**Loading and unloading Kernel Modules**

### Writing the kernel module

### What is a module?

Module is a program that is dynamically linked to the kernel and runs in the kernel space.

The below code shows a simple “hello world” module

# include <linux/module.h>

#include <linux/init.h>

int my\_init(void) /\*ENTRY POINT FOR THE MODULE \*/

{

 printk("<1>Hello, world\n");

 return 0;

}

void my\_cleanup(void) ) /\*EXIT POINT FOR THE MODULE

{

 printk("<1>Goodbye world\n");

}

module\_init(my\_init); /\*macro specifying for your entry point

module\_exit(my\_cleanup); ); /\*macro specifying for your exit point

### How to compile a module:

Once you have everything set up, creating a makefile for your module is straightforward.

for the “hello world” example, a single line will suffice: EXAMPLE: Create a simple makefile by

root# vi Makefile

> obj-m := hello.o #inside the makefile type this and save your file

Execute the command given below on the shell to generate your object file with filename.ko extension

root# make -C /lib/modules/2.6<version>/build M=`pwd` modules

 The assignment above (which takes advantage of the extended syntax provided by GNU *make*) states that there is one module to be built from the object file *hello.o*. The resulting module is named *hello.ko* after being built from the object file.

If, instead, you have a module called *module.ko* that is generated from two source files (called, say, *file1.c* and *file2.c*), the correct incantation would be:

obj-m := module.o

module-objs := file1.o file2.o

For a makefile like those shown above to work, it must be invoked within the context of the larger kernel build system. If your kernel source tree is located in, say, your *~/kernel-version directory*, the *make* command required to build your module (typed in the directory containing the module source and makefile) would be:

make -C /lib/modules/<version>/build M=`pwd` modules

This command starts by changing its directory to the one provided with the – C option (that is, your kernel source directory). There it finds the kernel’s top-level makefile. The M= option causes that makefile to move back into your module source directory before trying to build the modules target. This target, in turn, refers to the list of modules found in the obj-m variable, which we’ve set to *module.o* in our examples. Typing the previous *make* command can get tiresome after a while, so the kernel developers have developed a sort of makefile idiom, which makes life easier for those building modules outside of the kernel tree. Write your makefile as follows:

### Writing a complete makefile to compile a module:

**#vi Makefile**

obj-m := hello.o

KERNELDIR? = /lib/modules/$(shell uname -r)/build

PWD := $(shell pwd)

default:

$(MAKE) -C $(KERNELDIR) M=$(PWD) modules

The printk function is defined in the Linux kernel and behaves similarly to the standard C library function printf. As memtioned earlier kernel does not have access to c library and kernel needs its own printing. The module can call printk because, after insmod has loaded it, the module is linked to the kernel and kernel libraries which is needed by printk. The string <1> is the priority of the message. ( printk discussed later in debugging).

### How to insert module in kernel space:

You can insert the module in kernel space by calling insmod (loading the module) and rmmod (unloading the module). Note that only the superuser can load and unload a module. This mechanism of inserting a module in the running kernel is called “dynamic loading”.

After compiling the module as memtioned earlier insert the module in kernel space.

root# **insmod hello.ko**

### Removing the module from kernel space :

root# **rmmod hello**

### Listing the modules:

root# **lsmod**

The lsmod program produces a list of the modules currently loaded in the kernel.

Some other information, such as any other modules making use of a specific module,

is also provided. lsmod works by reading the /proc/modules virtual file. Information

on currently loaded modules can also be found in the sysfs virtual filesystem

under /sys/module.

### Kernel Modules versus Applications

* An application program performs a single task from beginning to end, a module registers itself in order to serve future requests, and its "main" function terminates immediately.
* In other words, the task of the function *init\_module* (the module's entry point) is to prepare for later invocation of the module's functions; it's as though the module were saying, "Here I am, and this is what I can do."
* The second entry point of a module, *cleanup\_module*, gets invoked just before the module is unloaded. It should tell the kernel, "I'm not there anymore; don't ask me to do anything else." The ability to unload a module is one of the features of modularization that you'll most appreciate, because it helps cut down development time; you can test successive versions of your new driver without going through the lengthy shutdown/reboot cycle each time.
* An application program uses the library function, the linking stage resolves external references using the appropriate library of functions. *printf* is one of those callable functions and is defined in *libc*.
* A module, on the other hand, is linked only to the kernel, and the only functions it can call are the ones exported by the kernel; there are no libraries to link to. The *printk* function used in *hello.c* earlier, for example, is the version of *printf* defined within the kernel and exported to modules. It behaves similarly to the original function, with a few minor differences, the main one being lack of floating-point support.

Figure 2-1 shows how function calls and function pointers are used in a module to add new functionality to a running kernel.



**Figure 2-1. Linking a module to the kernel**

Because no library is linked to modules, source files should *never* include the usual header files. Only functions that are actually part of the kernel itself may be used in kernel modules. Anything related to the kernel is declared in headers found in *include/linux* and *include/asm* inside the kernel sources (usually found in */usr/src/linux*). Older distributions (based on *libc* version 5 or earlier) used to carry symbolic links from */usr/include/linux*and */usr/include/asm* to the actual kernel sources, so your *libc* include tree could refer to the headers of the actual kernel source you had installed. These symbolic links made it convenient for user-space applications to include kernel header files, which they occasionally need to do.

### Name Space Pollution:

Namespace pollution is what happens when there are many functions and global variables whose names aren't meaningful enough to be easily distinguished.

Developers can't afford to fall into such an error when writing kernel code because even the smallest module will be linked to the whole kernel. The best approach for preventing namespace pollution is to declare all your symbols as static and to use a prefix that is unique within the kernel for the symbols you leave global. Also note that you, as a module writer, can control the external visibility of your symbols.

Most versions of *insmod* (but not all of them) export all non-static symbols if they find no specific instruction in the module; that's why it's wise to declare as static all the symbols you are not willing to export.

### The kernel symbol table:

The table contains the addresses of global kernel items functions and variables that are needed to implement modularized drivers. The public symbol table can be read in text form from the file.

**Command to see kernel symbols:**

 root# cat */proc/ ksyms or root# ksyms*

### The Usage Count

The system keeps a usage count for every module in order to determine whether the module can be safely removed. The system needs this information because a module can't be unloaded if it is busy: you can't remove a filesystem type while. the filesystem is mounted, and you can't drop a char device while a process is using it, or you'll experience some sort of segmentation fault or kernel panic when wild pointers get dereferenced. In modern kernels, the system can automatically track the usage count for you.

### Passing Parameters to modules:

Several parameters that a driver needs to know can change from system to system. These parameter values can be assigned at load time by *insmod* or *modprobe(*will be discussed shortly*)*; the latter can also read parameter assignment from its configuration file (*/etc/modprobe.*  *conf*). The commands accept the specification of several types of values on the command line. As a way of demonstrating this capability, imagine a much-needed enhancement to the “hello world” module (called *hellop*) shown at the beginning of this chapter. We add two parameters: an integer value called howmany and a character string called whom. Such a module could then be loaded with a command line such as:

insmod hellop howmany=10 whom="Mom"

insmod hellop howmany=10 // when loaded with fewer parameters the, the module works with the values provided in the code.

Upon being loaded that way, *hellop* would say “Hello, Mom” 10 times. However, before *insmod* can

change module parameters, the module must make them available. Parameters are declared with the module\_param macro, which is defined in *moduleparam.h*. module \_ param takes three parameters: the name of the variable, its type, and a permissions mask to be used for an accompanying sysfs entry. The macro should be placed outside of any function and is typically found near the head of the source file. So *hellop* would declare its parameters and make them available to *insmod* as follows:

static char \*whom = "world";

static int howmany = 1;

module\_param (howmany, int, S\_IRUGO);

module\_param (whom, charp, S\_IRUGO);

Numerous types are supported for module parameters:

bool, charp, int , long, short, uint, ulong, ushort.

A Boolean (true or false) value (the associated variable should be of type int). A char pointer value . Basic integer values of various lengths. The versions starting with u are for unsigned values. Memory is allocated for user-provided strings, and the pointer is set accordingly.

 Array parameters, where the values are supplied as a comma-separated list, are also supported by the module loader. To declare an array parameter, use:

module\_param\_array (name,type,num,perm);

Where name is the name of your array (and of the parameter), type is the type of the array elements, num is an integer variable, and perm is the usual permissions value. If the array parameter is set at load time, num is set to the number of values supplied.

The module loader refuses to accept more values than will fit in the array. If you really need a type that does not appear in the list above, there are hooks in the module code that allow you to define them; see *moduleparam.h* for details on how to do that. All module parameters should be given a default value; *insmod* changes the value only if explicitly told to by the user. The module can check for explicit parameters by testing parameters against their default values. The final *module\_param* field is a permission value; you should use the definitions found in *<linux/stat.h>*. This value controls who can access the representation of the module parameter in sysfs. If perm is set to 0, there is no sysfs entry at all; otherwise, it appears under */sys/module*\* with the given set of permissions. Use S\_IRUGO for a parameter that can be read by the world but cannot be changed; S\_IRUGO|S\_IWUSR allows root to change the parameter. Note that if a parameter is changed by sysfs, the value of that parameter as seen by your module changes, but your module is not notified in any other way. You should probably not make module parameters writable, unless you are prepared to detect the change and react accordingly

### Macros for Documentation:

**MODULE\_AUTHOR (name)**

Puts the author's name into the object file

**MODULE\_DESCRIPTION (desc)**

Puts a description of the module into the object file

**MODULE\_SUPPORTED\_DEVICE (dev)**

Places an entry describing what device is supported by this module. Comments in the kernel source suggest that this parameter may eventually be used to help with automated module loading, but no such use is made at this time.