Device Driver Training
Session 1

Contents

- Understanding kernel modules
- Advantages of kernel modules
- Writing a simple kernel module
- Compiling the kernel module
- Module Utilities
- Module dependencies
- Modules with multiple files
- Kernel space Vs User space
- Application Vs Kernel module

Kernel Modules

- Linux kernel has the ability to extend at runtime the set of features offered by the kernel
- This means that you can add functionality to the kernel while the system is up and running
- Each piece of code that can be loaded and unloaded into the kernel at runtime is called a ***module***
  - Which are pluggable to the operating system that adds functionality
  - Can be added on the fly
  - Allow us to change functionality on the fly
  - Allow us to read and write
  - Extends the functionality of the kernel without the need to reboot the system

Kernel Modules cont..

- Each module is made up of object code that can be dynamically linked to the running kernel
- Modules are pieces of code that can be loaded and unloaded into the kernel upon demand
- One type of module is the device driver, which allows the kernel to access hardware connected to the system
- Two types of interfaces are available to the designer to use the module :
    - Device driver and /proc
    - Generally /proc interface is for reading and device driver interface is for writing

Advantages of kernel module

- Modules make it easy to develop drivers without rebooting: load, test, unload, rebuild & again load and so on
- Useful to keep the kernel size to the minimum (essential in embedded systems)
- Without modules , would need to build a kernel every time in order to add new functionality directly into the kernel image
- Also useful to reduce boot time, you don't need to spend time initializing device that may not be needed at boot time
- Once loaded, modules have full control and privileges in the system
- That's why only the root user can load and unload the modules

Writing Hello module

```
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kernel.h>

static int hello_init(void) {
        printk(KERN_INFO “Hello :This is my first kernle module\n");
        return 0;
}
static void hello_exit(void) {
        printk(KERN_INFO “Bye, unloading the module\n");
}
module_init(hello_init);
module_exit(hello_exit);
MODULE_DESCRIPTION("Greeting module");
MODULE_AUTHOR(“eltra");
MODULE_LICENSE("GPL");
```

Writing Hello module cont..

- Headers specific to the linux kernel <linux/xxx.h>
  - No access to the usual C library
- An initialization function
  - Called when the module is loaded
  - returns an error code (0- success, negative value on failure)
  - Declared by the module_init() macro
- A cleanup function
  - Called when the module is unloaded
  - Declared by the module_exit() macro
- Declare init as "static" => file-limited scope

Writing Hello module cont..

- Metadata information declared used
  - MODULE_DESCRIPTION
    - Human readable statement of what the module does
  - MODULE_AUTHOR
    - Stating who wrote the module
  - MODULE_LICENSE
    - Generally GPL – General Public License
    - The GPL allows anybody to redistribute, and even sell, a product covered by the GPL, as long as the recipient has access to the source and is able to exercise the same rights
    - The main goal of such a license is to allow the growth of knowledge by permitting everybody to modify programs at will
  - MODULE_VERSION
    - For a code revision number

Writing Hello module cont..

- Printk
  - The server can’t use stdlib due to userspace/kernel space issues
  - Most of C library is implemented in the kernel
  - printk is printf for kernel programs
  - One of the differences is that *printk* lets you classify messages according to their severity by associating different *loglevels*, or priorities, with the messages
  - There are eight possible loglevel strings, defined in the header *<linux/kernel.h>*

Writing Hello module cont..

- KERN_EMERG
  - Used for emergency messages, usually those that precede a crash
- KERN_ALERT
  - A situation requiring immediate action
- KERN_CRIT
  - Critical conditions, often related to serious hardware or software failures
- KERN_ERR
  - Used to report error conditions;
  - device drivers often use KERN_ERR to report hardware difficulties

Writing Hello module cont..

- KERN_WARNING
  - Warnings about problematic situations that do not, in themselves, create serious problems with the system
- KERN_NOTICE
  - Situations that are normal, but still worthy of note. A number of security-related conditions are reported at this level
- KERN_INFO
  - Informational messages
  - Many drivers print information about the hardware they find at startup time at this level
- KERN_DEBUG
  - Used for debugging messages

Writing Hello module cont..

- Kernel Log
  - When a new module is loaded, related information is available in the kernel log
  - The kernel keeps its messages in a circular buffer
  - Kernel log messages are available through the ‘dmesg’ command
- Module symbols
  - When a module is loaded, any symbol exported by the module becomes part of the kernel symbol table
  - In the usual case, a module implements its own functionality without the need to export any symbols at all
  - You need to export symbols, however, whenever other modules may benefit from using them
  - New modules can use symbols exported by your module
  - To export a symbol - EXPORT_SYMBOL(name);

Compiling a module

- Out of tree
  - When the code is outside of the kernel source tree, in a different directory
  - Advantage: Easier to handle than modifications to the kernel itself
  - Disadvantage:
    - Not integrated to the kernel configuration/compilation process
    - Needs to be build separately
    - Driver cannot be built statistically if needed
- Inside the kernel tree
  - Well integrated into the kernel configuration/compilation process
  - Driver can be build statistically if needed

Makefile

```
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
```

- obj –m
  - States that there is one module to be built from the object file hello1.o
  - The resulting module is named *hello1.ko* after being built from the object file

Makefile

- Make command
  - -C option
    - This command starts by changing its directory to the one provided with the -C option that is, your kernel source directory
  - 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 *hello1.o* in our example

Module utilities

- insmod <module_name>.ko
  - Load the given module
  - Full path of module is needed
- rmmod <module_name>
  - Unloads the given module
- lsmod
  - Displays the list of modules loaded
  - Check cat /proc/modules
- modinfo <module_name>
  - Gets information about the module: parameters, license, descriptions and dependencies

Module utilities

- insmod <module_name>.ko
  - Load the given module
  - Full path of module is needed
- rmmod <module_name>
  - Unloads the given module
- lsmod
  - Displays the list of modules loaded
  - Check cat /proc/modules
- modinfo <module_name>
  - Gets information about the module: parameters, license, descriptions and dependencies

Module dependencies

- Some kernel module can depend on other modules, which need to be loaded first
- Dependencies are described in /lib/modules/<kernel-version>/modules.dep
- sudo modprobe <module_name>
    - Loads all the modules the given module depends on
    - Modprobe looks into /lib/modules/<kernel-version> for the object file corresponding to the given module
- Sudo modprobe –r <module_name>
    - Remove the module and all dependent modules, which are no longer needed

Hello2 sample

```
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kernel.h>

static int __init hello2_init(void) {
	printk(KERN_INFO “Hello :This is my first kernle module\n");
	return 0;
}
static void __exit hello2_exit(void) {
	printk(KERN_INFO “Bye, unloading the module\n");
}
module_init(hello2_init);
module_exit(hello2_exit);
```

__init and __exit macro

- __init
    - it is a hint to the kernel that the given function is used only at initialization time
    - The module loader drops the initialization function after the module is loaded, making its memory available for other uses
- __exit to specify that cleanup is for unloading only

Hello3 sample

```
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kernel.h>
static int hello_3_data _initdata=5;
static int __init hello3_init(void) {
        printk(KERN_INFO “Hello :This is my first kernle module\n");
        return 0;
}
static void __exit hello3_exit(void) {
        printk(KERN_INFO “Bye, unloading the module\n");
}
module_init(hello3_init);
module_exit(hello3_exit);
```

- __initdata is same as __init but used for variable declaration used for initialization purpose only

Hello4 sample

```
#define DRIVER_AUTHOR ”neepa@eitra.org>"
#define DRIVER_DESC "A sample driver"

static int __init init_hello_4(void) {
        printk(KERN_INFO "Hello, world 4\n");
        return 0;
}
static void __exit cleanup_hello_4(void) {
        printk(KERN_INFO "Goodbye, world 4\n");
}
module_init(init_hello_4);
module_exit(cleanup_hello_4);
MODULE_LICENSE("GPL"); /*We added licensing and information that removes the “kernel is tainted message” */
MODULE_AUTHOR(DRIVER_AUTHOR); /* Who wrote this module? */
MODULE_DESCRIPTION(DRIVER_DESC); /* What does this module do */
```

Module Parameters

- Several parameters that a driver needs to know can change from system to system
- These can vary from the device number to use to numerous aspects of how the driver should operate
- They will needed to specify some parameters at load time
- Specified at load time:
  - Insmod and modprobe (/etc/modprobe.conf)
- 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(name, data type, permissions)
- Provide description of module parameter with
  - MODULE_PARAM_DESC(name, “desc”);

Module Parameters cont..

- Permission are as specified in <linux/stat.h>:
  - S_IRUGO => read-only for world
  - S_IWUSR => user-only write
  - S_IWGRP => Group write
  - S_IWOTH => Others write
- Data Types
  - Int
  - Short
  - Long
  - Bool
  - String
  - To pass parameter array
    - module_param_array(name,type,nump,perm);
    - Values are supplied as a comma-separated list

Hello5 sample

```
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kernel.h>
static int myint = 420;
module_param(myint, int, S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH);

static int __init hello_5_init(void)
{
        int i;
        printk(KERN_INFO "Hello, world 5\n=============\n");
        printk(KERN_INFO "myint is an integer: %d\n", myint);
        return 0;
}
static void __exit hello5_exit(void) {
        printk(KERN_INFO "Bye, unloading the module\n");
}

# insod hello5.ko myint=123
```

Multiple files

- Modules can be build having multiple source files
  - start.c
  - stop.c

- Module Makefile having multiple source file

```
obj-m += startstop.o
startstop-objs := start.o stop.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
```

Multiple files cont..

- Start.c

```
#include <linux/kernel.h>
#include <linux/module.h>
int init_module(void) {
        printk(KERN_INFO "Hello, world – this is kernel speaking\n");
        return 0;
}
```

- Stop.c

```
#include <linux/kernel.h>
#include <linux/module.h>

void cleanup_module(void) {
        printk(KERN_INFO ”Good Bye\n");

}
```

User space Vs Kernel space

- User space Vs Kernel space
  - A module runs in *kernel space*, whereas applications run in *user space*
  - This concept is at the base of operating systems theory
  - The role of the operating system, in practice, is to provide programs with a consistent view of the computer's hardware
  - In addition, the operating system must account for independent operation of programs and protection against unauthorized access to resources
  - This non trivial task is possible only if the CPU enforces protection of system software from the applications
  - Every modern processor is able to enforce this behavior
  - The chosen approach is to implement different operating levels in the CPU itself
  - The levels have different roles, and some operations are disallowed at the lower levels

User space Vs Kernel space cont..

- Program code can switch from one level to another only through a limited number of gates
- Unix systems are designed to take advantage of this hardware feature, using two such levels
- All current processors have at least two protection level
- The kernel executes in the highest level (also called *supervisor mode*), where everything is allowed
- Applications execute in the lowest level (the so-called *user mode*), where the processor regulates direct access to hardware and unauthorized access to memory
- Refer these execution modes as *kernel space* and *user space*

User space Vs Kernel space cont..

- These terms encompass not only the different privilege levels inherent in the two modes, but also the fact that each mode can have its own memory mapping—its own address space—as well
- Unix transfers execution from user space to kernel space whenever an application issues a system call or is suspended by a hardware interrupt
- Kernel code executing a system call is working in the context of a process —it operates on behalf of the calling process and is able to access data in the process's address space
- Code that handles interrupts, on the other hand, is asynchronous with respect to processes and is not related to any particular process
- The role of a module is to extend kernel functionality; modularized code runs in kernel space
- Usually a driver performs both the tasks outlined previously: some functions in the module are executed as part of system calls, and some are in charge of interrupt handling

Application Vs Kernel Module

- Application
  - Performs single task from beginning to end
  - Application can call functions, which it doesn't define
  - The linking stage resolves the external references loading the appropriate libraries. E.g libc for 'printf' function

- Kernel module
  - Module registers itself to serve the future request and its 'main' function terminates on loading
  - The module is linked only to the kernel and it can refer only the functions that are exported by the kernel
  - No C library is linked with the kernel

Thank you