Modular Kernel

From OSDev Wiki

Contents

Kernel Designs

Models

Monolithic Kernel
Microkernel
Hybrid Kernel
Exokernel
Nano/Picokernel
Cache Kernel
Virtualizing Kernel
Megalithic Kernel

Other Concepts

Modular Kernel
Higher Half Kernel
64-bit Kernel
Bare Bones

What is a Modular Kernel

A modular kernel is an attempt to merge the good points of kernel-level drivers and third-party drivers. In a modular kernel, some part of the system core will be located in independent files called *modules* that can be added to the system at run time. Depending on the content of those modules, the goal can vary such as:

- only loading drivers if a device is actually found
- only load a filesystem if it gets actually requested
- only load the code for a specific (scheduling/security/whatever) policy when it should be evaluated
- etc.

The basic goal remains however the same: keep what is loaded at boot-time minimal while still allowing the kernel to perform more complex functions. The basics of modular kernel are very close to what we find in implementation of *plugins* in applications or *dynamic libraries* in general.

What are some advantages and disadvantages for a Modular Kernel?

Advantages

- The most obvious is that the kernel doesn't have to load everything at boot time; it can be expanded as needed. This can decrease boot time, as some drivers won't be loaded unless the hardware they run is used (NOTE: This boot time decrease can be negligible depending on what drivers are modules, how they're loaded, etc.)
- The core kernel isn't as big
- If you need a new module, you don't have to recompile.

Disadvantages

- It may lose stability. If there is a module that does something bad, the kernel can crash, as modules should have full permissions.
- ...and therefore security is compromised. A module can do anything, so one could easily write an evil module to crash things. (Some OSs, like Linux, only allow modules to be loaded by the root user.)
- Coding can be more difficult, as the module cannot reference kernel procedures without kernel symbols.

What does a Modular Kernel look like ?

There are several components that can be identified in virtually every modular kernel:

the core
: this is the collection of features in the kernel that are absolutely mandatory regardless of whether you have modules or not.

the modules loader
: this is a part of the system that will be responsible of preparing a module file so that it can be used as if it was a part of the core itself.

the kernel symbols table
: This contains additional information about the core and loaded modules that the module loader needs in order to *link* a new module to the existing kernel.

the dependencies tracking
: As soon as you want to *unload* some module, you'll have to know whether you can do it or not. Especially, if a module *X* has requested symbols from module *Z*, trying to unload *Z* while *X* is present in the system is likely to cause havoc.

modules
: Every part of the system you might want (or don't want) to have.

How can such a system boot in first place ?

Modularization must be done within certain limits if you still want your system to be able to boot. Pushing *all* the filesystems and device drivers (including boot device driver) into module will probably make the boot time a hard time. Following solutions can however be used:

- The kernel is provided with an extremely simple filesystem (e.g. SCO's BFS) driver and that filesystem contains modules to access the rest of system storage (e.g. module for ATA, SCSI, EXT2FS, ReiserFS, FAT, NTFS ...).
- The kernel comes with a built-in native file system driver and other storage modules as well as primary configuration files should be stored using that native filesystem. This was the approach followed by Linux, and as soon as some people decided to have reiser everywhere, ext2-fs only kernels start having trouble on some machines.
- The bootloader knows it should not only load the *kernel* but also a collection of pre-configured modules so that the kernel only needs to check those pre-loaded modules and initialize them to access other modules and primary configuration files. This basically means that your bootloader is somehow an OS of its own such as GRUB

Anyway, ramdisk drivers and dedicated boot partitions/reserved sectors will be your friends.

See Also

Forum Threads

- Design of a basic module loader
- Calling a function knowing its name
- Ideas for IPC
- Coff i386 relocations
- Device Drivers Interface
- Loading Drivers into Kernel
- Design & Implementation of Extensible OS