Module Programming

Content

- What is a Kernel module?
- Application Programming vs Module Programming
- Module Utilities
- Kernel message logging
- Module Parameters
- Module dependencies (Export Symbols)
- Kernel specific GCC Extensions (__init and __exit)
- The Role of the Device Driver? What are the things required to writing a device driver?

What is a Kernel module?

- **Modules:** Add a given functionality to the kernel (drivers, file system support, and many others).
- **Modules** are pieces of code that can be loaded and unloaded in to the kernel upon demand. They extend the functionality of the kernel without the need to reboot the system.
- **Modules** are pre-built binaries (similar to shared object) with can help load code & data in to the kernel memory segment.

Loadable kernel modules

- **Kernel symbol table:** The names & addresses of all the kernel functions are present in their table.
- **Advantages of Kernel Modules:**
  - Modules make it easy to develop drivers without rebooting: **load, test, unload, rebuild, load...**
  - Useful to keep the kernel image size to the minimum (essential in GNU/Linux distributions for PCs).
  - Also useful to reduce boot time: you don't spend time initializing devices and kernel features that you only need later.
- **Drawbacks of Kernel Modules:**
  - Caution: once loaded, have full access to the whole kernel address space. No particular protection.

User space vs Kernel space

User Space - Application

1. Application starts with main() function and when main() function returns the application terminates.

2. All the Standard C library functions are available to the application.

3. When we build an application, it will get compiled and linked to libraries so that executable file will be generated.

Kernel Space - Module

1. In kernel Module, there will not be any main() function.

2. But kernel module can not call standard library functions. It can call only kernel functions, which are present, in side the kernel.

3. But when we build kernel module, it only will get compiled, it will not get linked to kernel functions, as kernel in not available as a library.

Module Utilities

To dynamically load or unload a driver, use these commands, which reside in the /sbin directory, and must be executed with root privileges:

- lsmod – Lists currently loaded modules
- modinfo <module_file> - Gets information about a module
- insmod <module_file> – Inserts/loads the specified module file
- modprobe <module> – Inserts/loads the module, along with any dependencies
- rmmod <module> – Removes/unloads the module.

Kernel message logging

printf	printk
Floating point used (%f,%lf)	No floating point
Dump the output to some console.	All printk calls put this output in to the log ring buffer of the kernel.

- All the printk output, by default **/var/log/messages** for all log values.
- This file is not readable by the normal user. Hence, user space utility **“dmesg”**.

Kernel message logging

- There are eight macros defined in linux/kernel.h in the kernel source, namely:

#define KERN_EMERG “<0>” /* system is unusable */

#define KERN_ALERT “<1>” /* action must be taken immediately */

#define KERN_CRIT “<2>” /* critical conditions */

#define KERN_ERR “<3>” /* error conditions */

#define KERN_WARNING “<4>” /* warning conditions */

#define KERN_NOTICE “<5>” /* normal but significant condition */

#define KERN_INFO “<6>” /* informational */

#define KERN_DEBUG “<7>” /* debug-level messages */

Kernel specific GCC Extensions

- Kernel 'C' is standard 'C' with some additional extensions from the 'C' compiler, GCC.
- Macros __init & __exit are just two of there extensions.
- However, these do not have any relevance in case we are using them for a dynamically loadable driver, but instead, when the same code gets built into the kernel
- This is a beautiful example of how the kernel and GCC work hand-in-hand to achieve a lot of optimization, and many other tricks we will see as we go along.
- And that is why the Linux kernel can only be compiled using GCC-based compilers—a closely knit bond.

Kernel functions return guidelines

- The kernel programming guideline for returning values from a function. Any kernel function needing error handling typically returns an integer-like type.
- For an **error**, we return a **negative** number: a minus sign appended with a macro that is available from a kernel header include linux/errno.h
- For **success**, **zero** is the most common return value.
- For **some additional information**, a **positive** value is returned, the value indicating the information, such as the number of bytes transferred by the function.

The Role of a Device Driver

- Mechanisms not Policies
- Concurrent Accesses
  - Case I: One Device, many applications concurrently
  - Case II: Many Devices, one application
  - Case III: Many Devices, Many Applications
- Platform Portability should be considered
- System Modularity should be considered