Linux Kernel
Linux Loadable Kernel Module(LKM)

By : TUSHAR DAVE N.

This document has been created using Internet sources. Any user can modify or distribute this document.

History :

"Despite being "MONOLITHIC" IN SENSE OF THE WHOLE kernel running in single protection domain,the Linux kernel is modular...",said by Robert Love in the book, "Linux Kernel Development". This means that user can insert or remove code(module) from the kernel at runtime. The whole magic is supported by the Linux Loadable kernel module(LKM).

LKM did not exist in the beginning. The concept and implementation of LKM has started around 1995.(Linux kernel 1.2)"Fifteen years ago, in 1992, two heavyweights in the field of operating system design were entangled in what would become a classic discussion on the fundamentals of kernel architecture (Abrahamsen (no date)). The heavyweights in question were Andrew S. Tanenbaum, the author of “Operating systems design and implementation” (Tanenbaum & Woodhull, 2006) and Linus Torvalds, the then-young and upcoming writer of what would become one of the most successful operating system kernels in computer history: the Linux kernel. Like most of his colleagues at that time, Tanenbaum was a proponent of the micro kernel architecture, while Torvalds, more of a pragmatist than Tanenbaum, argued that the monolithic design made more sense. The discussion gained an extra dimension because of the fact that Torvalds had studied Tanenbaum’s book in great detail before he started writing the first version of the Linux kernel." (Thom Holwerda, March 2007)

Monolithic vs Micro kernel

Monolithic kernel:

Monolithic means kernel is designed as a single process in single address space. This means if there is one bug in the kernel, let say in the network code of the kernel then whole kernel will crash and you must reboot the system. The proponent of this design cites simplicity as a main goal. Because the whole kernel is in one address space so the communication is simple and any defined function can be called from anywhere in the kernel. In order to achieve the goal proponents of the monolithic kernel also argue that the design itself force the programmer to write clear and bug-free code because the consequences of the bugs can be devastating. Unfortunately, writing bug-free code is nearly impossible, Fig. 1

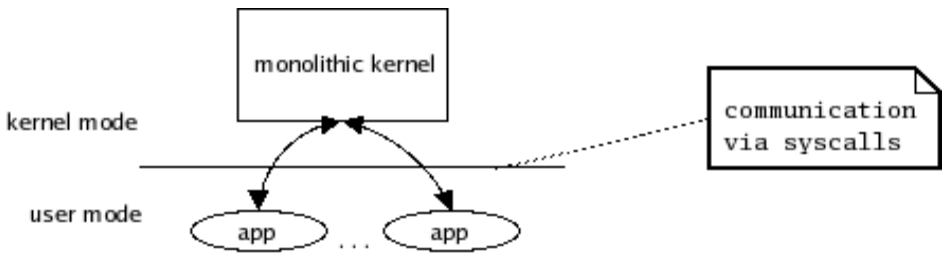

Fig. 1

Micro-kernel:

Micro-kernel design solves the problem that bug can cause by moving critical parts of the kernel from kernel space to user space, where they run as a unique process (called servers). These servers cannot talk to each other without permission to do so. So the bug in network system that crash the monolithic-kernel can not crash the system in micro-kernel design. Servers can talk to each other with IPC methods. This model can take advantage of memory protection features provided by modern processors. The design seems interesting and better but on the contrast there is a drawback of communication between the servers that significantly affect the performance of the system though on the other hand they produce reliable system,Fig 2.

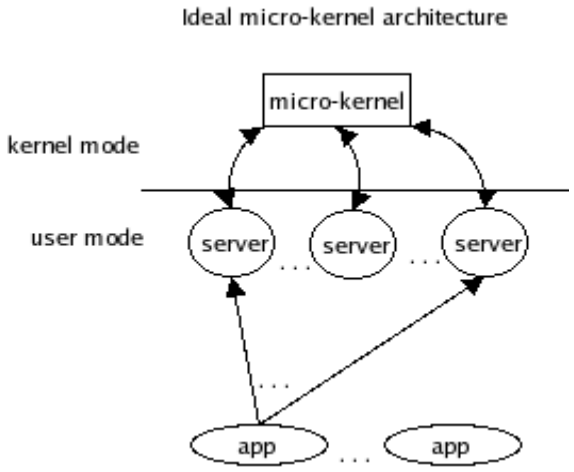

Fig. 2

(*now onwards the term module means loadable kernel module (LKM))

What is LKM?

LKM is a module that can be loaded or unloaded dynamically at run time without rebooting or recompiling the kernel. This dynamically loadable objects into the base kernel image at runtime is known as loadable kernel module.

LKM allows user to add device driver , file system driver , system call etc., into the kernel with an ease. Before LKM was not exist, kernel developers must recompile the whole kernel for testing, debugging after inserting module/code into the kernel.

Now a days using LKM user can insert the required modules at run time whenever necessary. This has reduced the size of the base kernel image. As many of you have seen during kernel building process when you type 'make config' or 'make menuconfig', it asks you lots of questions about the configuration options that controls the build process. These options are either boolean (yes/no) or tristate (yes/no/modules). When you specify 'module' settings it means you want to compile object as module (that is dynamically loadable object).In the case of tristate 'yes' option explicitly means compiling the code into main kernel image. So be specific, whenever you are not sure load the objects as module hence reduce the size of the base kernel.

"LKM are not user program, they are part of kernel and if misused then they will crash the system."

Lets take a look at the process of writing "hello word" kernel module.
Prerequisites before writing loadable kernel module,

1. of course a Linux system with Linux kernel source header files, generally it is installed in
   usr/src/linux-x.x.x/ directory (or source code of linux kernel)
2. Kernel version must match with the kernel you are running

****

```
//File Name hello.c

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

int init_module(void)
{
        printk(KERN_INFO "Hello world 1.\n");

         // A non 0 return means init_module failed; module can't be loaded.
         return 0;
}
void cleanup_module(void)
{
        printk(KERN_INFO "Goodbye world 1.\n");
}
```

****

Each modules must have two functions entry and exit .In hello.c module, they are - init_module(void) which is called when user insert the module with 'insmod' command. 'init_module' either register new function handler or replace some functions of the kernel with

the other functions. 'cleanup_module' called when user do 'rmmod' and it cleans up whatever insmod did. Because you are doing kernel programming your old friend **printf** is not any more valid here. So to print the message in kernel we can use **printk** with proper priority. Note that kernel output goes to the kernel ring buffer and not to stdout, because stdout is process specific. To inspect messages on the kernel ring buffer, you can use the **dmesg** utility. The message can be displayed in /log/var/message if your /etc/syslog.conf is properly set (most of the time it is).

Modules are compiled not like C program. Modules are basically compiled inside kernel source tree or externally in your own working directory. As there is no significant difference between them ,I am showing you how to compile externally. To compile the kernel module that we just wrote, we need to write the Makefile that in turn call the top level Makefile of the kernel and build the kobject(.ko).

****

```
//Makefile
obj-m := hello.o
// creates module object , if you want to create normal object file then do obj-y := hello.o

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

****

Because your module is written outside the source tree your Makefile must tell the kernel where to find the needed information for compilation and the Base kernel Makefile(in Linux world the top level Makefile is also referred as 'kbuild').That is the reason we use "make -C /lib/modules/$(uname -r)/build", where build is softlink to /usr/src/linux (-C option for changing the directory before make). Also in the top level Makefile you can see if you want to compile the modules then you must specify the M="present working directory" (before 2.6 in 2.4 there is SUBDIR option instead of M which is still valid for compatibility.)

(The information about base Makefile can be found in /Documentation/kbuild/Makefile.txt and the file resides in /usr/src/linux/)

The code to handle LKM is written in kernel/module.c and the LKM loader is in kernel/kmod.c
*/

lets do 'make' and compile the module.....

****

```
tushar@tushar-laptop:~$ make
make -C /lib/modules/2.6.20-16-generic/build M=/home/tushar modules
make[1]: Entering directory `/usr/src/linux-headers-2.6.20-16-generic'
  CC [M]  /home/tushar/hello.o
  Building modules, stage 2.
  MODPOST 1 modules
  CC      /home/tushar/hello.mod.o
  LD [M]  /home/tushar/hello.ko
make[1]: Leaving directory `/usr/src/linux-headers-2.6.20-16-generic'
```

****

The generated files are...

hello.o , hello.ko, hello.mod.c , hello.mod.o

Back in 2.4 days, the module file are generated with only .o extension, which is changed in 2.6,the new kernel module object files are named with .ko extension because of the special modinfo section. you can use 'modinfo hello.ko' to see the information about the module

****

```
tushar@tushar-laptop:~$ modinfo hello.ko
filename:       hello.ko
srcversion:     140276773A3090F6F33891F
depends:
vermagic:       2.6.20-16-generic SMP mod_unload 586
tushar@tushar-laptop:~$
```

****

you can insert the module using '**insmod'** command (you must be root)

```
tushar@tushar-laptop:~$ insmod hello.ko
```

" insmod calls the sys_init_module system call that is defined in kernel/module.c and load the module into kernel".Again, you can see the generated message by the kernel, after inserting the module, in /log/var/message or you can use 'dmesg' command on shell prompt to see the message generated by the kernel.

The message is like.....

```
[11532.104000] hello: module license 'unspecified' taints kernel.
[11532.104000] Hello world 1.
```

“Unspecified taints kernel? “-> this is because in kernel 2.4 and later, a mechanism was devised to identify code licensed under the GPL (and friends) so people can be warned that

the code is non open-source. This is accomplished by the **MODULE_LICENSE()** macro which is demonstrated in the next piece of code. By setting the license to GPL, you can skip the warning from being printed. This license mechanism is defined and documented in linux/module.h:

“Similarly, MODULE_DESCRIPTION() is used to describe what the module does, MODULE_AUTHOR() declares the module's author, and MODULE_SUPPORTED_DEVICE() declares what types of devices the module supports. so when you add these macros with values you will see the message will be gone.”[1]

Finally the module can be unloaded by **rmmod modulename**

```
tushar@tushar-laptop:~$ rmmod hello
tushar@tushar-laptop:~$ dmesg
[11916.816000] Goodbye world 1.
```

Since Linux 2.4 arrival, you can rename the init and cleanup functions of your modules; no strict rule to called them init_module() and cleanup_module() respectively. This is done with the module_init() and module_exit() macros. These macros are defined in linux/init.h. The only caveat is that your init and cleanup functions must be defined before calling the macros, otherwise you'll get compilation errors. Here's an example of this technique. We are also adding MODULE_LISENCE("GPL") and other macros.

```
/****************************************************************************** File name : csc239.c ****************************/

#include <linux/module.h>
#include <linux/kernel.h>
#define LICENSE "GPL"
#define DESCRIPTION "CSc239 Presentation"
#define AUTHOR "TUSHAR DAVE"

MODULE_LICENSE(LICENSE);
MODULE_DESCRIPTION(DESCRIPTION);
MODULE_AUTHOR(AUTHOR);

int my_variable=1;
int my_init(void)
{

	printk(KERN_INFO "how r you?\n");
	return 0;
}

void my_exit(void)
{
	printk(KERN_INFO "bye\n");
```

```
}
EXPORT_SYMBOL_GPL(my_variable);/* you can see the exported variable in /proc/kallsyms
*/
module_init(my_init);
module_exit(my_exit);
```

****

Compile and insert with 'make';'insmod modulename.ko', in our case 'insmod hello.ko'

You can see the loaded module using **lsmod** command
'**lsmod' lists the loaded modules. lsmod reads the file /proc/modules.**

```
tushar@tushar-laptop:~$ make
make -C /lib/modules/2.6.20-16-generic/build M=/home/tushar
make[1]: Entering directory `/usr/src/linux-headers-2.6.20-16-generic'
  CC [M]  /home/tushar/csc239.o
  Building modules, stage 2.
  MODPOST 1 modules
  CC      /home/tushar/csc239.mod.o
  LD [M]  /home/tushar/csc239.ko
make[1]: Leaving directory `/usr/src/linux-headers-2.6.20-16-generic'
tushar@tushar-laptop:~$ modinfo csc239.ko
filename:       csc239.ko
author:         TUSHAR DAVE
description:    CSc239 Presentation
license:        GPL
srcversion:     4E301C9577E6E12FBD838D1
depends:
vermagic:       2.6.20-16-generic SMP mod_unload 586
```

LKM generally uses function provided by the kernel. In the linux kernel, LKM user can only use those symbols (functions and data) exported by the kernel. The symbols exported by the kernel are in /proc/kallsyms. You can also use **nm** or **objdump** to see the objects your module is using.(for more detail do 'man nm'). In addition LKM users can also export symbols to the kernel by macro EXPORT_SYMBOL(your_SYMBOL) and EXPORT_SYMBOL_GPL(your_SYMBOL).See we export my_variable symbol in above code. You can see the symbol by

```
tushar@tushar-laptop:~$ cat /proc/kallsyms|grep my_variable
f8d90050 r __ksymtab_my_variable        [csc239]
f8d9005c r __kstrtab_my_variable        [csc239]
f8d90058 r __kcrctab_my_variable        [csc239]
f8d904b8 d my_variable  [csc239]
005c5795 a __crc_my_variable    [csc239]
```

(d signifies the variable is in defined data section)

Module dependencies :

“Good programmers are lazy. They don't try to reinvent the wheel in every program they write. Module programmers are not an exception. When they start to write a new module, they search documentation to know if there is a module already written that could help them in their challenges. This produces some dependencies between modules. For instance, if you want to write a driver for you new usb digital camera, you can write it from scratch, but it would be faster if you use the usb-core module to support the operations that your module performs. If you do that, your module will depend on the usb-core module [1]”

Loading and unloading modules when there is dependencies

command : “**modprobe** - same as insmod but handles dependencies ,modprobe intelligently adds or removes a module from the Linux”[4] kernel.

(For more information see 'man modprobe')

***lets hack the kernel using LKM,We are going to get the system call tables base address and then change sys_open function address to our_sys_open function address.**

****

//filename : syscallhack.c
/*system call base address can be found manually by

cat /boot/System.map-2.6.20-16-generic | grep sys_call_table

*/

```
#include <linux/module.h> //every module need this
#include <linux/kernel.h> // for printk
#include <linux/unistd.h> // for system call

#define LICENSE "GPL"
#define DESCRIPTION "CSc239 Presentation"
#define AUTHOR "TUSHAR DAVE"

MODULE_LICENSE(LICENSE);
MODULE_DESCRIPTION(DESCRIPTION);
MODULE_AUTHOR(AUTHOR);

unsigned int *sys_call_table; // pointer stores sys_call_table address
```

```
asmlinkage long (*original_sys_open)(const char *,int,int);



asmlinkage long our_sys_open(const char *filename,int flags,int mode)

{
        printk(KERN_INFO "SPY on  OPEN SYSTEM CALL \n");

        return original_sys_open(filename,flags,mode); /* call the original function */

}

int my_init(void)

    {

        printk(KERN_INFO "I am Hacking the system call\n");

        sys_call_table=0xc02f4500; /*assign the system call table base address to sys_call_table*/

        original_sys_open = sys_call_table[__NR_open]; /* Save the original sys_open address , so we can call the original sys_open function */

        sys_call_table[__NR_open] = (void *)our_sys_open; /* change the system call base table entry of sys_open to our_sys_open function */
        return 0;
    }

void my_exit(void)
    {


        if(sys_call_table[__NR_open] != our_sys_open) /* This check is must because if this is the case then somebody else also played with sys_call_table's sys_open */

            {

                printk(KERN_ALERT "Somebody else also played with system call table\n");

                printk(KERN_ALERT "The system may be left in unstable state\n");



            }
```

```
        sys_call_table[__NR_open] = (void *)original_sys_open;
 /*             assign the original address for system integrity */

        printk(KERN_INFO "Leaving the kernel\n");

    }

module_init(my_init);

module_exit(my_exit);
**********************************************************************
```

What should be your next step from here?

1. How to build the kernel module in source tree? (Hint : Linux Kernel Development by Robert Love)
2. Go to the reference website, there are more examples of LKM with detail explanation.
3. How to split the source code of module in different files and how to compile them?
4. Debugging utilities for the kernel
5. How to write Device Driver?
6. How to get system call table base address by writing code and not manually using System.map file?

Reference:

[1].http://tldp.org/LDP/lkmpg/2.6/html/
[2].http://www.linuxforums.org/programming/introducing_lkm_programming_part_i.html
[3].http://lxr.linux.no (Browse Linux Source Code Online)
[4].Linux MAN pages

Some good websites for Linux geeks:

1. There is an interactive kernel map available on web http://www.linuxdriver.co.il/kernel_map
2. You can also browse Linux kernel source code online http://lxr.linux.no
3. www.lwn.net
4. www.kerneltrap.org
5. www.kernel.org (official site for Linux kernel posting)