FreeBSD Documentation Portal

10.1. Synopsis

The kernel is the core of the FreeBSD operating system. It is responsible for managing memory, enforcing security controls, networking, disk access, and much more. While much of FreeBSD is dynamically configurable, it is still occasionally necessary to configure and compile a custom kernel.

After reading this chapter, you will know:

- When to build a custom kernel.
- How to take a hardware inventory.
- How to customize a kernel configuration file.
- How to use the kernel configuration file to create and build a new kernel.
- How to install the new kernel.
- How to troubleshoot if things go wrong.

All of the commands listed in the examples in this chapter should be executed as `root`.

10.2. Why Build a Custom Kernel?

Traditionally, FreeBSD used a monolithic kernel. The kernel was one large program, supported a fixed list of devices, and in order to change the kernel's behavior, one had to compile and then reboot into a new kernel.

Today, most of the functionality in the FreeBSD kernel is contained in modules which can be dynamically loaded and unloaded from the kernel as necessary. This allows the running kernel to adapt immediately to new hardware and for new functionality to be brought into the kernel. This is known as a modular kernel.

Occasionally, it is still necessary to perform static kernel configuration. Sometimes the needed functionality is so tied to the kernel that it can not be made dynamically loadable. Some security environments prevent the loading and unloading of kernel modules and require that only needed functionality is statically compiled into the kernel.

Building a custom kernel is often a rite of passage for advanced BSD users. This process, while time consuming, can provide benefits to the FreeBSD system. Unlike the GENERIC kernel, which must support a wide range of hardware, a custom kernel can be stripped down to only provide support for that computer's hardware. This has a number of benefits, such as:

- Faster boot time. Since the kernel will only probe the hardware on the system, the time it takes the system to boot can decrease.
- Lower memory usage. A custom kernel often uses less memory than the GENERIC kernel by omitting unused features and device drivers. This is important because the kernel code remains resident in physical memory at all times, preventing that memory from being used by applications. For this reason, a custom kernel is useful on a system with a small amount of RAM.

- Additional hardware support. A custom kernel can add support for devices which are not present in the GENERIC kernel.

Before building a custom kernel, consider the reason for doing so. If there is a need for specific hardware support, it may already exist as a module.

Kernel modules exist in /boot/kernel and may be dynamically loaded into the running kernel using kldload(8). Most kernel drivers have a loadable module and manual page. For example, the ath(4) wireless network driver has the following information in its manual page:

```
Alternatively, to load the driver as a module at boot time, place the
following line in loader.conf(5):

    if_ath_load="YES"
```

Adding `if_ath_load="YES"` to /boot/loader.conf will load this module dynamically at boot time.

In some cases, there is no associated module in /boot/kernel. This is mostly true for certain subsystems.

10.3. Finding the System Hardware

Before editing the kernel configuration file, it is recommended to perform an inventory of the machine's hardware. On a dual-boot system, the inventory can be created from the other operating system. For example, Microsoft®'s Device Manager contains information about installed devices.

Some versions of Microsoft® Windows® have a System icon which can be used to access Device Manager.

If FreeBSD is the only installed operating system, use dmesg(8) to determine the hardware that was found and listed during the boot probe. Most device drivers on FreeBSD have a manual page which lists the hardware supported by that driver. For example, the following lines indicate that the psm(4) driver found a mouse:

```
psm0: <PS/2 Mouse> irq 12 on atkbdc0
psm0: [GIANT-LOCKED]
psm0: [ITHREAD]
psm0: model Generic PS/2 mouse, device ID 0
```

Since this hardware exists, this driver should not be removed from a custom kernel configuration file.

If the output of `dmesg` does not display the results of the boot probe output, instead read the contents of /var/run/dmesg.boot.

Another tool for finding hardware is pciconf(8), which provides more verbose output. For example:

```
% pciconf -lv
ath0@pci0:3:0:0:        class=0x020000 card=0x058a1014 chip=0x1014168c rev=0x01 hdr=0x00
    vendor     = 'Atheros Communications Inc.'
    device     = 'AR5212 Atheros AR5212 802.11abg wireless'
    class      = network
    subclass   = ethernet
```

This output shows that the ath driver located a wireless Ethernet device.

The `-k` flag of man(1) can be used to provide useful information. For example, it can be used to

display a list of manual pages which contain a particular device brand or name:

```
# man -k Atheros
ath(4)                  - Atheros IEEE 802.11 wireless network driver
ath_hal(4)              - Atheros Hardware Access Layer (HAL)
```

Once the hardware inventory list is created, refer to it to ensure that drivers for installed hardware are not removed as the custom kernel configuration is edited.

10.4. The Configuration File

In order to create a custom kernel configuration file and build a custom kernel, the full FreeBSD source tree must first be installed.

If /usr/src/ does not exist or it is empty, source has not been installed. Source can be installed with Git using the instructions in “Using Git”.

Once source is installed, review the contents of /usr/src/sys. This directory contains a number of subdirectories, including those which represent the following supported architectures: amd64, i386, powerpc, and sparc64. Everything inside a particular architecture’s directory deals with that architecture only and the rest of the code is machine independent code common to all platforms. Each supported architecture has a conf subdirectory which contains the GENERIC kernel configuration file for that architecture.

Do not make edits to GENERIC. Instead, copy the file to a different name and make edits to the copy. The convention is to use a name with all capital letters. When maintaining multiple FreeBSD machines with different hardware, it is a good idea to name it after the machine’s hostname. This example creates a copy, named MYKERNEL, of the GENERIC configuration file for the `amd64` architecture:

```
# cd /usr/src/sys/amd64/conf
# cp GENERIC MYKERNEL
```

MYKERNEL can now be customized with any `ASCII` text editor. The default editor is vi, though an easier editor for beginners, called ee, is also installed with FreeBSD.

The format of the kernel configuration file is simple. Each line contains a keyword that represents a device or subsystem, an argument, and a brief description. Any text after a `#` is considered a comment and ignored. To remove kernel support for a device or subsystem, put a `#` at the beginning of the line representing that device or subsystem. Do not add or remove a `#` for any line that you do not understand.

It is easy to remove support for a device or option and end up with a broken kernel. For example, if the ata(4) driver is removed from the kernel configuration file, a system using `ATA` disk drivers may not boot. When in doubt, just leave support in the kernel.

In addition to the brief descriptions provided in this file, additional descriptions are contained in NOTES, which can be found in the same directory as GENERIC for that architecture. For architecture independent options, refer to /usr/src/sys/conf/NOTES.

When finished customizing the kernel configuration file, save a backup copy to a location outside of /usr/src.

Alternately, keep the kernel configuration file elsewhere and create a symbolic link to the file:

```
# cd /usr/src/sys/amd64/conf
# mkdir /root/kernels
# cp GENERIC /root/kernels/MYKERNEL
```

```
# ln -s /root/kernels/MYKERNEL
```

An `include` directive is available for use in configuration files. This allows another configuration file to be included in the current one, making it easy to maintain small changes relative to an existing file. If only a small number of additional options or drivers are required, this allows a delta to be maintained with respect to GENERIC, as seen in this example:

```
include GENERIC
ident MYKERNEL

options         IPFIREWALL
options         DUMMYNET
options         IPFIREWALL_DEFAULT_TO_ACCEPT
options         IPDIVERT
```

Using this method, the local configuration file expresses local differences from a GENERIC kernel. As upgrades are performed, new features added to GENERIC will also be added to the local kernel unless they are specifically prevented using `nooptions` or `nodevice`. A comprehensive list of configuration directives and their descriptions may be found in config(5).

To build a file which contains all available options, run the following command as `root`:

```
# cd /usr/src/sys/arch/conf && make LINT
```

10.5. Building and Installing a Custom Kernel

Once the edits to the custom configuration file have been saved, the source code for the kernel can be compiled using the following steps:

Procedure: Building a Kernel

1. Change to this directory:
2. Compile the new kernel by specifying the name of the custom kernel configuration file:

   ```
   # make buildkernel KERNCONF=MYKERNEL
   ```

3. Install the new kernel associated with the specified kernel configuration file. This command will copy the new kernel to /boot/kernel/kernel and save the old kernel to /boot/kernel.old/kernel:

   ```
   # make installkernel KERNCONF=MYKERNEL
   ```

4. Shutdown the system and reboot into the new kernel. If something goes wrong, refer to The kernel does not boot.

By default, when a custom kernel is compiled, all kernel modules are rebuilt. To update a kernel faster or to build only custom modules, edit /etc/make.conf before starting to build the kernel.

For example, this variable specifies the list of modules to build instead of using the default of building all modules:

```
MODULES_OVERRIDE = linux acpi
```

Alternately, this variable lists which modules to exclude from the build process:

```
WITHOUT_MODULES = linux acpi sound
```

Additional variables are available. Refer to make.conf(5) for details.

10.6. If Something Goes Wrong

There are four categories of trouble that can occur when building a custom kernel:

`config` fails

If `config` fails, it will print the line number that is incorrect. As an example, for the following message, make sure that line 17 is typed correctly by comparing it to GENERIC or NOTES:

```
config: line 17: syntax error
```

`make` fails

If `make` fails, it is usually due to an error in the kernel configuration file which is not severe enough for `config` to catch. Review the configuration, and if the problem is not apparent, send an email to the FreeBSD general questions mailing list which contains the kernel configuration file.

The kernel does not boot

If the new kernel does not boot or fails to recognize devices, do not panic! Fortunately, FreeBSD has an excellent mechanism for recovering from incompatible kernels. Simply choose the kernel to boot from at the FreeBSD boot loader. This can be accessed when the system boot menu appears by selecting the "Escape to a loader prompt" option. At the prompt, type `boot kernel.old`, or the name of any other kernel that is known to boot properly.

After booting with a good kernel, check over the configuration file and try to build it again. One helpful resource is /var/log/messages which records the kernel messages from every successful boot. Also, dmesg(8) will print the kernel messages from the current boot.

When troubleshooting a kernel make sure to keep a copy of a kernel that is known to work, such as GENERIC. This is important because every time a new kernel is installed, kernel.old is overwritten with the last installed kernel, which may or may not be bootable. As soon as possible, move the working kernel by renaming the directory containing the good kernel:

```
# mv /boot/kernel /boot/kernel.bad
# mv /boot/kernel.good /boot/kernel
```

The kernel works, but ps(1) does not

If the kernel version differs from the one that the system utilities have been built with, for example, a kernel built from -CURRENT sources is installed on a -RELEASE system, many system status commands like ps(1) and vmstat(8) will not work. To fix this, recompile and install a world built with the same version of the source tree as the kernel. It is never a good idea to use a different version of the kernel than the rest of the operating system.