Techniques for upgrading and customizing the Linux kernel

KERNEL TECH

If you work with third-party hardware drivers, or even if you just need to fix a broken system, someday you might need to upgrade the Linux kernel. **BY KLAUS KNOPPER**

A technical expert will tell you that the kernel is *the* Linux – the Hardware Abstraction Layer and everything else you see on your screen is mostly application software from the GNU collection. Linux is the operating system core that makes a computer usable in a Unix-like way. On the technical level, a kernel consists of the following basic components:

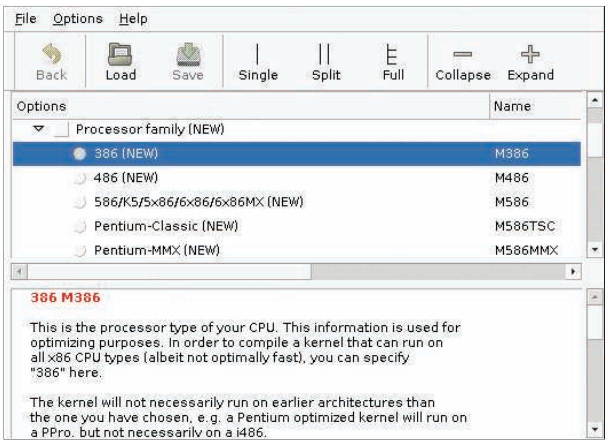
- support for hardware and corresponding drivers;
- a so-called *scheduler*, which distributes available computing power (CPU cycles) and hardware resources among application programs, thus allowing the programs to run independently of each other without causing deadlocks or conflicts;
- a virtual memory and filesystem manager that makes memory and disk space available to programs.

Most users just accept the kernel that comes with their Linux distro without seriously tinkering with it. However, if you happen to need a driver or system component that isn't built into your Linux system, or even if you just like tinkering, you might one day face the task of replacing, rebuilding, or extending the kernel running on your system. In this article, I describe some techniques for working with the Linux kernel.

Why Upgrade?

Kernel 2.6.28 is released, and when you read through the changelogs, you notice a vast number of bug fixes and feature enhancements that create an impression of huge performance and stability benefits with a simple package update. Unfortunately, the reality is not quite so simple. Because of the rapid kernel development process, new releases appear almost every month. Most new releases do not provide a major change unless you are looking for something very specific. And, even if a new component or subsystem is announced in the news, it is unlikely that you will find the latest kernel version ready to install in your distro's repository. Most distributions provide stable and well-tested snapshots that might contain some selected new features and improvements of newer releases but keep the old version number for compatibility with third-party modules. If you are looking for the newest (and

robert fori, Fotolia



Options	Name
Processor family (NEW)	
386 (NEW)	M386
486 (NEW)	M486
586/K5/5x86/6x86/6x86MX (NEW)	M586
Pentium-Classic (NEW)	M586TSC
Pentium-MMX (NEW)	M586MMX

386 M386

This is the processor type of your CPU. This information is used for optimizing purposes. In order to compile a kernel that can run on all x86 CPU types (albeit not optimally fast), you can specify "386" here.

The kernel will not necessarily run on earlier architectures than the one you have chosen, e.g. a Pentium optimized kernel will run on a PPro, but not necessarily on a i486.

Figure 1: Use make gconfig to compile the kernel in a Gtk-based environment.

probably not well-tested) version, you will have to compile the kernel yourself.

Why would anyone even bother to upgrade the Linux kernel? If you spend a lot of time hacking your Linux system, you might find yourself needing to repair a system that broke because you forgot to activate an important option. Or, in some cases, a new kernel might contain a driver or support a module that offers improved hardware support. In other situations, the update might address an urgent security problem.

Checking Kernel Version

To find out which kernel is currently running on your system, open a shell and type

```
uname -a
```

which should output something such as:

```
Linux eeepc 2.6.26.5-eeepc #13 PREEMPT Thu Oct 9 04:04:42 CEST 2008 i686 GNU/Linux
```

Another command

```
cat /proc/version
```

provides additional information on the compiler used to build the kernel.

As you will learn, this information comes in handy when you are working with kernel modules:

```
Linux version 2.6.26.5-eeepc (knopper@Koffer) (gcc version 4.3.2 (Debian 4.3.2-1) ) #13 PREEMPT Thu Oct 9 04:04:42 CEST 2008
```

The preceding output reveals several facts about the system:

- The kernel is in the 2.6 kernel series.
- The minor release version is 26.
- The patch level (mostly for bug fixes) is 5.
- It was presumably compiled for an Eee PC system, although this setting can be any text string specified as *EXTRAVERSION* in the kernel Makefile.
- The system architecture is based on i686, which supports the machine instruction sets of Pentium 2 and higher, but not older 386- or 486-based computers.
- The GCC compiler version used when compiling the kernel from its source was version 4.3.2 on a Debian system. The resulting binary was the 13th time the compiler was run for this source, and the compile was performed on Thursday October 9, 2008.
- This kernel is preemptible. That is, the system is optimized as a desktop system with a quick interactive response rather than as a compute server.

This detailed information on the state of the current system provides a starting point for understanding how to proceed with a kernel upgrade.

Package Upgrade

Most Linux distributions provide an easy means for upgrading the kernel through a package management system. If you don't need a customized or optimized kernel, updating the kernel through your distro's package system is often much easier than compiling and manually installing the kernel on your own.

Here, I describe how to upgrade your kernel through the Debian-based Aptitude package management system. The concepts are similar for other systems. If your distro uses an alternative package tool, consult the vendor documentation.

The Debian kernel packages used to have the name *kernel-image*. This name has recently changed to *linux-image*. Command-line gourmets will prefer to install the new kernel image with a text-based command

```
aptitude install linux-image-686
```

instead of with a GUI-based package manager; in either case, the steps executed are basically the same:

1. The package is decompressed and unpacked into a new location. The static part of the kernel goes to */boot/vlinuz-versionnumber-architecture*; kernel modules go to */lib/modules/versionnumber*.
2. Scripts check to see whether an initial ramdisk is necessary for your system; if it is, the necessary modules are set up in a file called */boot/initrd.img-versionnumber-architecture*. The system tool *mkinitramfs* is responsible for this step. Its configuration files are at */etc/initramfs-tools/**, which is where you will go to make certain configuration changes, such as changing the ramdisk configuration. Unless the module names have changed, or unless you plan to activate software RAID or LVM, you should not have to do much there on your own.
3. The bootloader is made aware of the new kernel as a choice for booting. Unless the old kernel is removed (which should not happen automatically), it will still be in the bootloader configuration file, which allows you to switch back to the old kernel interactively as the system starts.

Before you reboot the system, check the following:

Listing 1: /etc/lilo.conf

```
image=/vmlinuz
        initrd=/initrd.img
        label=Linux
image=/vmlinuz.old
        initrd=/initrd.img.old
        label=Linux -old
```

Kernel Options

The kernel has some *interactive* qualities – you can boot the system with the use of kernel options that affect the way some part of the kernel works and even change certain kernel settings during run time without rebooting. Technically oriented users often enjoy just browsing through the new kernel's options to look for new settings to play with.

- Did you previously have to install or recompile additional kernel modules for the system to start? In the rare case that your primary boot medium controller needs a driver that is not part of the kernel or ramdisk, you will have to compile and install the necessary module before you reboot; otherwise, it might be difficult to get the system up and running with the new kernel.
- Is the bootloader correctly prepared for the new kernel? For instance, LILO (the Linux Loader – one of the first filesystem-independent bootloaders for Linux) should have entries similar to Listing 1 in the */etc/lilo.conf* file.

If you are using the GRUB bootloader, the */boot/grub/menu.lst* file will need entries similar to those in Listing 2.

In Listing 1, note that */vmlinuz.old* and */initrd.img.old* are symbolic links to the old but still-existing kernel and initrd files in */boot*. This approach makes it possible to boot the old kernel if the new one isn't working as expected. If you change */etc/lilo.conf* manually, run the *lilo* command as root before rebooting, because the LILO bootloader needs to update its record of the kernel file location. The GRUB bootloader, on the other hand, can find the files on its own using its own filesystem driver.

If it seems that your system is ready for the new kernel, reboot to see that everything works. If the new kernel doesn't start for some reason, select the old kernel in the boot menu to restore the previous configuration.

Compiling and Customizing the Kernel

If you feel like tuning up your kernel for a specific situation, or if you are looking for features that aren't present in your distribution's kernel default kernel build, you can always try your luck compiling the kernel yourself. Start by installing the C compiler and assembler (the gcc and the binutils packages). On Debian, for example, enter

```
sudo aptitude install ⮐
binutils gcc make
```

then fetch and unpack the kernel source from kernel.org [1] or one of its mirrors. An alternative to installing the latest version is to obtain the last major release and apply any subsequent patches:

```
wget -c http://www.kernel.org/⮐
pub/linux/kernel/v2.6/⮐
linux-2.6.28.tar.bz2
tar jxvf linux-2.6.28.tar.bz2
```

The next steps depend on whether you want to change something in your old kernel configuration or keep everything as is and just do the upgrade.

After you have unpacked the new kernel source, it is much easier to copy your old kernel setup to the new directory first if you don't plan on making lots of changes. This strategy saves you having to go through all the hundreds of options one by one and guessing which setting will match your system.

The entire collection of kernel options and settings is stored in a file called *.config* (note the dot at the beginning; the file is *hidden*, kind of) inside the kernel source directory, *which is linux-2.6.28/.config* in this example.

In Debian, you can find a copy of the *.config* file for your current kernel in the same directory in which the binary kernel is installed (*/boot/config-kernelversion*). For other distributions, you might have to look inside the source package matching the installed binary kernel package.

After copying the old kernel configuration to the new source directory, change into the new kernel source directory,

```
cd linux-2.6.28
```

start the kernel configuration, and browse through all the available options.

Depending on whether you have installed the Qt3 or Gtk2 toolkit development environment, you can compile and start a graphical kernel configuration front end with

```
make xconfig
```

for a Qt-based environment or

```
make gconfig
```

for a Gtk-based environment (Figure 1).

If neither of these commands work because development files are missing, the text-based alternative requires only the *ncurses* libraries:

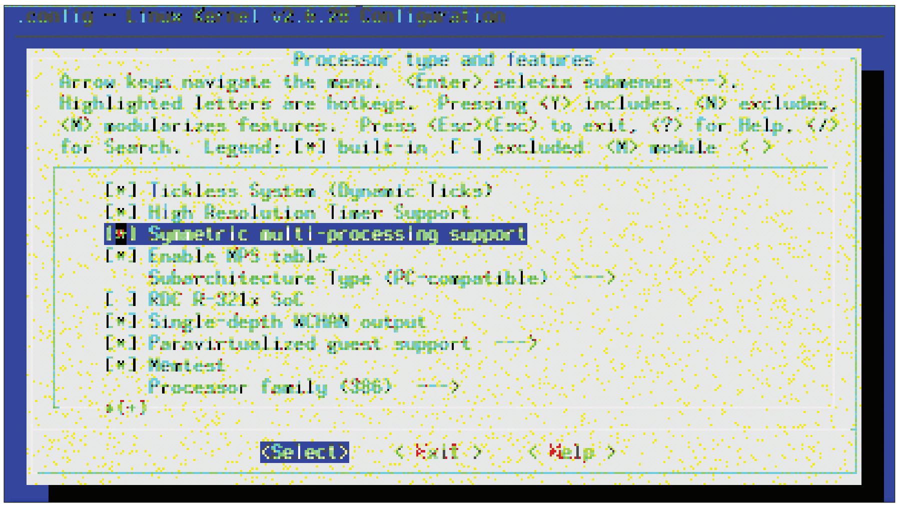
```
make menuconfig
```

This command was used to create the screen shot in Figure 2.

As a last resort,

```
make config
```

will always work, but it requires you to acknowledge each and every kernel



```
.config - Linux Kernel v2.6.28 Configuration
                         Processor type and features
  Arrow keys navigate the menu.  <Enter> selects submenus --->.
  Highlighted letters are hotkeys.  Pressing <Y> includes, <N> excludes,
  <M> modularizes features.  Press <Esc><Esc> to exit, <?> for Help, </>
  for Search.  Legend: [*] built-in  [ ] excluded  <M> module  < >

      [*] Tickless System (Dynamic Ticks)
      [*] High Resolution Timer Support
      [*] Symmetric multi-processing support
      [*] Enable MPS table
          Subarchitecture Type (PC-compatible)  --->
      [ ] RDC R-321x SoC
      [*] Single-depth WCHAN output
      [*] Paravirtualized guest support  --->
      [*] Memtest
          Processor family (386)  --->
      v(+)

                  <Select>    < Exit >    < Help >
```

Figure 2: For text-based environments, try make menuconfig.

Listing 2: /boot/grub/menu.lst

```
title=Linux
root (hd0,0)
kernel /vmlinuz
root=/dev/hda1
initrd /initrd.img

title=Linux-old
root (hd0,0)
kernel /vmlinuz.old root=/dev/hda1
initrd /initrd.img.old
```

option one after the other, so this command is therefore quite tiresome.

Some options are just on/off (like certain features that affect the static part of the kernel); other features allow the user to compile a driver into the kernel or as a module that can be loaded from disk after the initial filesystem is activated.

To compile a kernel that is optimized for your system, you should investigate your hardware. My recommendation is that you compile the hard disk controller responsible for the boot disk, as well as the general disk driver (IDE/SATA) into the kernel.

To find out which kernel driver is right for your hardware, try

```
lspci -vmm -k
```

which will also show you the name of the kernel component or module that matches a specific chipset.

Usually, it does not hurt to compile driver modules for hardware you don't have (yet). These drivers will just be ignored until new hardware is detected and Udev, the automatic on-demand hardware detection system, loads them. Just watch out for mutually exclusive drivers. The USB system, for instance, supports a number of alternative drivers that aren't always interchangeable. The low-performance USB block driver (ub) is known to kill performance and the stability of fast USB storage devices that would otherwise run perfectly with the alternative usb-storage driver.

During kernel configuration, you will find a number of options that seem important but are not really self-explanatory. The built-in configuration help file gives a brief overview (which is not always helpful); you'll find more documentation inside the kernel source *Documentation* directory – the file called *kernel-parameters.txt* is especially worth reading. The safest approach, in any case, is just to keep the default, which is the option that works for most hardware configurations.

After you are done configuring kernel options, leave the configuration GUI with *Save Changes*.

Now you can start the compiler with a simple

```
make
```

which can take some time to complete. If you rerun this procedure, it is a good idea to remove old binaries with *make clean* before restarting the process.

For some of the more experimental kernel modules, compilation can fail with certain kernel and compiler versions. Unless you are familiar with the C language and feel ready to change the source code directly, it is easiest to just deactivate the offending driver.

After a successful compilation, you can install the kernel:

- manually, by typing *sudo make install*, which copies *arch/i386/boot/bzImage* to */boot/vmlinuz-** and all kernel modules to */lib/modules/versionnumber/*. To make the bootloader aware of a new kernel boot option, you still have to configure *lilo.conf* (for LILO) or *menu.lst* (for GRUB) manually.
- by creating a package for your distribution and installing that package. The package manager should take care of doing all necessary bootloader modifications and, if necessary, creating an initial ramdisk file.

For Debian, the package helper for creating kernel packages is *make-kpkg*, which you can invoke inside the kernel source directory:

```
make-kpkg --us --uc ↩
--rootcmd fakeroot ↩
kernel_image
```

Then, install the resulting kernel package:

```
sudo dpkg -i ../↩
linux-image-kernelversion*.deb
```

For RPM-based distributions, you will have to look into the old kernel source package's *.spec* file, modify it for the new kernel source version, and run *rpm -ba specfile* to start the compile and package creation.

Keep in mind that you will have to (re-)compile all additional modules that are not part of the original kernel source. (Read on for more about working with Linux kernel modules.)

Working with Kernel Modules

Before you throw away your old Linux kernel and upgrade the whole base system, keep in mind that Linux offers a less radical solution for integrating new drivers and features. Loadable Kernel Modules (LKM) are bits of executable code that are not part of the static (base)

Hard Disk Drivers

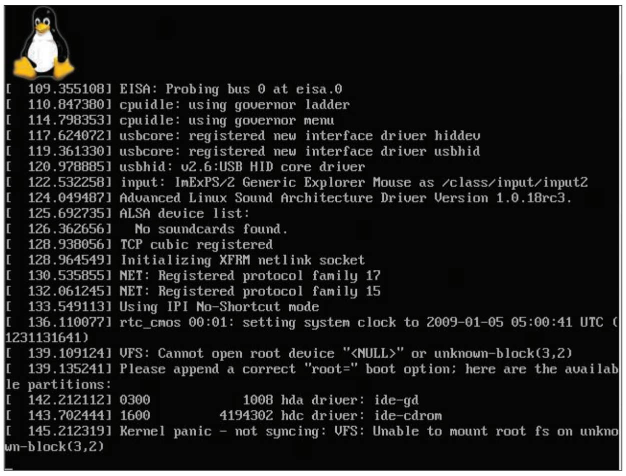
For some hardware, two alternative drivers might both work fine, but you still have to choose one of them. IDE hard disk controllers, for instance, work with both the traditional IDE block device drivers and the newer PATA interface, which is connected to SATA. For controllers that have both SATA and IDE ports, the SATA/PATA combination is most likely the best choice. Enabling both the IDE and SATA/PATA driver at the same time for the same controller can sometimes work: As soon as interrupt and I/O resources are blocked by one driver, the other driver silently fails. Sometimes it does not go so well, and each driver block parts of the other, so direct memory access (DMA) becomes unavailable, hard disks slow down or disappear, or timeouts and resets occur.

If you decide to use the SATA/PATA driver for a hard disk controller instead of the IDE driver you used before, make sure to change */dev/hda* to */dev/sda* in */etc/fstab* because PATA treats IDE hard disks like SCSI disks. The same precaution applies to the *root=/dev/hda1* lines in the *lilo.conf* or *menu.lst* bootloader files.

No Initial Ramdisk?

Most distributions compile only a minimum subset of drivers directly into the static kernel then install all available hardware drivers as modules into the root filesystem. The drivers necessary for mounting the root filesystem are stored inside the initial ramdisk. I personally prefer going without an initial ramdisk for hard disk installations and then compiling the drivers necessary for hard disk access directly into the kernel. The same applies for USB drivers that *might* be needed at a very early stage of the boot process (e.g., USB keyboards and USB storage). If the root filesystem has been partly damaged and you can't load any more drivers from the filesystem, you might still be able to mount additional media from an emergency shell and do system recovery. Also, the boot process is somewhat simpler without the intermediate initial ramdisk step, but that, again, is just a matter of personal preference.

Anzeige
wird
separat
angeliefert



```
[  109.355108] EISA: Probing bus 0 at eisa.0
[  110.847380] cpuidle: using governor ladder
[  114.798353] cpuidle: using governor menu
[  117.624072] usbcore: registered new interface driver hiddev
[  119.361330] usbcore: registered new interface driver usbhid
[  120.978885] usbhid: v2.6:USB HID core driver
[  122.532258] input: ImExPS/2 Generic Explorer Mouse as /class/input/input2
[  124.049487] Advanced Linux Sound Architecture Driver Version 1.0.18rc3.
[  125.692735] ALSA device list:
[  126.362656]   No soundcards found.
[  128.938056] TCP cubic registered
[  128.964549] Initializing XFRM netlink socket
[  130.535855] NET: Registered protocol family 17
[  132.061245] NET: Registered protocol family 15
[  133.549113] Using IPI No-Shortcut mode
[  136.110077] rtc_cmos 00:01: setting system clock to 2009-01-05 05:00:41 UTC (
1231131641)
[  139.109124] VFS: Cannot open root device "<NULL>" or unknown-block(3,2)
[  139.135241] Please append a correct "root=" boot option; here are the availab
le partitions:
[  142.212112] 0300             1008 hda driver: ide-gd
[  143.702444] 1600          4194302 hdc driver: ide-cdrom
[  145.212319] Kernel panic - not syncing: VFS: Unable to mount root fs on unkno
wn-block(3,2)
```

Figure 3: If your kernel panics, try to determine the point in the startup process when the error occurred.

kernel but are, instead, loaded separately at a later stage of the startup process. Device drivers, file system drivers, and other custom extensions are often implemented as kernel modules. Keeping the code in the form of a separate module eliminates the need for a full system upgrade just to add a single component.

The kernel provides hundreds of drivers for different hardware, but sometimes, especially with very new notebooks, some drivers (such as WLAN, LAN, and camera drivers) are only available in independent projects that have not managed to get their drivers accepted into the mainstream kernel yet. In other cases, the license might not support integration of the code into the base kernel, or the code was not tested well enough to fit the quality standards of the core kernel development team. In these situations, you might need to obtain the code for the kernel module and build it yourself.

Advanced modules in the form of source archives can be found at sites such as SourceForge [2]. MadWifi [3] (for some popular new WiFi chipsets) and GSPCA [4] (for webcams) are prominent examples of kernel modules available online. Unfortunately, it is sometimes difficult to compile module source code with the newest kernels because changes in the kernel API can cause compilation errors.

Before compiling additional modules, be aware that, for this task, you need the exact kernel source that was used for building the binary kernel that will accept the new module during run time, as well as the same GCC compiler that was used to build that kernel. Under certain circumstances, it is also possible to load modules compiled for a (slightly) different kernel with *insmod -f*, but this approach has the potential to make your system unstable because certain hardware-specific machine instructions and symbols inside the kernel won't match. If you installed your kernel from your favorite distribution's installation resource (DVD or Internet repositories), chances are good that you will find the corresponding source there.

The kernel 2.6 Makefile system provides an easy way to find the right options to compile additional modules that work with the kernel – which saves module developers some work.

As an example of how to compile and install a kernel module, I will use cloop, the compressed loopback device, which I frequently have to recompile for Live CD systems when upgrading the kernel.

The Cloop source code is available online [5]. To unpack the tarball, use:

```
tar zxvf  cloop_2.628-2.tar.gz
```

After changing into the *cloop-2.628* directory, compile the module with:

```
make obj-m=cloop.o cloop-objs=↩
compressed_loop.o -C ↩
```

Fitting the Hardware

If you want to compile a kernel that runs on a variety of different boards and processors (or at least *86-compatible variants), read the processor-specific option help file carefully and opt for generic optimizations and conservative settings rather than speed and processor-specific features. A kernel compiled for 80386 processors will run on any recent Pentium or AMD processor; a kernel compiled for newer processors will not work on earlier processor types. The performance advantage of a processor-specific kernel is rather low (around 5–8%) because desktop programs usually make comparably fewer calls to the processor's extended features, unless you are playing a fast game with quick calculations and high throughput. Even compiling the kernel for native 64-bit processors might not be advisable if you plan to run 32-bit applications. Most 64-bit CPUs can run 32-bit applications, but not vice versa.

The maximum supported memory size can be a problem: Processors with Physical Address Extension (PAE) support can use up to 64GB of RAM, but a kernel compiled with PAE will crash immediately on processors that don't support it. The safe option is the 4GB limit, which works for most 32-bit processors, of which only about 3GB is usable RAM and the rest is for internal addressing. On machines that will never have more than 1GB of RAM, the *no high memory support* option enables the fastest memory address scheme.

Options that improve performance and make the kernel more flexible are all located in the *processor type and features* section. Here, you can safely select *Symmetric Multiprocessing* (but not necessarily the SMP/hyperthreading-optimized schedulers), *Preemptible Kernel (Low-Latency Desktop)*, and *Generic x86 Support* (optimizations for an entire processor family). For all other options, read the help file before making a change. Some options are harmless and improve system performance under certain circumstances, whereas others limit the range of processors on which the kernel will work.

Enabling Symmetric Multiprocessing (SMP) usually does not hurt, even for old processors that definitely do not support it. The kernel checks to see whether or not the processor can use SMP (or hyperthreading); if not, single-processor procedures are used. Enabling SMP for non-SMP systems makes the kernel slightly larger, but you won't notice a difference in performance speed unless you run a very old or slow computer. Some boards, however, incorrectly report having a second processor when, in fact, only one is installed, creating an SMP-enabled kernel crash. For these situations, the kernel boot option *nosmp* or *maxcpus=0* can force single-CPU mode. For third-party, binary-only kernel modules (that probably will have to be loaded with *insmod -f*), it might be necessary to run a non-SMP kernel because of an incompatible instruction API in those modules.

```
/mnt/knoppix.build/↩
Microknoppix/Kernel/↩
linux-2.6.28 M=`pwd`
```

This procedure is quite generic and should work with most module sources. A Makefile must be present in the module source directory, but the file can be empty if *obj-m* and *modulename-objs* are set as variables on the *make* command line. The *obj-m = cloop.o* statement tells the kernel Makefile that the module's main object is called *cloop.o*, and *cloop-objs = compressed_loop.c* says to compile the C source file *compressed_loop.c* as (only) a component of *cloop.(k)o*. Everything else is handled by the kernel Makefile, located inside the directory */mnt/knoppix.build/Microknoppix/Kernel/linux-2.6.28*, which was given on the *make* command line along with the *-C* option. The compilation process is shown in Listing 3.

Afterwards, the module *cloop.ko*, which is ready to be loaded by *insmod*, is present in the current directory. Some modules come with their own Makefile, which you should try first, but almost certainly, you will have to specify the kernel sources location somewhere before compiling. If no symlink */usr/src/linux* that points to that directory exists, the command

```
sudo ln -snf ↩
/path/to/kernel/source ↩
/usr/src/linux
```

is sometimes helpful if you are tired of searching for a way to tell a module's Makefile where to look for the kernel source.

If a module source directory is placed inside another directory called *modules* one directory above the kernel-source, *make-kpkg* will try to compile the module automatically after the kernel and create a Debian package from it.

Add-on modules should be installed in the module tree */lib/modules/kernelversion/* (sometimes a subdirectory called *extra* is used) and prepared for automatic dependency loading by calling:

```
depmod -ae
```

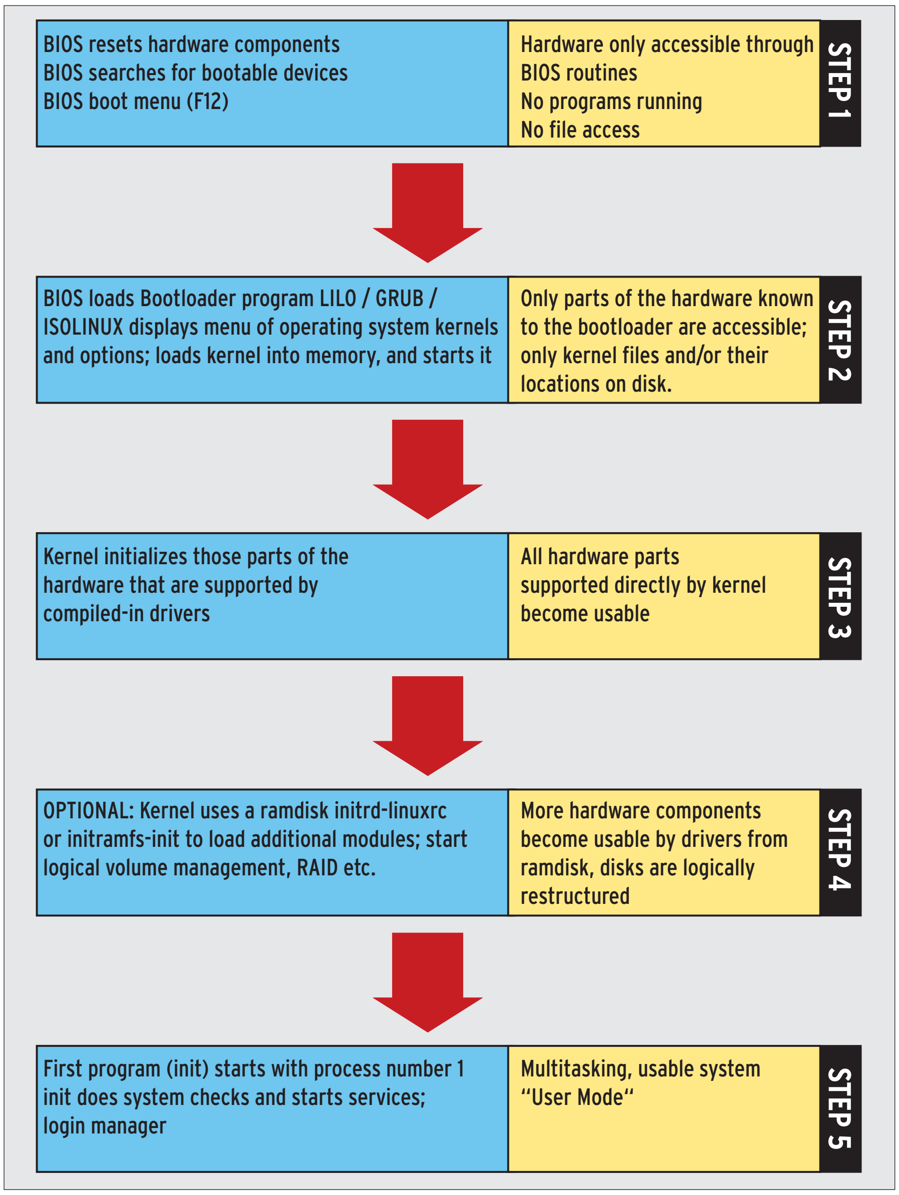
If the current kernel is not the same version as the kernel you want to use the module with, add the kernel version number as a last command argument. Now you should be able to load the module with the *modprobe modulename* command.

Watch *dmesg* for any signs of errors after module loading. If the module version does not match the kernel in use, you will see the exact error message there, rather than on the shell where you started *insmod* or *modprobe*. The message *invalid module format -- symbol versions mismatch* indicates that the module was not compiled with kernel source matching the currently running kernel.

I Just Killed My System!

Occasionally a kernel update seems successful, yet the system won't boot afterwards. Don't panic (even if your kernel just did). Figure 3 shows an example of the output that might appear if your system doesn't start. Before delving into the details of what to do in this situation, it is a good idea to review the way a typical *86-based PC starts up. Before all the multitasking begins, the system navigates a very linear procedure. Figure 4 shows the five major steps your computer goes through after you switch on the power. (Step 4 is optional, but most distributions use it.) The early part of the process is operating system–independent. OS-specific procedures don't start until step 3. If something goes wrong and the system doesn't start, identifying the place in the process where the failure occurred is the first step in uncovering the source of the problem.

If the BIOS is unable to identify a bootable device, the message will say some-


BIOS resets hardware components
BIOS searches for bootable devices
BIOS boot menu (F12)
Hardware only accessible through BIOS routines
No programs running
No file access
STEP 1
BIOS loads Bootloader program LILO / GRUB / ISOLINUX displays menu of operating system kernels and options; loads kernel into memory, and starts it
Only parts of the hardware known to the bootloader are accessible; only kernel files and/or their locations on disk.
STEP 2
Kernel initializes those parts of the hardware that are supported by compiled-in drivers
All hardware parts supported directly by kernel become usable
STEP 3
OPTIONAL: Kernel uses a ramdisk initrd-linuxrc or initramfs-init to load additional modules; start logical volume management, RAID etc.
More hardware components become usable by drivers from ramdisk, disks are logically restructured
STEP 4
First program (init) starts with process number 1 init does system checks and starts services; login manager
Multitasking, usable system "User Mode"
STEP 5
Figure 4: Understanding the Linux boot process.

thing like *no bootable harddisk found, hit return to continue*. Step 2 failures usually end with a bootloader message that says it cannot load the kernel file from hard disk, which means you mistyped the file name in the configuration, or you forgot to run LILO after changing *lilo.conf*, or GRUB does not have the necessary filesystem plugins available to find the kernel file on disk. Maybe the file name is too complicated for the simple GRUB filesystem implementation, or again, maybe you mistyped the name or entered the wrong hard disk in *menu.lst*.

In Figure 3, step 3 also was apparently OK because no fatal error message or freeze occurred during the first hardware initialization by the kernel. Because the output doesn't display an *unable to load ramdisk* message, you might think that step 4 cannot possibly have gone wrong, but it's still possible that the ramdisk loaded by the bootloader into memory was overwritten when the kernel image was decompressed into memory. Typically, this problem occurs when the static kernel gets too large to fit into memory before the start of the ramdisk location (a fixed address for most bootloaders), which is the case when the compressed kernel image exceeds approximately 2.5MB in size. In this case, I did not even use a ramdisk; instead, all drivers necessary to mount the root filesystem are compiled into the kernel.

The boot went fine until step 5, which is when the kernel should mount the root filesystem and give control to the first program, *init*. Possible reasons for a problem at this stage might be:

- The filesystem type needed for accessing the root partition was not compiled into the kernel, and it is not present as a module inside a ramdisk.
- The controller driver for the hard disk is not present (which is not the case in Figure 3).
- The wrong root partition was given as a boot argument to the kernel, either by the bootloader or as a boot command-line option.
- The hard disk is really broken (or wrongly configured in the BIOS).

Other causes also could have played a role in the failure, but the preceding alternatives are the most common. If driver support is missing, either for the hard disk, the controller, or the filesystem, kernel reconfiguration and recompilation is necessary, which means you have to reactivate your old kernel first. If the old kernel is no longer present or not working, try a Live system from USB flash or CD/DVD. From a root shell, mount the root partition

```
mount -o dev /dev/sda1 ⮐
/media/sda1
```

and do a

```
chroot /media/sda1
```

to access the root filesystem as you would have if the system were able to boot up directly. From there, you can mount all partitions

```
mount -a
```

and eventually compile a new kernel, fix the bootloader, and retry. Likely you don't want to recompile as root, so just switch to normal user mode with *su - username*. Please don't forget to remount all mounted partitions – at least read only, if not unmounting – to force-write changed data to disk:

```
umount -arvf
```

Conclusion

Installing a new kernel is not like installing a new version of OpenOffice, which will definitely add new features and enhancements to your everyday work. New major releases or experimental kernels often run slower and less smoothly because of side effects that have not been considered by the kernel developers. Unlike application software, a new kernel does not necessarily provide better service or more possibilities. If your current kernel is stable and runs smoothly and you have no sudden system resets, freezes, or "hangs," you should have no reason to believe that a new kernel will be an enormous improvement.

Unless you experience errors that are harmful in your usage scenario, just keep your old kernel and don't worry about being up to date. The primary reasons for upgrading the kernel are to correct a problem you are experiencing or to add new hardware that is not supported by the current kernel.

If you work with a variety of different hardware drivers, or even if you have the need to customize your system for a particular application or use, the techniques described in this article will help you get started with building and upgrading the Linux kernel. ■

Booting from a Running Kernel

In some situations, another interesting option is to boot a new kernel directly from a running Linux system. This option only works if the running kernel supports the *kexec* system call and the *kexec* utilities are installed.

```
kexec --initrd=⮐
/boot/initrd.img-2.6.28 ⮐
-append="root=/dev/sda1" ⮐
-l /boot/vmlinuz-2.6.28
kexec -e
```

This technique skips steps 1 (BIOS) and 2 (bootloader), loading and starting the new kernel (and the initial ramdisk) directly.

Listing 3: Output

```
make: Entering directory `/mnt/knoppix.build/Microknoppix/Kernel/linux-2.6.28'
  LD      /mnt/knoppix.build/Microknoppix/Kernel/cloop-2.628/built-in.o
  CC [M]  /mnt/knoppix.build/Microknoppix/Kernel/cloop-2.628/compressed_loop.o
  LD [M]  /mnt/knoppix.build/Microknoppix/Kernel/cloop-2.628/cloop.o
  Building modules, stage 2.
  MODPOST 1 modules
  CC      /mnt/knoppix.build/Microknoppix/Kernel/cloop-2.628/cloop.mod.o
  LD [M]  /mnt/knoppix.build/Microknoppix/Kernel/cloop-2.628/cloop.ko
make: Leaving directory `/mnt/knoppix.build/Microknoppix/Kernel/linux-2.6.28'
```

INFO

[1] Kernel.org: *http://www.kernel.org*
[2] SourceForge: *http://sourceforge.net/*
[3] MadWifi: *http://madwifi-project.org/wiki/About/MadWifi*
[4] GSPCA: *http://mxhaard.free.fr/*
[5] Cloop source code: *http://debian-knoppix.alioth.debian.org/sources/*