Building A Custom Linux Kernel

Brian Mays

October 2000

Topics

- The Goal
- The Motivation
- The Cost
- Procedure
- Summary
- Building Debian Kernel Packages

The Goal

Present a procedure for building a custom kernel for an x86 computer using this laptop as a demonstration vehicle. Although, the laptop is running Debian GNU/Linux, a *generic* procedure—suitable for any distribution of Linux—will be presented first. The presentation will end with a demonstration of the the recommended Debian procedure, that uses tools specially designed for the Debian distribution.

The Motivation

What does the kernel do, anyway?

The Unix kernel acts as a mediator for your programs and your hardware.

Most importantly,

- it manages memory and allocates it to the running programs (processes);
- it ensures that they all get a fair share of the processor's cycles; and
- it provides an interface for programs to talk to the hardware.

Why would I want to upgrade my kernel?

Newer kernels generally . . .

- offer the ability to talk to more types of hardware (that is, they have more device drivers),
- often have better process management,
- can often run faster than the older versions,
- could be more stable than the older versions, and finally
- newer kernels fix silly bugs that appear in the older versions.

Why would I want to build a custom kernel?

Reduced RAM usage. The monolithic part of the kernel stays in RAM all the time. A custom built kernel is generally smaller than a generic one.

Reduced disk usage. A smaller kernel requires less disk space.

Improved Performance. Custom kernels can be optimized for newer, faster processors. Generic kernels are usually built to run on anything (i.e., 386).

Special Hardware. Some weird hardware do not work with the kernels that come with a distribution. Sound card support is common reason for building a custom kernel.

Problem Solving. Nothing is perfect. Sometimes the only way to get a particular service (say Appletalk) running is to compile a custom kernel.

Because you can. Seriously, the major advantage offered by an open source OS like Linux is that **you** make the important decisions about your software, not the vendor.

The Cost

How much disk space do I need?

It depends on your particular system configuration. The compressed Linux source is nearly 18 megabytes large at version 2.2.16. Uncompressed and built with a moderate configuration, the source takes up over 75 MB.

The size of the kernel source is growing constantly.

How long does it take?

With newer machines, the compilation takes dramatically less time than older ones; an AMD K6-2/300 with a fast disk can do a 2.2.x kernel in about four minutes. On old Pentiums, 486s, and 386s, be prepared to wait, possibly hours, days . . .

Procedure

Fetching the Sources

Sources are **available** via anonymous ftp from `ftp.kernel.org` in `/pub/linux/kernel/v`x`.`y

(Keep in mind that versions ending in an odd number are *development* releases and may be *unstable*.)

Typically, they are **labelled** `linux-`x`.`y`.`z`.tar.gz`.

(The sites often carry files with a suffix of `.bz2`, which have been compressed with *bzip2*, a tighter compression scheme.)

It's best to use `ftp.`cc`.kernel.org`, where cc is a two-letter country code. For example, `ftp.at.kernel.org` is for Austria, and `ftp.us.kernel.org` is for the United States.

Unpacking

Usually, the source is unpacked under the `/usr/src` directory. (Of course, this must be done as *root*.)

Unpack the source with

```
tar zxpf linux-x.y.z.tar.gz
```

When finished, there will be a new `linux` subdirectory containing the kernel source tree.

A file name `README` contains useful information, including instructions for *configuring*, *compiling*, and *installing* the kernel.

Configuring

Before it can be compiled, the kernel must be configured. Use the following procedure:

```
make mrproper
```

This insures that we are working with a pristine source tree.

```
make xconfig
```

This interactively configures the kernel. Note that `xconfig` uses an X (windows) interface, `menuconfig` uses a menu-based interface, and `config` is for the old-timers who are accustomed to answering “yes/no” questions.

The configuration is stored in a file called “`.config`”. Save this file if you want to use it later.

```
make dep
```

This sets up the dependencies. It tells the compiler what files to compile.

Build the Image

A compiled kernel is called an “image.” To build it, use

```
make zImage
```

(Use “`bzImage`” if the kernel is too large for `zImage`.)

The kernel can load additional “modules.” These must be built separately, using

```
make modules
```

and installed into `/lib/modules/`x`.`y`.`z with

```
make modules_install
```

Building modules for PCMCIA devices is another matter altogether. It will not be covered in this presentation.

Testing

A safe, fairly easy method of testing the new kernel is to write the kernel to a floppy and boot off of the floppy. (This laptop runs only linux—many systems dual-boot linux and other operating systems on the same machine.)

To test the new kernel from a floppy, put a floppy disk into the (first) drive and run the following:

```
make zdisk
```

This will write the kernel to the floppy.

Then reboot the computer from the floppy

```
/sbin/shutdown -r now
```

After you have rebooted, look for weird messages with "`dmesg | more`" and check for general weirdness. You might need to build the kernel more than once in order to figure out all the right configuration choices.

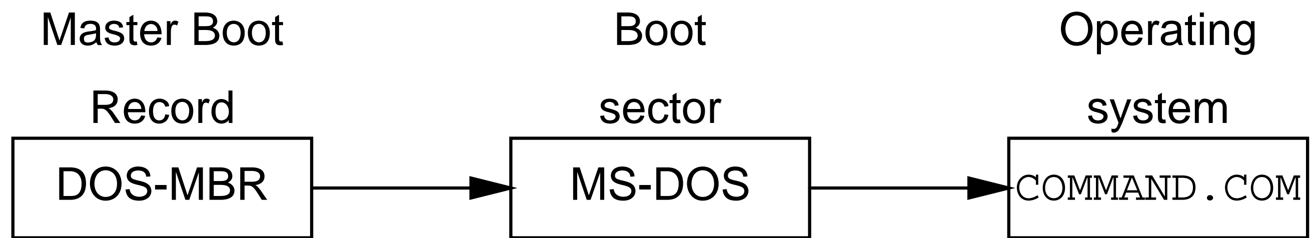
LILO

What is LILO?

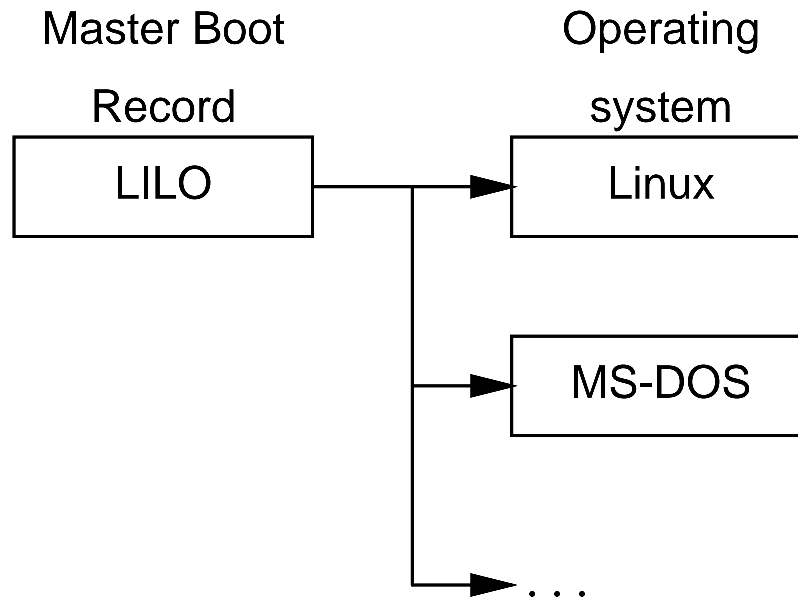
LILO is a versatile boot loader for Linux. It . . .

- does not depend on a specific file system,
- can boot Linux kernel images from floppy disks and from hard disks, and
- can even act as a “boot manager” for other operating systems.

DOS



LILO



Configuring LILO

The next step is to reconfigure `/etc/lilo.conf`. A typical configuration is

`boot=/dev/hda`	*boot device (MBR of first drive)*
`root=/dev/hda3`	*device that should be mounted as root*
`compact`	*enables map compaction to reduce load time and keeps the map smaller*
`install=/boot/boot.b`	*Install the specified file as the new boot sector (default is* `/boot/boot.b`*)*
`delay=40`	*delay (in 1/10 secs) before booting the first image*
`map=/boot/map`	*the location of the map file (default is* `/boot/map`*)*
`read-only`	*mount the root filesystem read-only*

`image=/vmlinuz`	*new kernel image*
`  label=linux`	*label*
`  alias=l`	*a short alias*
`image=/vmlinuz.old`	*old kernel image (backup)*
`  label=old`	*correctly labeled*
`other=/dev/hda1`	*the DOS/Windows partition*
`  label=dos`	*correctly labeled*
`  table=/dev/hda`	*device containing the partition table*

Finishing

The finial step is to run

```
make zlilo
```

or

```
make install
```

This will backup the old image (to `vmlinux.old`), place the new image in the correct location, and run `lilo` to install the boot loader.

Once again, reboot the computer using

```
/sbin/shutdown -r now
```

and check for general weirdness.

Summary

1. Fetch and unpack the source
2. Configure the kernel (`make xconfig`)
3. Build the image
4. Test (by booting from a floppy)
5. Configure LILO
6. Install the image and rerun LILO

Building Debian Kernel Packages

Debian makes it easy to build kernels. The `make-kpkg` utility found in Debian's `kernel-package` package allows the user to easily build Debian packages containing custom kernels.

This provides the following advantages:

- `make-kpkg` automates the kernel build procedure.
- The Debian packaging system (`dpkg`) catalogs and controls all files, such as the kernel image, system maps, etc.
- The maintainer scripts in the newly created package eliminate conflicts between kernel versions, and *automatically* and *transparently* handle switching to the new kernel.
- Using packages simplifies placing the same kernel on multiple systems.

Procedure

1. Install the `kernel-package` package.
2. Unpack the kernel source (can be found in the `kernel-source-`$x.y.z$ package).
3. Configure the kernel:

   ```
   make xconfig
   ```

4. Run `make-kpkg` with

   `make-kpkg --revision=`*number* `kernel_image`

   where *number* is your chosen revision number (e.g., `custom.1.0`).
5. Install the kernel image on one or more machines

   `dpkg -i ../kernel-image-`$x.y.z$_*number*_*arch*`.deb`

   (The package will offer to build a boot floppy and run LILO.)
6. Restart the computer:

   ```
   shutdown -r now
   ```

Fakeroot

Kernels can easily be built under your user's account (i.e., *not* as root) by using the `fakeroot` utility (found in the Debian package with the same name).

To accomplish this, set the `ROOT_CMD` environment variable:

```
export ROOT_CMD='fakeroot --'
```

before executing `make-kpkg`.

The End