Linux kerneld mini−HOWTO

Table of Contents

Linux kerneld mini−HOWTO

Henrik Storner

kerneld−howto@linuxdoc.org

Revision History

Revision v2.0	22 May 2000
conversion from HTML to DocBook SGML.	

Table of Contents

About the kerneld mini–HOWTO

This document explains how to install and use the automatic kernel module loader "kerneld". The latest released version of this document can be found at the Linux Documentation Project

Credits

This document is based on an original HTML version 1.7 dated July 19, 1997 by Henrik Storner <storner@osiris.ping.dk> and was revised and translated to DocBook DTD by Gary Lawrence Murphy <garym@teledyn.com> May 20, 2000.

The following people have contributed to this mini–HOWTO at some point:

- Bjorn Ekwall bj0rn@blox.se
- Ben Galliart bgallia@luc.edu
- Cedric Tefft cedric@earthling.net
- Brian Miller bmiller@netspace.net.au
- James C. Tsiao jtsiao@madoka.jpl.nasa.gov

If you find errors in this document, please send email to <kerneld-howto@linuxdoc.org>. Your comments, encouragement and suggestions are welcome and appreciated, and help ensure this guide remains current and accurate.

What is kerneld?

The kerneld feature was introduced during the 1.3 development kernels by Bjorn Ekwall. It allows kernel modules such as device drivers, network drivers and filesystems to be loaded automatically when they are needed, rather than having to do it manually with **modprobe** or **insmod**.

And for the more amusing aspects, although these are not (yet ?) integrated with the standard kernel:

- It can be setup to run a user−program instead of the default screen blanker, thus letting you use any program as a screen−saver.
- Similar to the screen−blanker support, you can also change the standard console beep into something completely different.

kerneld consists of two components:

- Support in the Linux kernel for sending requests to a daemon requesting a module for a certain task.
- A user−space daemon that can figure out what modules must be loaded to fulfill the request from the kernel.

Both components must be working for the kerneld support to function; it is not enough that only one or the other has been setup.

Why do I want to use it ?

There are some good reasons for using kerneld. The ones I will mention are mine, others have other reasons.

- If you have to build kernels for several systems that only differ slightly − different kind of network card, for instance − then you can build a single kernel and some modules, instead of having to build individual kernels for each system.
- Modules are easier for developers to test. You don't need to reboot the system to load and unload the driver; this applies to all modules, not just kerneld−loaded ones.
- It cuts down on the kernel memory usage leaving more memory available for applications. Memory used by the kernel is *never* swapped out, so if you have 100Kb worth of unused drivers compiled into your kernel, they are simply wasting RAM.
- Some of the things I use, the ftape floppy−tape driver, for instance, or iBCS, are only available as modules, but I don't want to bother with loading and unloading them whenever I need them.
-

People making Linux distributions don't have to build 284 different boot images: Each user loads the drivers he needs for just his hardware. Most modern Linux distributions will detect your hardware and will only load those modules actually required.

Of course, there are also reasons why you may not want to use it. If you prefer to have just one kernel image file with all of your drivers built in, you are reading the wrong document.

Where can I pick up the necessary pieces ?

The support in the Linux kernel was introduced with Linux 1.3.57. If you have an earlier kernel version, you will need to upgrade if you want the kerneld support. The current Linux kernel sources can be found at most Linux FTP archive sites including:

- Kernel.Org Archive
- Metalab Linux Archive
- TSX−11 at MIT

The user−space daemon is included with the modules package. These are normally available from the same place as the kernel sources

> **Note:** If you want to try module−loading with the latest *development* kernels, you should use the newer modutils package and not the modules. Always check the `Documentation/Changes` file in the kernel sources for the minimum required version number for your kernel image. Also see about the problems with modules and 2.1 kernels.

How do I set it up?

First get the necessary parts: A suitable kernel and the latest modules package. Then you should install the module utilities as per the instructions included in the package. Pretty simple: Just unpack the sources and run **make install**. This compiles and installs the following programs in `/sbin`: **genksysm**, **insmod**, **lsmod**, **modprobe**, **depmod** and **kerneld**. I recommend you add some lines to your startup−scripts to do some necessary setup whenever you boot Linux. Add the following lines to your `/etc/rc.d/rc.S` file (if you are running Slackware), or to `/etc/rc.d/rc.sysinit` if you are running SysVinit, i.e. Debian, Corel, RedHat, Mandrake or Caldera:

```
        # Start kerneld - this should happen very early in the
        # boot process, certainly BEFORE you run fsck on filesystems
        # that might need to have disk drivers autoloaded
        if [ -x /sbin/kerneld ]
        then
                /sbin/kerneld
        fi

        # Your standard fsck commands go here
        # And you mount command to mount the root fs read-write

        # Update kernel-module dependencies file
        # Your root-fs MUST be mounted read-write by now
        if [ -x /sbin/depmod ]
        then
                /sbin/depmod -a
        fi
```

These commands may already be installed in your SysV init scripts. The first part starts kerneld itself. The second calls **depmod −a** at startup to build a list of all available modules and analyzes their inter−dependencies. The depmod map then tells kerneld if one module needs to have another loaded before it will itself load.

> **Note:** Recent versions of kerneld have an option to link with the GNU gdbm library, libgdbm. If you enable this when building the module utilities, *kerneld will not start if libgdbm is not available* which may well be the case if you have `/usr` on a separate partition and start kerneld before `/usr` is mounted. The recommended solution is to move `/usr/lib/libgdbm` to `/lib`, or to link kerneld statically.

Next, unpack the kernel sources, configure and build a kernel to your liking. If you have never done this before, you should definitely read the README file at the top level of the Linux sources. When you run *make xconfig* to configure the kernel, you should pay attention to some questions that appear early on:

```
  Enable loadable module support (CONFIG_MODULES) [Y/n/?] Y
```

You need to select the loadable module support, or there will be no modules for kerneld to load! Just say Yes.

```
  Kernel daemon support (CONFIG_KERNELD) [Y/n/?] Y
```

This, of course, is also necessary. Then, a lot of the things in the kernel can be built as modules − you will see questions like

```
  Normal floppy disk support (CONFIG_BLK_DEV_FD) [M/n/y/?]
```

where you can answer with an M for "Module". Generally, only the drivers necessary for you to boot up your system should be built into the kernel; the rest can be built as modules.

Essential drivers
Essential drivers required to boot your system must be compiled into the core kernel and cannot be loaded as modules. Typically this will include the hard−disk driver and the driver for the root filesystem. If you have a dual−boot machine and rely on files found in the foreign partition, you must also compile support for that filesystem into the core kernel.

When you have gone through the **make config**, compile and install the new kernel and the modules with **make dep clean bzlilo modules modules_install**.

Phew.

> **Compiling a Kernel Image:** The **make zImage** command will stop short of installing a kernel and will leave the new kernel image in the file `arch/i386/boot/zImage`. To use this image, you will need to copy it to where you keep your boot−image and install it manually with LILO.
>
> For more information about configuring, building and installing your own kernel, check out the Kernel−HOWTO posted regularly to `comp.os.linux.answers`, and available from the Linux Documentation Project and its mirrors.

Trying out kerneld

Now reboot with the new kernel. When the system comes back up, you can run **ps ax**, and you should see a line for kerneld:

```
    PID TTY STAT  TIME COMMAND
     59  ?  S     0:01 /sbin/kerneld
```

One of the nice things with kerneld is that once you have the kernel and the daemon installed, very little setup is needed. For a start, try using one of the drivers that you built as a module; it is more likely than not that it will work without further configuration. If I build the floppy driver as a module, I could put a DOS floppy in the drive and type

```
  osiris:~ $ mdir a:
   Volume in drive A has no label
   Volume Serial Number is 2E2B−1102
   Directory for A:/

  binuti~1 gz       1942 02−14−1996  11:35a binutils−2.6.0.6−2.6.0.7.diff.gz
  libc−5~1 gz      24747 02−14−1996  11:35a libc−5.3.4−5.3.5.diff.gz
          2 file(s)        26689 bytes
```

The floppy driver works! It gets loaded automatically by kerneld when I try to use the floppy disk.

To see that the floppy module is indeed loaded, you can run **/sbin/lsmod** to list all currently loaded modules:

```
osiris:~ $ /sbin/lsmod
Module:        #pages:  Used by:
floppy            11    0 (autoclean)
```

The "(autoclean)" means that the module will automatically be removed by kerneld when it has not been used for more than one minute. So the 11 pages of memory (= 44kB, one page is 4 kB) will only be used while I access the floppy drive − if I don't use the floppy for more than a minute, they are freed. Quite nice, if you are short of memory for your applications!

How does kerneld know what module to load?

Although kerneld comes with builtin knowledge about the most common types of modules, there are situations where kerneld will not know how to handle a request from the kernel. This is the case with things like CD−ROM drivers or network drivers, where there are more than one possible module that can be loaded.

The requests that the kerneld daemon gets from the kernel is for one of the following items:

- a block−device driver
- a character−device driver
- a binary format
- a tty line discipline
- a filesystem
- a network device
- a network service (e.g. rarp)
- a network protocol (e.g. IPX)

The kerneld determines what module should be loaded by scanning the configuration file `/etc/conf.modules`[1]. There are two kinds of entries in this file: Paths where the module−files are located, and aliases assigning the module to be loaded for a given service. If you don't have this file already, you could create it by running

```
/sbin/modprobe −c | grep −v '^path' /etc/conf.modules
```

If you want to add yet another path directive to the default paths, you *must include all the default paths as well*, since a path directive in `/etc/conf.modules` will *replace*all the ones that modprobe knows by default!

Normally you don't want to add any paths by your own, since the built−in set should take care of all normal setups (and then some...), I promise!

On the other hand, if you just want to add an alias or an option directive, your new entries in `/etc/conf.modules` will be *added* to the ones that modprobe already knows. If you should *redefine* an alias or an option, your new entries in `/etc/conf.modules` will override the built−in ones.

Block devices

If you run **/sbin/modprobe −c**, you will get a listing of the modules that kerneld knows about, and what requests they correspond to. For instance, the request that ends up loading the floppy driver is for the block−device that has major number 2:

```
osiris:~ $ /sbin/modprobe −c | grep floppy
alias block−major−2 floppy
```

Why `block−major−2` ? Because the floppy devices `/dev/fd*` use major device 2 and are block devices:

```
osiris:~ $ ls −l /dev/fd0 /dev/fd1
brw−rw−rw−   1 root     root       2,   0 Mar  3  1995 /dev/fd0
brw−r−−r−−   1 root     root       2,   1 Mar  3  1995 /dev/fd1
```

Character devices

Character devices are dealt with in a similar way. E.g. the ftape floppy tape driver sits on major−device 27:

```
osiris:~ $ ls −lL /dev/ftape
crw−rw−−−−   1 root     disk      27,   0 Jul 18  1994 /dev/ftape
```

However, kerneld does not by default know about the ftape driver − it is not listed in the output from **/sbin/modprobe −c**. So to setup kerneld to load the ftape driver, I must add a line to the kerneld configuration file, `/etc/conf.modules`:

```
alias char−major−27 ftape
```

Network devices

You can also use the device name instead of the `char−major−xxx` or `block−major−yyy` setup. This is especially useful for network drivers. For example, a driver for an ne2000 netcard acting as `eth0` would be loaded with

```
alias eth0 ne
```

If you need to pass some options to the driver, for example to tell the module about what IRQ the netcard is using, you must add an "options" line:

```
options ne irq=5
```

This will cause kerneld to load the NE2000 driver with the command

```
/sbin/modprobe ne irq=5
```

Of course, the actual options available are specific to the module you are loading.

Binary formats

Binary formats are handled in a similar way. Whenever you try to run a program that the kernel does not know how to load, kerneld gets a request for `binfmt−xxx`, where `xxx` is a number determined from the first few bytes of the executable. So, the kerneld configuration to support loading the binfmt_aout module for ZMAGIC (a.out) executables is

```
alias binfmt-267 binfmt_aout
```

Since the magic number for ZMAGIC files is 267, if you check `/etc/magic`, you will see the number 0413; keep in mind that `/etc/magic` uses octal numbers where kerneld uses decimal, and octal 413 = decimal 267. There are actually three slightly different variants of a.out executables (NMAGIC, QMAGIC and ZMAGIC), so for full support of the binfmt_aout module we need

```
alias binfmt-264 binfmt_aout  # pure executable (NMAGIC)
alias binfmt-267 binfmt_aout  # demand-paged executable (ZMAGIC)
alias binfmt-204 binfmt_aout  # demand-paged executable (QMAGIC)
```

a.out, Java and iBCS binary formats are recognized automatically by kerneld, without any configuration.

Line disciplines (slip, cslip and ppp)

Line disciplines are requested with `tty−ldisc−x`, with `x` being usually 1 (for SLIP) or 3 (for PPP). Both of these are known by kerneld automatically.

Speaking of ppp, if you want kerneld to load the bsd_comp data compression module for ppp, then you must add the following two lines to your `/etc/conf.modules`:

```
alias tty-ldisc-3 bsd_comp
alias ppp0 bsd_comp
```

Network protocol families (IPX, AppleTalk, AX.25)

Some network protocols can be loaded as modules as well. The kernel asks kerneld for a protocol family (e.g. IPX) with a request for `net−pf−X` where `X` is a number indicating what family is wanted. E.g. `net−pf−3` is AX.25, `net−pf−4` is IPX and `net−pf−5` is AppleTalk; These numbers are determined by the AF_AX25, AF_IPX etc. definitions in the linux source file `include/linux/socket.h`. So to autoload the IPX module, you would need an entry like this in `/etc/conf.modules`:

```
alias net-pf-4 ipx
```

See Common Problems for information about how you can avoid some annoying boot−time messages related to undefined protocol families.

File systems

kerneld requests for filesystems are simply the name of the filesystem type. A common use of this would be to load the isofs module for CD−ROM filesystems, i.e. filesystems of type iso9660:

```
alias iso9660 isofs
```

Devices requiring special configuration

Some devices require a bit of extra configuration beyond the normal aliasing of a device to a module.

- Character devices on major number 10: The miscellaneous devices
- SCSI devices
- Devices that require special initialization

---

char−major−10 : Mice, watchdogs and randomness

Hardware devices are usually identified through their major device numbers, e.g. ftape is `char−major−27`. However, if you look through the entries in `/dev` for char major 10, you will see that this is a bunch of very different devices, including

- Mice of various sorts (bus mice, PS/2 mice)
- Watchdog devices
- The kernel `random` device
- APM (Advanced Power Management) interface

These devices are controlled by several different modules, not a single one, and therefore the kerneld configuration for these *misc. devices* use the major number *and* the minor number:

```
        alias char-major-10-1 psaux     # For PS/2 mouse
        alias char-major-10-130 wdt     # For WDT watchdog
```

You need a kernel version 1.3.82 or later to use this; earlier versions do not pass the minor number to kerneld, making it impossible for kerneld to figure out which of the misc. device modules to load.

Loading SCSI drivers: The `scsi_hostadapter` entry

Drivers for SCSI devices consist of a driver for the SCSI host adapter (e.g. an Adaptec 1542), and a driver for the type of SCSI device you use, e.g. a hard disk, a CD−ROM or a tape−drive. All of these can be loaded as modules. However, when you want to access e.g. the CD−ROM drive that is connected to the Adaptec card, the kernel and kerneld only knows that it needs to load the `sr_mod` module in order to support SCSI CD−ROM's; it does not know what SCSI controller the CD−ROM is connected to, and hence does not know what module to load to support the SCSI controller.

To resolve this, you can add an entry for the SCSI driver module to your `/etc/conf.modules` that tells kerneld which of the many possible SCSI controller modules it should load:

```
        alias scd0 sr_mod               # sr_mod for SCSI CD−ROM's ...
        alias scsi_hostadapter aha1542  # ... need the Adaptec driver
```

This only works with kernel version 1.3.82 or later.

This works if you have only one SCSI controller. If you have more than one, things become a little more difficult.

In general, you cannot have kerneld load a driver for a SCSI host adapter, if a driver for another host adapter is already installed. You must either build both drivers into your kernel (not as modules), or load the modules manually.

> **Tip:** There *is* a way that you can have kerneld load multiple SCSI drivers. James Tsiao came up with this idea:
>
> You can easily have kerneld load the second scsi driver by setting up the dependency in your modules.dep by hand. You just need an entry like:
>
> ```
>      /lib/modules/2.0.30/scsi/st.o: /lib/modules/2.0.30/scsi/aha1542.o
> ```
>
> To have kerneld load the `aha1542.o` before it loads `st.o`. My machine at home is set up almost exactly like the setup above, and it works fine for all my secondary scsi devices, including tape, cd−rom, and generic scsi devices. The drawback is that **depmod −a** can't autodetect these dependencies, so the user needs to add them by hand, and not run **depmod −a** on boot up. But once it is set up, kerneld will autoload the `aha1542.o` just fine.

You should be aware, that this technique only works if you have different kinds of SCSI devices on the two controllers, for example, hard disks on one controller, and cd−rom drives, tapes or generic SCSI devices on another.

When loading a module isn't enough: The `post−install` entry

Sometimes, just loading the module is not enough to get things working. For instance, if you have your sound card compiled as a module, it is often convenient to set a certain volume level. Only problem is, the setting vanishes the next time the module is loaded. Here is a neat trick from Ben Galliart (`<bgallia@luc.edu>`):

> The final solution required installing the setmix package and then adding the following line to my `/etc/conf.modules`:

```
post-install sound /usr/local/bin/setmix -f /etc/volume.conf
```

What this does is that after the sound module is loaded, kerneld runs the command indicated by the `post-install sound` entry. So the sound module gets configured with the command **/usr/local/bin/setmix −f /etc/volume.conf**.

This may be useful for other modules as well, for example the `lp` module can be configured with the `tunelp` program by adding

```
        post-install lp tunelp options
```

For kerneld to recognize these options, you will need a version of kerneld that is 1.3.69f or later.

> **Note:** An earlier version of this mini−HOWTO mentioned a pre−remove option, that might be used to run a command just before kerneld removed a module. However, this has never worked and its use is therefore discouraged − most likely, this option will disappear in a future kerneld release. The whole issue of module settings is undergoing some change at the moment, and may look different on your system by the time you read this.

Spying on kerneld

If you have tried everything, and just cannot figure out what the kernel is asking kerneld to do, there is a way of seeing the requests that kerneld receives, and hence to figure out what should go into `/etc/conf.modules`: The **kdstat** utility.

This nifty little program comes with the modules−package, but it is not compiled or installed by default. To build it, go to the directory where you have the kerneld sources and type **make kdstat**. Then, to make kerneld display information about what it is doing, run **kdstat debug** and kerneld will start spewing messages on the console about what it is doing. If you then try and run the command that you want to use, you will see the kerneld requests; these can be put into `/etc/conf.modules` and aliased to the module needed to get the job done.

To turn off the debugging, run **/sbin/kdstat nodebug**.

Special kerneld uses

I knew you would ask about how to setup the screen−saver module!

The `kerneld/GOODIES` directory in modules package has a couple of kernel patches for screen−saver and console−beep support in kerneld; these are not yet part of the official kernel, so you will need to install the kernel−patches and rebuild the kernel.

To install a patch, you use the patch command:

```
cd /usr/src/linux
patch −s −p1 /usr/src/modules−*/kerneld/GOODIES/blanker_patch
```

Then rebuild and install the new kernel.

When the screen−saver triggers, kerneld will run the command `/sbin/screenblanker`; this file may be anything you like, for example, a shell script that runs your favorite screen−saver.

When the kernel wants to unblank the screen, it sends a SIGQUIT signal to the process running `/sbin/screenblanker`. Your shell script or screen−saver should trap this, and terminate. Remember to restore the screen to the original text mode!

Common problems and things that make you wonder

1. Why do I get Cannot locate module for net−pf−`X` messages when I run ***/sbin/ifconfig****?*

2. After starting kerneld, my system slows to a crawl when I activate my ppp−connection

3. kerneld does not load my SCSI driver!

4. modprobe complains about gcc2_compiled being undefined

5. My sound driver keeps forgetting its settings for volume etc

6. DOSEMU needs some modules; how can I get kerneld to load those ?

7. Why do I get Ouch, kerneld timed out, message failed messages ?

8. Mount doesn't wait for kerneld to load the filesystem module

9. kerneld fails to load the `ncpfs` module

10. kerneld fails to load the `smbfs` module

11. I built everything as modules, and now my system cannot boot or kerneld fails to load the root filesystem module!

12. kerneld will not load at boot time; it complains about libgdbm

13. I get Cannot load module `xxx` but I just reconfigured my kernel without `xxx` support!

14. I rebuilt my kernel and modules, and still get messages about unresolved symbols when booting

15. I installed Linux 2.1/2.3 and now I cannot load any modules!

16. What about dial−on−demand networking?

1. Why do I get Cannot locate module for net−pf−`X` messages when I run **/sbin/ifconfig**?

Around kernel version 1.3.80, the networking code was changed to allow loading protocol families (e.g. IPX, AX.25 and AppleTalk) as modules. This caused the addition of a new kerneld request: `net-pf-X`, where `X` is a number identifying the protocol (see `/usr/src/linux/include/linux/socket.h` for the meaning of the various numbers). Unfortunately, **ifconfig** accidentally triggers these messages, so a lot of people get a couple of messages logged when the system boots and it runs **ifconfig** to setup the loopback device. The messages are harmless, and you can disable them by adding the lines

```
alias net-pf-3 off      # Forget AX.25
alias net-pf-4 off      # Forget IPX
alias net-pf-5 off      # Forget AppleTalk
```

to `/etc/conf.modules`. Of course, if you do use IPX as a module, you should not add a line to disable IPX.

2. After starting kerneld, my system slows to a crawl when I activate my ppp−connection

There have been a couple of reports of this. It seems to be an unfortunate interaction between kerneld and the tkPPP script that is used on some systems to setup and monitor the PPP connection. The script apparently runs loops while running **ifconfig**. This triggers kerneld, to look for the `net−pf−X` modules (see above), keeping the system load high and possibly pouring lots of Cannot locate module for `net−pf−X` messages into the system log. There is no known workaround, other than not use tkPPP, or change it to use some other way of monitoring the connection.

3. kerneld does not load my SCSI driver!

Add an entry for the SCSI hostadapter to your `/etc/conf.modules`. See the description of the `scsi_hostadapter` entry above.

4. modprobe complains about gcc2_compiled being undefined

This is a bug in the module utilities, that show up only with binutils 2.6.0.9 and later, and it is also documented in the release note for the binutils. So read that, or fetch an upgrade to the module−utilities that fix this bug.

5. My sound driver keeps forgetting its settings for volume etc

The settings for a module are stored inside the module itself when it is loaded. So when kerneld auto−unloads a module, any settings you have made are forgotten, and the next time the module loads it reverts to the default settings.

You can tell kerneld to configure a module by running a program after the module has been auto−loaded. See Pre/Post Install on the `post−install` entry.

6. DOSEMU needs some modules; how can I get kerneld to load those ?

You cannot. None of the dosemu versions, official or development versions, support loading the dosemu modules through kerneld. However, if you are running kernel 2.0.26 or later, you do not need the special dosemu modules any longer; just upgrade dosemu to 0.66.1 or higher.

7. Why do I get Ouch, kerneld timed out, message failed messages ?

When the kernel sends a request off to to kerneld, it expects to receive an acknowledgment back within one second. If kerneld does not send this acknowledgment, this message is logged. The request is retransmitted, and should get through eventually.

This usually happens on systems with a very high load. Since kerneld is a user−mode process, it is scheduled just like any other process on the system. At times of high load, it may not get to run in time to send back the acknowledgment before the kernel times out.

If this happens even when the load is light, try restarting kerneld. Kill the kerneld process, and start it again with the command **/usr/sbin/kerneld**. If the problem persists, you should mail a bug report to `<linux−kernel@vger.rutgers.edu>`, but *please* make sure that your versions of the kernel, kerneld and the module utilities are up−to−date before posting about the problem. Check the requirements in `linux/Documentation/Changes`

8. Mount doesn't wait for kerneld to load the filesystem module

There has been a number of reports that the mount(8) command does not wait for kerneld to load the filesystem module. **lsmod** does show that kerneld loads the module, and if you repeat the mount command immediately it will succeed. This appears to be a bug in the module−utilities version 1.3.69f that affects some Debian users. It can be fixed by getting a later version of the module−utilities.

9. kerneld fails to load the `ncpfs` module

You need to compile the ncpfs utilities with −DHAVE_KERNELD. See the ncpfs`Makefile`.

10. kerneld fails to load the `smbfs` module

You are using an older version of the smbmount utilities. Get the latest version (0.10 or later) from the SMBFS archive one TSX−11

11. I built everything as modules, and now my system cannot boot or kerneld fails to load the root filesystem module!

You cannot modularize *everything*: The kernel must have enough drivers built in for it to be able to mount your root filesystem, and run the necessary programs to start kerneld[2]. You cannot modularize

- the driver for the hard disk where your root filesystem lives
- the root filesystem driver itself
- the binary format loader for init, kerneld and other programs

12. kerneld will not load at boot time; it complains about libgdbm

Newer versions of kerneld need the GNU dbm library, `libgdbm.so`, to run. Most installations have this file in `/usr/lib`, but you are probably starting kerneld before the `/usr` filesystem is mounted. One symptom of this is that kerneld will not start during boot−up (from your rc−scripts), but runs fine if you start it by hand after that system is up. The solution is to either move the kerneld startup to after your `/usr` is mounted, or move the gdbm library to your root filesystem, e.g. to `/lib`.

13. I get Cannot load module `xxx` but I just reconfigured my kernel without `xxx` support!

The Slackware installation (possibly others) builds a default `/etc/rc.d/rc.modules` which does an explicit modprobe on a variety of modules. Exactly which modules get modprobed depends on the original kernel's configuration. You have probably reconfigured your kernel to exclude one or more of the modules that is getting modprobed in rc.modules, thus, the error message(s). Update your rc.modules by commenting out any modules you no longer use, or remove the `rc.modules` entirely and let kerneld load the modules when they are needed.

14. I rebuilt my kernel and modules, and still get messages about unresolved symbols when booting

You probably reconfigured/rebuilt your kernel and excluded some modules. You've got some old modules that you no longer use hanging around in the `/lib/modules` directory. The easiest fix is to delete your `/lib/modules/x.y.z` directory and do a **make modules_install** from the kernel source directory again. Note that this problem only occurs when reconfiguring your kernel without changing versions. If you see this error when moving to a newer kernel version you've got some other problem.

15. I installed Linux 2.1/2.3 and now I cannot load *any* modules!

Odd numbered Linux are development kernels. As such, it should be expected that things break from time to time. One of the things that has changed significantly is the way modules are handled, and where the kernel and modules are loaded into memory.

In brief, if you want to use modules with a development kernel, you must

- read the `Documentation/Changes` file and see what packages need upgrading on your system
- use the latest modutils package, available from AlphaBits on Red Hat or the mirror site at TSX−11

I recommend using at least kernel 2.1.29, if you want to use modules with a 2.1 kernel.

16. What about dial−on−demand networking?

kerneld originally had some support for establishing dial−up network connections on demand; trying to send packets to a network without being connected would cause kerneld to run the `/sbin/request_route` script to setup a PPP or SLIP connection.

This turned out to be a bad idea. Alan Cox of Linux networking fame wrote on the linux−kernel mailing list

> The request−route stuff is obsolete, broken and not required [...] Its also removed from 2.1.x trees.

Instead of using the request−route script and kerneld, I highly recommend Eric Schenk's diald package to manage your demand dialing.

Notes

[1]
: Some distributions call this file `modules.conf`

[2]
: Actually, this is not true. Late 1.3.x and all 2.x kernels support the use of an initial ram−disk that is loaded by LILO or LOADLIN; it is possible to load modules from this disk very early in the boot process. How to do it is described in the `linux/Documentation/initrd.txt` file that comes with the kernel source−files.