Loadable Kernel Modules

CPSC 457

Mohammad Reza Zakerinasab
May 18, 2016
Outline

- About Linux Kernel (quick review)
- An Introduction to Linux Kernel Modules
- Help on Assignment 1

Have your Linux VM boot up and logged in.
About Linux Kernel

- The layered structure of Linux:
  - User applications / code libraries
  - System calls
  - Kernel

- The kernel is responsible for maintaining the important abstractions of the operating system
  - Kernel code executes in kernel mode with full access to all the physical resources of the computer
  - Kernel symbols and data structures are mapped into multiple process address spaces.

- The system libraries define a standard set of functions through which applications interact with the kernel, and implement much of the operating-system functionality that does not need the full privileges of kernel code.
Modifying the Kernel?

- What if we need to modify something in the kernel?
  - Add a new hardware?
  - Support a new file system?
  - Introduce a new networking protocol?

- Kernel recompilation is tiresome and error prone.
  - What the user thinks about this?

- We need a mechanism to modify the kernel while the kernel is running, and without recompilation.
  - Solution: Loadable Kernel Modules (LKM)
Loadable Kernel Modules

- Kernel modules are sections of kernel code that can be compiled, loaded, and unloaded independent of the rest of the kernel.
- A kernel module may typically implement a device driver, a file system, or a networking protocol.
- The module interface allows third parties to write and distribute, on their own terms, device drivers or file systems that could not be distributed under the GPL.
  
  **Warning**: Non-GPL LKM's will “taint” the kernel.

- Kernel modules allow a Linux system to be set up with a standard, minimal kernel, without any extra device drivers built in.
Managing LKMs by the Kernel

- Modules are stored in the file system as ELF object files.
  - Executable and Linkable Format
  - Common format for executables, object code, shared libraries, and core dumps.
- The `insmod` program can be used to load a compiled LKM.
- For each module the kernel allocates memory area containing:
  - a module object
  - null terminated string that represents module's name
  - the code that implements the functions of the module
- Symbols used in an LKM are available in the kernel space (see `/proc/kallsyms`).
- The kernel keeps track of the use of modules, so that no module is unloaded while another module or kernel is using it (see `/proc/modules`).
Utilities to Manage LKMs

- **lsmod**
  - Lists modules that are already loaded.
  - Check it on your Linux VM.
  - Also check `cat /proc/modules`
- **modinfo**
  - Displays module information.
  - Check `modinfo usb_storage | more`
- **insmod**
  - Inserts module into the kernel
  - `insmod <module-name>.ko`
- **rmmod**
  - Removes module from the kernel, only if no one is using it.
  - `rmmod <module-name>.ko`
Hello World from the Kernel (hello-1.c)

```c
#include <linux/module.h>    /* Needed by all modules */
#include <linux/kernel.h>    /* Needed for KERN_INFO */

MODULE_LICENSE("GPL");

int init_module(void)
{
    printk(KERN_INFO "Hello world!\n");
    // A non 0 return means init_module failed; module can't be loaded.
    return 0;
}

void cleanup_module(void)
{
    printk(KERN_INFO "Goodbye.\n");
}
```

http://www.tldp.org/LDP/lkmpg/2.6/html/x121.html
Compiling the Kernel Module

```
obj-m += hello-1.o

all:
    make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules

clean:
    make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
```

http://www.tldp.org/LDP/lkmpg/2.6/html/x181.html
Compiling the Kernel Module

- Run `make`
- Files created after building the module:
  - `hello-1.o`
    - Module object file before linking
  - `hello-1.mod.c`
    - Contains module information
  - `hello-1.mod.o`
    - `hello-1.mod.c` after compilation
  - `modules.order`
    - The order in which two or modules get linked
  - `modules.symvers`
    - Symbol versions, if any
  - `hello-1.ko`
    - Kernel module object file after linking `hello-1.o` and `hello-1.mod.o`
Introducing `printk()`

- `printk()` provides a logging mechanism for the kernel, and is used to log information or give warnings to the syslog.
  - Similar to `printf`. Kernel does not support float numbers.
- `printk()` supports 8 priority level for the message, from emergency messages to warning messages.
  - `KERN_EMERG`
  - `KERN_ALERT`
  - `KERN_CRIT`
  - `KERN_ERR`
  - `KERN_WARNING`
  - `KERN_NOTICE`
  - `KERN_INFO`
  - `KERN_DEBUG`
- You can ignore the logging level. The default logging level is assumed.
Testing the Kernel Module

- Insert the kernel module into the kernel
  - `sudo insmod hello-1.ko`
- Check if the module is in the list
  - `lsmod`
  - Can you find out the size of the module and the number of processes that are using the module?
- Check the kernel logs
  - `dmesg`
  - Can you see the module message? Now try this:
    - `dmesg | tail -1`
- Remove the module and check the kernel log messages
  - `rmmod hello-1.ko`
  - `dmesg | tail -1`
**Introducing `dmesg`**

- `dmesg` command writes the kernel messages into standard output (which by default is the display screen).
- `dmesg` reads the kernel messages from a special buffer called the *kernel ring buffer*.
  - Buffer is a portion of the computer memory used to temporarily store data that is sent to (or received from) an external device.
  - A ring buffer overwrites the old data when new data is available.
- Use `man dmesg`. How you can print more information about the messages?
- What about this?
  - `sudo cat /var/log/messages | tail -2`
Kernel Modules vs. Userland Applications

- **Applications**
  - Run in user space
  - Execute in the user mode
  - Restricted access to hardware
  - Each application has its own address space

- **Kernel modules**
  - Run in kernel space
  - Execute in the supervisor mode
  - Everything is allowed
  - Share the same address space
#include <linux/module.h>  /* Needed by all modules */
#include <linux/kernel.h>   /* Needed for KERN_INFO */
#include <linux/init.h>     /* Needed for the macros */

MODULE_LICENSE("GPL");

int my_init_function(void)
{
    printk(KERN_INFO "Hello world 2\n");
    return 0;
}

void my_cleanup_function(void)
{
    printk(KERN_INFO "Goodbye 2.\n");
}

module_init(my_init_function);
module_exit(my_cleanup_function);
#include <linux/module.h> /* Needed by all modules */
#include <linux/kernel.h> /* Needed for KERN_INFO */
#include <linux/init.h> /* Needed for the macros */

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Peter Jay Salzman"); /* Who wrote this module? */
MODULE_DESCRIPTION("Hello Module"); /* What does this module do */
MODULE_SUPPORTED_DEVICE("testdevice");

static int __init init_hello_world(void)
{
    printk(KERN_INFO "Hello world \n");
    return 0;
}

static void __exit cleanup_hello_world(void)
{
    printk(KERN_INFO "Goodbye \n");
}

module_init(init_hello_world);
module_exit(cleanup_hello_world);

http://www.tldp.org/LDP/lkmpg/2.6/html/x279.html Minor modifications to make the code compilable on VM.
Passing Command Line Arguments to a Module

- If needed, modules can take command line arguments.
  - ./insmod mymodule.ko myvariable=5
  - But not with the argc/argv structure.

- To pass an argument to your module:
  - Declare respective variables as global.
  - Use the `module_param()` macro (`linux/moduleparam.h`).
  - At runtime, `insmod` will fill the variables with any command line arguments that are given.
  - Module parameters can be integer data types (short, int, long), signed or unsigned, strings, or arrays.
  - We don’t cover the details. If interested, please see [http://www.tldp.org/LDP/lkmpg/2.6/html/x323.html](http://www.tldp.org/LDP/lkmpg/2.6/html/x323.html).
Allocating Memory in the Kernel [1]

- Allocating memory in the kernel is not as simple as allocating memory in user space.
  - The kernel's memory is not pageable.
  - The kernel usually wants physically contiguous memory.
  - Often, the kernel must allocate the memory without sleeping.
  - Mistakes in the kernel have a much higher price than they do elsewhere.

- A General-Purpose Allocator
  - `#include <linux/slab.h>
  ```
  void * kmalloc(size_t size, int flags);
  ```
  - Similar to user space's `malloc()`, except that it takes a second argument, `flags`.  

Allocating Memory in the Kernel [2]

- The flags field controls the behavior of memory allocation.
  - Three groups: action modifiers, zone modifiers and types.
- Action modifiers tell the kernel how to allocate memory.
  - Example: If the kernel can sleep (block) during allocation.
- Zone modifiers tell the kernel from where the request should be satisfied.
  - Example: If the memory must be accessible by hardware through direct memory access (DMA).
- Type flags specify a type of allocation. They group together relevant action and zone modifiers into a single type.
- We don’t cover the details. If interested, please see [http://www.linuxjournal.com/article/6930](http://www.linuxjournal.com/article/6930).
Further Reading

The Linux Kernel Programming Guide
http://www.tldp.org/LDP/lkmpg/2.6/html/

Writing a Linux Kernel Module
http://derekmolloy.ie/writing-a-linux-kernel-module-part-1-introduction/

Kernel Korner - Allocating Memory in the Kernel
http://www.linuxjournal.com/article/6930