fsck/e2fsck.shared $MYBOOT/rootfs/sbin/e2fsck
strip misc/mke2fs
mv misc/mke2fs $MYBOOT/rootfs/sbin
```

Now we still need to be able to load keyboard definitions for foreign keyboards. Now enter the `kbd` subdirectory. Run the local configure command and then create the file `defines.h` with the following contents:

```
#define LC_ALL 0
```

Now you can make the program without errors, but apparently it is a bit buggy. Move the loadkeys binary and key maps to the rootfs directory.

```
strip src/loadkeys
mv src/loadkeys $MYBOOT/rootfs/usr/bin
mkdir $MYBOOT/rootfs/usr/share/kbd
cp -a data/keymaps $MYBOOT/usr/share/kbd
```

Now you can weed out a lot of those keymap files that we do not need. Start with removing the amiga, atari, mac and sun directories. In the i386 directories there are probably only a few maps you want to keep, one for each country that your product may need to be used. All files that are left can be compressed with gzip. I kept the keymaps for Belgium, France, Germany, UK and US (used almost exclusively in the Netherlands) and all include directories (and these are probably not even needed). If you want to test loadkeys using the familiar chroot trick, this only works on a text console and you may need some of the device nodes already in place (see next chapter).

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`.

[11] 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.

7 Populating the Root File System

The binaries and libraries are already installed in the `rootfs` directory as well as some support files, such as the key maps and the terminfo file. Now we will complete the root file system. First create the remaining directories in rootfs.

```
cd $MYBOOT/rootfs
mkdir dev tmp etc proc mnt etc/init.d target usr/scripts
```

Add the device nodes. We will only add the necessary device nodes: two floppy disks, four IDE hard disks with 8 partitions each, two SCSI hard disk with 8 partitions each, two SCSI CD-ROMs, 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 hdc b 22 0
mknod hdc1 b 22 1
mknod hdc2 b 22 2
mknod hdc3 b 22 3
mknod hdc4 b 22 4
mknod hdc5 b 22 5
mknod hdc6 b 22 6
```

```
mknod hdc7 b 22 7
mknod hdc8 b 22 8
mknod hdd b 22 64
mknod hdd1 b 22 65
mknod hdd2 b 22 66
mknod hdd3 b 22 67
mknod hdd4 b 22 68
mknod hdd5 b 22 69
mknod hdd6 b 22 70
mknod hdd7 b 22 71
mknod hdd8 b 22 72
mknod sda b 8 0
mknod sda1 b 8 1
mknod sda2 b 8 2
mknod sda3 b 8 3
mknod sda4 b 8 4
mknod sda5 b 8 5
mknod sda6 b 8 6
mknod sda7 b 8 7
mknod sda8 b 8 8
mknod sdb b 8 16
mknod sdb1 b 8 17
mknod sdb2 b 8 18
mknod sdb3 b 8 19
mknod sdb4 b 8 20
mknod sdb5 b 8 21
mknod sdb6 b 8 22
mknod sdb7 b 8 23
mknod sdb8 b 8 24
mknod sr0 b 11 0
mknod sr1 b 11 1
mknod tty c 5 0
mknod console c 5 1
mknod tty1 c 4 1
mknod tty2 c 4 2
mknod tty3 c 4 3
mknod tty4 c 4 4
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. In our example we do copy the file `scripts/inittab` from the busybox directory to `/etc` and make the following changes to it:

- Comment out the lines that contain getty.
- Change the first line containing 'askfirst' (shell on the console) as follows:

  ```
  ::respawn:-/usr/scripts/install_top
  ```

I created the file`/etc/init.d/rcS`, which must have execute permissions.

```
#!/bin/sh
mount -t proc none /proc
```

For the time being create a dummy install script `/usr/scripts/install_top` (remember to make it executable):

```
#!/bin/sh
dialog --msgbox "This can be your install script" 5 50
exit 0
```

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.

8 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. We build it under the `myboot` directory. First `cd` to the `myboot/linux-2.4.21` 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 486, switch off SMP support, leave the rest at defaults.
- General setup menu: switch off hot-pluggable devices, system V IPC and sysctl support. Support ELF binaries, other formats can be disabled.
- Parallel port support can be switched off, unless you want to enable it for PLIP networking or parallel port storage devices (ZIP disk, CD-ROM).
- SCSI: enable SCSI CD-ROM support, enable a large selection of low level drivers as modules.
- Network device support: enable a large selection of Ethernet cards as modules.
- Code maturity, Module support, Memory Technology, Parallel port, Plug and play, Multi-device, Telephony, I2O, Amateur radio, ISDN, Old CDROM, Input core, Multimedia, Sound, and kernel hacking submenus: disable everything, if it was not already disabled.
- Block device submenu: support floppy, loop device, 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 vfat. Enable iso9660, NFS (client only) and ext3. If you want to experiment with other file systems such as reiserfs, you must enable support for them.
- Console drivers. Keep VGA text console enabled.
- Exit and say Yes to save changes.

Of course you must adapt the configuration to the target system you are using are you are anticipating your target audience to use. This kernel tries to be useful for a large number of systems, from 486DX onward. The strategy is to keep things in if they enable a user to get started with an installation, so it must be possible

to access SCSI harddisks and CD-ROMs and also the local area network in case the machine does not have a CD-ROM. This kernel will be different from the one that is going to be used after installation. Support for soundcards and printers is probably unnecessary on this installation disk.

Now we only need to build the kernel:

```
make clean
make dep
make bzImage
```

The kernel described here should be around 900kB.

Next create the modules:

```
make modules
make INSTALL_MOD_PATH=$MYBOOT modules_install
cd $MYBOOT
tar zcvf modules.tar.gz lib
```

The modules will end up in a compressed tar archive, not normally stored on the main RAM disk.

9 Making a Bootable Diskette

We will use GRUB and the initial RAM disk, even though they may not be the optimum solution in all cases. Our choices were motivated as follows:

- GRUB is a modern boot loader, likely to stay around for some time. It does not depend on a special assembler (like LILO or SYSLINUX). It can be used almost everywhere and configuration files and kernels can be updated without the need to reinstall the boot loader. It *used* to be very trouble free to compile too.
- The initial RAM disk is a modern kernel feature, likely to stay around for some time. It works on CD-ROMs as well (as opposed to the kernel loaded RAM disk).
- GRUB can be instructed to read an initial RAM disk from a separate diskette, while other boot loaders cannot.

9.1 Compiling GRUB

The current version of GRUB does not compile without patching the source, at least not with the gcc-3.3 that came with Suse Linux 8.2.

First configure as follows:

```
./configure --disable-xfs --disable-reiserfs --disable-jfs \
            --disable-vstafs --disable-minix --disable-ffs
```

This will get some of the bloat out.

Next edit the file `stage2/fsys_reiserfs` (I know this file isn't even linked in) and remove the word 'long' from line 115, it should read:

```
__u32 j_mount_id;
```

This seems to be a real bug in the source code, but now we can simply make at last.

Getting 0.93 to compile is even more troublesome. The linker keeps complaining about missing `memcpy` Apart from the reiserfs bug, you have to edit the file `stage2/Makefile`. Find the line with STAGE2_CFLAGS and change it to:

```
STAGE2_CFLAGS = -Os -minline-all-stringops
```

But apply this fix only if your C compiler complains, earlier versions probably don't. If you apply the fix, do `make clean` first. Hopefully these bugs will be fixed very soon.

9.2 Creating the RAM Disk Image

Create a directory `mnt` below `$MYBOOT`. This directory is used as a temporary mount point, independent of mount points present in your Linux distribution.

Then run the following commands (put them in a shell script):

```
cd $MYBOOT
dd if=/dev/zero of=initrd.img bs=1k count=2000
mke2fs -F -N 300 initrd.img
mount -o loop initrd.img mnt
cp -a rootfs/* mnt
umount initrd.img
gzip -9 initrd.img
```

Now the contents of the root file system are contained in the compressed image file `initrd.img.gz`. This can be mounted as an initial RAM disk.

9.3 Creating the Boot Diskettes

The combined size of the kernel and the initial RAM disk are larger than a single diskette, so we have to make two diskettes. A third diskette will be needed to store the `modules.tar.gz` file.

The first diskette is the trickiest to make. It will contain the boot loader and the kernel. First put a fresh diskette into the drive and type the following commands:

```
fdformat /dev/fd0u1440
mke2fs /dev/fd0
mount /dev/fd0 mnt
mkdir mnt/boot
mkdir mnt/boot/grub
cp linux-2.4.21/arch/i386/boot/bzImage mnt/boot/kernel
cp grub-0.92/stage1/stage1 mnt/boot/grub
cp grub-0.92/stage2/stage2 mnt/boot/grub
```

Create the file `mnt/boot/grub/menu.lst` using an editor. It should contain the following lines.

```
title Linux Installation Disk
root (fd0)
kernel (fd0)/boot/kernel
pause Please insert the second diskette
initrd (fd0)/initrd.img.gz
```

Next unmount the diskette and start GRUB.

```
umount mnt
grub-0.92/grub/grub
```

Inside GRUB type the following commands:

```
root (fd0)
setup (fd0)
quit
```

After this, the diskette is bootable and contains the kernel. If you want to update the kernel, you only have to mount the diskette again and copy a new kernel file to it. You can also edit the `menu.lst` file on the diskette without the need for rerunning GRUB.

The second diskette will contain the RAM disk. Put a fresh diskette into the drive and type the following commands:

```
fdformat /dev/fd0u1440
mke2fs /dev/fd0
mount /dev/fd0 mnt
cp initrd.img.gz mnt
umount mnt
```

The third diskette will contain the modules. Put a fresh diskette into the drive and type the following commands:

```
fdformat /dev/fd0u1440
mke2fs /dev/fd0
mount /dev/fd0 mnt
cp modules.tar.gz mnt
umount mnt
```

9.4 Using the diskettes

Insert the first diskette into the floppy drive in order to boot. The GRUB menu presents you with a single item, the Linux Installation Disk. Press Enter and the boot loader will load your kernel. Insert the second diskette into the drive when the boot loader prompts you for it. Next the RAM disk will be loaded. After this, the kernel starts to decompress and print boot messages.

Next comes a screen that tells you that here could be your installation script. This does not do anything useful for now.

Press ALT-F2, ALT-F3 or ALT-F4 to switch to another virtual terminal. Press ALT-F1 to switch back to your ‘installer’. The other virtual terminals have shell prompts and if you are familiar with Linux, you should be familiar with these.

In the shell window you can mount CD-ROMs, diskettes and hard disk partitions. You have both fdisk and cfdisk to partition a hard disk. You have the `vi` editor to edit files. You can create and repair ext2 file systems. If you manage to get your Ethernet card working, there is even a possibility to mount an NFS file system or to obtain files from the Web using wget.

What about the third diskette? You can mount it with:

```
mount /dev/fd0 /mnt
```

Next you can use tar to list the contents of the `modules.tar.gz` file and to extract files from it. As the `insmod` command is also available, you can get your Ethernet card or SCSI host adapter to work, at least in theory.

10 Creating a CD-ROM

Most modern PCs can boot from CD-ROM and in the near future most computers will not even have floppy drives. Therefore we present an alternative configuration using a CD-ROM. In our method we are essentially creating three levels of disk image files:

- At the outermost level there is the iso file created by mkisofs.
- At the intermediate level there is the diskette image file that the BIOS can boot from.
- At the innermost level there is the compressed RAM disk image.

We have to start at the innermost level.

10.1 Creating the RAM disk image

Creating the RAM disk image for a CD is really the same as for a diskette, only this time we will fit the `modules.tar.gz` file into the RAM disk and hence the RAM disk will be larger. Run the following commands (put them in a shell script):

```
cd $MYBOOT
dd if=/dev/zero of=initrd.img bs=1k count=3000
mke2fs -F -N 300 initrd.img
mount -o loop initrd.img mnt
cp -a rootfs/* mnt
cp modules.tar.gz mnt/lib
umount initrd.img
gzip -9 initrd.img
```

10.2 Creating the Diskette Image

On the CD-ROM we will use a diskette image of 2.88MB. Real diskettes of this size are really rare, but most PCs can boot from a diskette image on a CD.

First prepare an image file of a 2.88MB diskette and copy GRUB to it.

```
dd if=/dev/zero of=isoboot.img bs=1k count=2880
mke2fs -F isoboot.img
mount -o loop isoboot.img mnt
mkdir mnt/boot
mkdir mnt/boot/grub
```

```
cp grub-0.92/stage1/stage1 mnt/boot/grub
cp grub-0.92/stage2/stage2 mnt/boot/grub
```

Next create the file `mnt/boot/grub/menu.lst` with the following contents:

```
title Linux Installation Disk
root (fd0)
kernel (fd0)/boot/kernel
initrd (fd0)/boot/initrd.img.gz
```

Unmount and run GRUB:

```
umount mnt
grub-0.92/grub/grub
```

Inside GRUB type the following commands:

```
device (fd0) isoboot.img
root (fd0)
setup (fd0)
quit
```

Now the diskette image (hopefully) contains a working boot loader. Keep a copy of this image, so you can add your kernels and RAM disk images later.

Finally copy your RAM disk and kernel to it:

```
mount -o loop isoboot.img mnt
cp linux-2.4.21/arch/i386/boot/bzImage mnt/boot/kernel
cp initrd.img.gz mnt/boot
umount mnt
```

This diskette image is already quite full, so this spells the worst for newer software versions or extensions. As a first measure we can remove the Ethernet drivers from `modules.tar.gz`. If we can boot from a CD-ROM, we may as well assume we have a CD-ROM and we can install from there. Otherwise you can consider the `isolinux` CD boot loader, which does not use diskette images, but can use kernels and RAM disk images anywhere on the CD.

It's worth considering the removal of both module support and network support from the kernel and to compile most common SCSI host adapters directly into the kernel. Functionality related to modules and networking can then also be removed from Busybox.

10.3 Creating the Bootable CD

First create a directory tree for the ISO image.

```
mkdir iso
mkdir iso/boot
mkdir iso/data
```

Next copy some files to the `data` subdirectory (in our example we use a hypothetical file `distro.tar.gz` and copy the diskette image to the `boot` directory.

```
cp distro.tar.gz iso/data
cp isoboot.img iso/boot
```

Now create the ISO image.

```
mkisofs -o bootcd.iso -b boot/isoboot.img \
        -c boot/boot.catalog -r iso
```

Finally burn it to a CD-ROM. Use the appropriate device ID.

```
cdrecord dev=0,1,0 -eject -pad -data bootcd.iso
```

This CD is bootable and from the booted Linux you should be able to mount the CD-ROM to access the data files.

11 Creating the Install Scripts

The diskettes and CD-ROM created so far, do not contain any installation scripts. They are usable as-is if you only need to install your freshly created Linux from Scratch distribution, but for a distribution that must be installed by unexperienced people, installation scripts are essential.

11.1 Example Installation Scripts

This section gives an overview of a set of example installation scripts. Their functionality is inspired by the Debian installer. These scripts are really simplified and should not be considered suitable for production use. The use of `dialog` and moderately complex shell programming are central to these installation scripts. All scripts are located in the `/usr/scripts` directory in the RAM disk and all must have execute permission.

Each invocation of the `dialog` command represents a visible screen that is presented to the user. Depending on the command type (e.g. `--msgbox` or

`--menu` a different type of screen is shown. The output of most commands (the actual selection) is printed on the standard error device `stderr`. The redirect `2>$TEMPNAME` causes this data to be stored in a temporary file. Using a `cat` command with back quotes we can put the contents of that temporary file in a variable. Note that many `dialog` commands are extended across multiple lines and the backslash must be the very last character on a line.

The top level script `install_top` sets a few variables that will be used throughout the installation. Next it starts with an introduction screen and it continues with the menu, which is repeated indefinitely. After each menu selection, a command is invoked, whose name is the concatenation of `install_` and the selection. Therefore we can call eight scripts from the top level script: `install_keyboard`, `install_partition`, `install_swap`, `install_filesys`, `install_modules`, `install_install`, `install_configure` and `install_reboot`.

```
#!/bin/sh
# Example top level install script.
export TEMPNAME=/tmp/choice
export SCRIPTDIR=/usr/scripts
export SOURCEDIR=/mnt
export TARGETDIR=/target

dialog --clear --msgbox  \
  "Welcome to the Linux Installation Disk\n
Press ENTER to start installation.\n
Press ALT-F2, ALT-F3 or ALT-F4 for a shell prompt.\n
\n
Please make sure that all your disks are backed up\n
before you start installation" 18 60

while true
do
dialog --clear --menu "Linux Installation Disk Main Menu" \
        18 60 8 \
        keyboard "Select keyboard layout" \
        partition "Partition a disk with cfdisk" \
        swap "Select a swap partition" \
        filesys "Create a file system" \
        modules "Load kernel modules" \
        install "Install Linux " \
        configure "Configure Linux " \
```

```
        reboot "Reboot the system " 2>$TEMPNAME
SELECTION=‘cat $TEMPNAME‘
$SCRIPTDIR/install_$SELECTION
done
```

The script `install_keyboard` selects one of a few keyboard layouts. The interesting thing is that it also stores the name of the selected map file into the file `/tmp/keyfile`. Form there, the configuration script can adjust the keyboard layout selection in the target system, so when the target system is rebooted, it will have the correct keyboard layout enabled as well.

```
#!/bin/sh
#Keyboard configuration script.

KEYDIR=/usr/share/kbd/keymaps/i386

dialog --clear  --menu "Select Keyboard Layout" 18 60 5\
    US "Standard US layout (common in the Netherlands)" \
    UK "UK layout" \
    DE "German layout (QWERTZ)" \
    FR "French layout (AZERTY)" \
    BE "Belgian layout (AZERTY)"  2>$TEMPNAME

SELECTION=‘cat $TEMPNAME‘

case $SELECTION
in
US ) KEYFILE=$KEYDIR/qwerty/us.map.gz ;;
UK ) KEYFILE=$KEYDIR/qwerty/uk.map.gz ;;
DE ) KEYFILE=$KEYDIR/qwertz/de.map.gz ;;
FR ) KEYFILE=$KEYDIR/azerty/fr.map.gz ;;
BE ) KEYFILE=$KEYDIR/azerty/be-latin1.map.gz ;;
esac
# Save name of keyboard file for later.
echo -n $KEYFILE /tmp/keyfile
loadkeys $KEYFILE
```

The script `install_partition` lets the user select a hard disk and invokes `cfdisk`.

```
#!/bin/sh
#Partition a hard disk.
```

```
dialog --clear --menu "Select a hard disk to partition"\
     18 60  6 \
     hda "Primary IDE master" \
     hdb "Primary IDE slave" \
     hdc "Secondary IDE master" \
     hdd "Secondary IDE slave" \
     sda "First SCSI" \
     sdb "Second SCSI" 2>$TEMPNAME
SELECTION=‘cat $TEMPNAME‘
cfdisk /dev/$SELECTION
```

The script `install_swap` lets the user select a swap partition. It initializes the input field with the string `/dev/`. Next it asks for confirmation, as `mkswap` does a really destructive job to the selected partition. It records the partition name in the temporary file `/tmp/swappart`. The use of an input box to select a partition is not the most user-friendly way to do this job. Ideally the script would read the partition table and put those partitions with type 0x82 in a menu list to select from, but this is more complex.

```
#!/bin/sh
# Select and install a swap partition

dialog --inputbox "Enter the name of your swap partition"\
        5 60 /dev/ 2>$TEMPNAME
SELECTION=‘cat $TEMPNAME‘

dialog --yesno \
   "Any data on $SELECTION will be erased forever!\n
Are you really sure you want to continue?" 18 60

if [ $? = 0 ]
then
   echo -n $SELECTION >/tmp/swappart
   mkswap $SELECTION
   swapon $SELCTION
fi
```

The script `install_filesys` is almost a copy of the `install_swap` script. Apart from the user-friendliness issue, this script should really be extended to allow multiple file system partitions, each on its own mount point.

```
#!/bin/sh
# Select and install a root partition

dialog --inputbox "Enter the name of your root partition"\
        5 60 /dev/ 2>$TEMPNAME
SELECTION=‘cat $TEMPNAME‘

dialog --yesno \
   "Any data on $SELECTION will be erased forever!\n
Are you really sure you want to continue?" 18 60

if [ $? = 0 ]
then
   echo -n $SELECTION >/tmp/rootfs
   mke2fs $SELECTION
fi
```

The script `install_modules` is not implemented yet. It should offer a way to access the `modules.tar.gz` file, it should offer a selection of all available modules and next it should selectively extract the selected modules from the tar file. Finally it should ask the user for module parameters and run `insmod` on the specified module. This is left as an exercise to the reader. Likewise there should be an `install_network` script to configure the network.

The script `install_install` lets the user select a CD-ROM drive to install from. In the real world there should be a way to install from other sources as well, such as the network (nfs, wget) or an existing hard disk partition. Next it mounts the CD-ROM on the source directory. Next it asks for the name of the root partition. The script tries to be smart and already suggests the saved partition name in `/tmp/rootfs` if it exists. It mounts the partition on `/target` and proceeds to extract the giant tarball `distro.tar.gz` that contains everything from the distribution.

```
#!/bin/sh
# This installs the Linux system onto the hard disk.

dialog --menu "Select CD-ROM to install from" 18 60 6 \
     hda "Primary IDE master" \
     hdb "Primary IDE slave" \
     hdc "Secondary IDE master" \
     hdd "Secondary IDE slave" \
     sr0 "First SCSI" \
     sr1 "Second SCSI" 2>$TEMPNAME
```

```
SELECTION=‘cat $TEMPNAME‘
mount -r -t iso9660 /dev/$SELECTION $SOURCEDIR
if [ $? != 0 ]
then
   dialog --msgbox "Mount failed!" 18 60
   exit 1
fi
if [ -f /tmp/rootfs ]
then
   ROOTFS=‘cat /tmp/rootfs‘
else
   ROOTFS=/dev/
fi
dialog --inputbox "Select root partition" 18 60\
       $ROOTFS 2>$TEMPNAME
ROOTFS=‘cat $TEMPNAME‘
echo -n $ROOTFS >/tmp/rootfs
mount -t ext2 $ROOTFS $TARGETDIR
if [ $? != 0 ]
then
   dialog --msgbox "Mount failed!" 18 60
   umount $SOURCEDIR
   exit 1
fi
cd $TARGETDIR
tar zxvf $SOURCEDIR/data/distro.tar.gz
```

The `install_reboot` script could not be simpler. At least it asks for confirmation.

```
#!/bin/sh
# Reboot script

dialog --yesno "Ready to reboot?" 18 60
if [ $? = 0 ]
then
  reboot
fi
```

11.2 Configuring Linux

What remains is the `install_configure` script. What it does is really dependent on the actual distribution that you installed. As a minimum it should do the following things:

- Set the default keyboard mapping depending on what you specified in the `install_keyboard` script.
- Create an `/etc/fstab` file containing the selected root and swap partitions.
- Make the system bootable. This is described in the next subsection.

Other things the configuration script could do:

- Set a root password.
- Set the time zone.
- Perform configuration of the network or to transfer the install-time configuration to the target system.
- Configure the kernel modules or to transfer the install-time module configuration to the target system.

After extracting the base system (the big tarball), you have the following options to proceed:

- Complete the installation using only programs that are on the RAM disk. There are only a few things that you absolutely have to de before you can reboot.
- Remain running from the RAM disk but also use programs and scripts that are on the freshly created hard disk partition. Binaries should be specially linked against `uClibc`.
- Perform a `chroot` into the root file system on the hard disk and run the second part of the install script from there. Once you have chroot-ed into the hard disk, you can normally use all shared libraries. The user need not be aware of the transition.
- Achieve the same result as before by using pivot_root. With pivot_root we can jump out of the initial RAM disk into the real root file system. We need to arrange that the post-install configuration script is run from init, but only the first time the system is run (the script removes itself from the init scripts). This has the advantage that the RAM of the RAM disk can be reclaimed.

In the latter two cases you are no longer limited by the programs and libraries available in the RAM disk. Without an intervening reboot you can continue to install and configure X, all your network services or just about anything. It is even possible to continue to use the system for production without a single reboot. This is not wise, because you want to test that the system is capable of rebooting.

11.3 Making the System Bootable

After the tarball `distro.tar.gz` is extracted, the root file system contains all files that make up the Linux distribution, including the kernel. A few files (such as `/etc/fstab` depend on your local situation and should be adjusted by the configuration script. Even then the system is not yet bootable[12]. Making the system bootable is an essential step that the configuration script must perform.

Assume that the kernel on the hard disk contains all drivers necessary to mount the root file system from the hard disk (all necessary SCSI drivers). If you go for a minimal kernel, you will probably need to create a custom initial RAM disk, one that loads the necessary modules. Some distributions contain the `mkinitrd` script.

Making the target system bootable consists of the following steps:

- Creating LILO/GRUB config file. The name of the root file system is already contained in the `/tmp/rootfs` file, so this can readily be substituted into the configuration file. For GRUB we will need to translate it to a GRUB partition name as well, which may be a bit tricky in the shell environment available on the boot disk,

- Running the LILO/GRUB installer. The normal LILO or GRUB binary cannot be run from the install system as it is linked against the wrong libraries. For now assume that we use a version of the GRUB installer that is linked against `uClibc`. This needs not be included in the RAM disk, it can be on the hard disk in a special directory, which may be removed later.

The GRUB that came on the diskette or CD-ROM that you started the installation with, is also capable of making the system on the hard disk bootable, but this is not something to expect from an end user.

In a real-world system the configuration script should accommodate at least three situations:

- Linux is alone on the system.

[12] But it can be booted using the GRUB fbppy or CD-ROM that you installed from, but this is not something for an end-user

- Linux must co-exist and be dual-bootable with an existing Windows installation.
- The system already has a boot manager of some sort (another GRUB, Windows NT boot manager) and the user will configure the boot manager to boot Linux as well.

In the former two cases, LILO or GRUB should be installed in the MBR of the hard disk, in the latter case, LILO or GRUB should be installed in the boot sector of a partition.

An alternative to making the system bootable from the hard disk is to create a custom boot diskette. In this case the configuration script only has to `dd` a pre-existing GRUB image to a formatted diskette and to write a custom `menu.lst` file to it. This saves the trouble of having to compile a custom `uClibc` version of GRUB and to keep existing boot managers intact. The problem is that in the near future, most new PCs won't have floppy drives, so this approach cannot be used.

11.4 Debugging Installation Scripts

There are at least three ways to debug the installation scripts:

- If the host system has the `dialog` utility, you can run the scripts on the host system. It is not wise to reformat random partitions though. Instead of doing a real `mke2fs` you could include an echo statement followed by `sleep 5`. This can at least be used to get most of the logic of your scripts right. Especially you can check that menus and other screens are displayed correctly.
- One step further is to `chroot` into the `rootfs` directory and to run the scripts from there. This way we can make sure that the shell, the `dialog` utility and the available shell commands are the same as on the target system.
- The next step is to create the RAM disk image that includes the scripts from the `/usr/scripts` directory and to copy it to a floppy or even a CD-ROM. Next we can boot on a real machine or a virtual PC product such as VMWARE or `bochs`. Preferably this is an old machine with a scratch hard disk that you can reformat as you like (or a hard disk image contained in a scratch file of a virtual PC product).

  You do not need to recreate the RAM disk image and reboot in order to fix every bug. Instead you can run a shell in a parallel virtual terminal. You can edit the scripts using the `vi` included in Busybox. You can kill the top

level install script to make it start all over. Mount a diskette and (regularly) copy the edited and debugged shell scripts to it. Warning: if you switch off the computer without saving the debugged shell scripts, you lose them of course. Later the debugged scripts can be copied from the diskette back to the `$MYBOOT/rootfs/usr/scripts` directory on the host system and the boot diskettes or CD-ROM can be rebuilt.

12 Conclusion

It is not only feasible to create your own Linux system from scratch, but also your custom installation diskettes. This way it becomes possible to make installation diskettes for machines that are no longer supported by modern Linux distributions. It is also possible to use this system to install special-purpose Linux distributions on a large number of machines in a company. If you wondered: that's the profit part of the title. If you managed to read this text so far, you obviously understand the fun part.

A few concluding remarks:

- Shell scripting and `dialog` may not be very user friendly, but it is possible to create a usable installer with them. Debian has done this for a while.
- I am apparently far removed from the professional level Linux installers. I've not yet developed scripts to select and install modules satisfactorily. My current diskettes have network support, but no practical way to load the necessary modules for the Ethernet card. Maybe I should borrow some Debian installation scripts.
- It is still possible to support 486 machines with 8kB of RAM, even with the latest kernel.
- With kernel 2.4 is not really possible to create single diskette installation disks if a reasonably featureful kernel is desired. Two diskette systems and 2.88MB diskette images (for a bootable CD) are still possible.
- This text is a general plan to create installation disks. Many features can be added or removed. Network support could be removed or on the other hand we could add a DHCP client and a small web browser. Support for reiserfs (or other advanced file systems) could be added (you need kernel support and the utilities to create the file systems). Maybe you want a different editor or even a Basic interpreter. It's all yours to decide.