The Linux Kernel Module Programming

The Linux Kernel Module Programming Guide

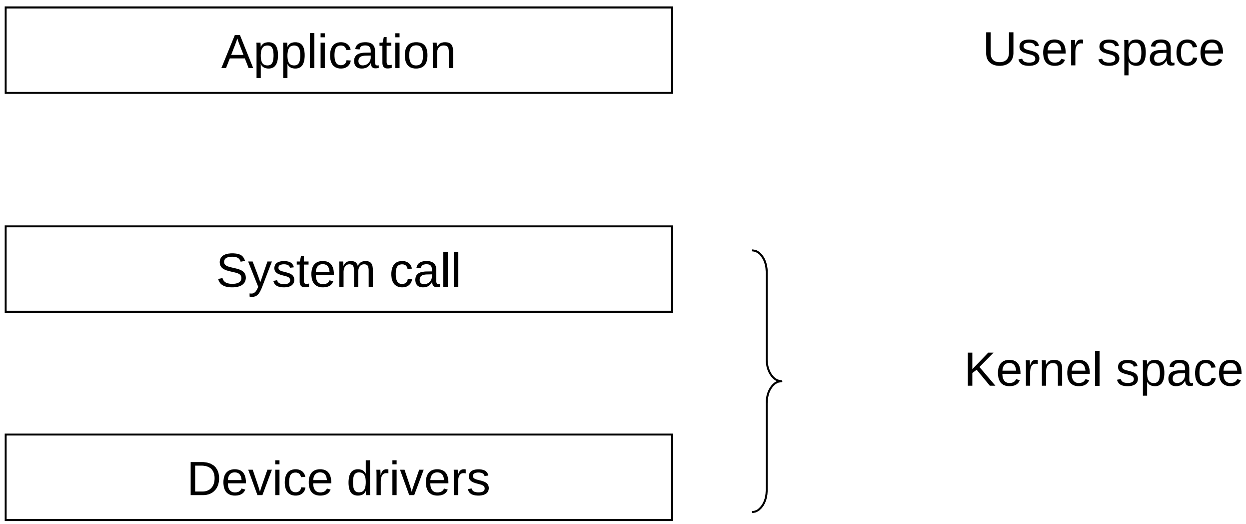
by

P.J. Saltzman, M. Burian, Ori Pomerantz

http://tldp.org/LDP/lkmpg/2.6/html/index.html

http://www.faqs.org/docs/kernel/x1206.html

IO architecture



Kernel programming

- No libc functions!!!
  - No **`printf`**, use **`printk`** instead
  - Some common functions implemented inside the kernel
- No memory protection
- Small, fixed size stack
- Synchronization and concurrency issues

Module commands

- Installed modules

```
lsmod
```

- Install a module

```
insmod module.ko
modprobe module
```

- Module dependency

```
depmod module.ko
```

Modules

hello-1.c

`printk(KERN_INFO ”format string”, variables)`

- **`init_module`**
- **`cleanup_module`**

Makefile

`insmod hello-1.ko`

Output goes in /var/log/messages

`dmesg |tail`

`rmmod hello-1`

Priority level

```
#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
```

Modules

hello-2.c

- **module_init   module_exit**

Macros

- **__init function_name**
- **__exit function_name**

hints to the kernel that the given function is used only once, the kernel module loader then release their memory (for built-in drivers only)

Makefile

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

obj-m += hello-2.o
```

Module documentation

hello-4.c

- **`Demonstrates module documentation`**

Module information displayed using

`modinfo hello-4.ko`

Passing command line arguments

hello-5.c

- `insmod hello-5.ko \`
  `mystring="lafax" myshort=123 myintArray=-1,3`
  `Then look at /sys/module/hello-5/parameters`
- `module_param(name, type, permissions)`
  - `Name : myshort, myint, mylong, mystring,`
  - `Type: short, int, long, charp`
  - `Permissions in /sys/module/    (sysfs)`
- `module_param_array(name, type, num, permissions)`
  - Example:

```
module_param_array(myintArray,int,&arr_argc,0550)
```

Modules spanning multiple files

start.c and stop.c

- Makefile

```
obj-m += startstop.o

startstop-objs := start.o stop.o
```

Special files for devices

- Major and minor numbers

```
more /usr/src/linux/Documentation/devices.txt
```

- Create a special file

```
mknod /dev/aaaa c 12 2
```

- Kernel uses the major number to determine which driver will be used
- The driver (not the kernel) deals with the minor number to distinguish between different devices of the same type.

Character device files

- `struct file_operations` in /usr/src/linux/include/linux/fs.h

- C99 way to assign functions to the fields of this structure

```
struct file_operations fops =
{
  .read = device_read,
  .write = device_write,
  .open = device_open,
  .release = device_release
};
```

The file structure

- `struct file filp` in /usr/src/linux/include/linux/fs.h
- Is not `struct FILE`

Registering a device driver

- Means assigning a major and minor number during the initialization of a module

What major number?

- See /usr/src/linux/Documentation/devices.txt
- Ask the kernel to assign a dynamic major number
  `alloc_chrdev_region( &dev_no, firstminor, num_minors, DEVICE_NAME)`

Dynamic major number

- If we ask the kernel to assign a new major number how we get the major number to create the device file?
  - Look at the end of file /var/log/messages
  - The driver prints the new assigned major number, and we create the device file by hand
  - Use the entry created in **`/proc/devices`** to get the major number and create the device file by hand or using a shell script

Unregistering a device

- A kernel module cannot be removed if a device driver is opened by a process because a successive system call would be directed to a memory location where the appropriate function does not exist any more.
- The cleanup_module has void type, thus it cannot return -1
- The third field in /proc/modules is a counter of the number of processes that are using a module
- The counter is managed by means of
  - **`try_module_get(THIS_MODULE)`** to increment the count.
  - **`module_put(THIS_MODULE)`** to decrement the count.

chardev_SDP_1

- **`cat`** (or open within a program) **`/dev/chardev_SDP-1`**
  - Reads data and print an acknowledge message with the number of times the device file has been read.

- **`echo something > /dev/chardev_SDP-1`**
  - Writing is not supported, but it is logged

Moving data between user and kernel space

- The **`put_user`** and **`get_user`** macros
  - **`put_user (k, u)`** copies one character from the kernel data segment to the user data segment
  - **`get_user (k, u)`** copies one character from the user data segment to the kernel data segment

  - Put and get seen from the kernel point of view

chardev_SDP_2

- **`cat`** or **`open`** with a program **`/dev/chardev_SDP-2`**
  - Reads data and print an acknowledge message with the number of times the device file has been read.

- **`echo something > /dev/chardev_SDP-2`**
- **`cat /dev/chardev_SDP-2`**
  - **`"something" will be read from chardev_SDP-2`**

chardev_SDP_2

- **`cat`** or **`open`** with a program **`/dev/chardev_SDP-2`**
  - Reads data and print an acknowledge message with the number of times the device file has been read.

- **`echo something > /dev/chardev_SDP-2`**
- **`cat /dev/chardev_SDP-2`**
  - **`"something" will be read from chardev_SDP-2`**

chardev_SDP_lab

- **`read`** and **`write`** with
  - **`copy_to_user`**
  - **`copy_from_user`**

/proc filesystem

- The **`/proc`** filesystem is an additional mechanism for the kernel and its modules to give information to processes. Ex.
  - **`/proc/modules`**
  - **`/proc/meminfo`**

procfs-3

Allows using the file system structures

Including permissions

- `cat /proc/buffer2k`
- `ls > /proc/buffer2k`
- `cat /proc/buffer2k`

ioctl

- The **ioctl** function is called with three parameters:
  - the file descriptor of the appropriate device file,
  - the ioctl number,
  - a parameter
- The **ioctl** number encodes the major device number, the type of the **ioctl**, the command, and the type of the parameter.
- This **ioctl** number is usually created by a macro call in a header file **(_IO, _IOR, _IOW or _IOWR** --- depending on the type). See **/usr/include/asm-generic/ioctl.h**
- If you want to use **ioctl** in your own kernel modules, it is best to receive an official **ioctl** assignment.

chardev-2.c and ioctl.c

- See the chardev.h file
- The device driver manage system calls
  - read (cat) from char_dev
  - write (echo … > char_dev
  - 3 types of ioctl for
    - setting a message
    - reading the n-th char of the message
    - getting the content of the device

Blocking process example

- The file **`/proc/sleep`** can only be opened by a single process at a time.
- If the file is already open, the kernel module calls

`wait_event_interruptible`

  - This function changes the status of the task to **`TASK_INTERRUPTIBLE`**
  - adds it to **`WaitQ`**, the queue of tasks waiting to access the file
  - calls the scheduler for context switching

Blocking process example

- When a process closes, the file **`module_close`** is called.
- This function wakes up all the processes in the queue.
- A previously waiting process starts at the point right after the call to **`module_interruptible_sleep_on`**.
- It sets a global variable to tell all the other processes that the file is still open and go on with its life.
- When the other processes are scheduled they see that the global variable is set and go back to sleep.

Blocking process examples

```
insmod sleep.ko

ls *.o > /proc/sleep
tail -f  /proc/sleep & cat /proc/sleep  &
kill %1
```

Use of **`O_NONBLOCK`** flag in opening **`/proc/sleep`**

```
cat_noblock /proc/sleep
tail -f  /proc/sleep &
cat_noblock /proc/sleep
kill %1
```

.

Workqueue interface

- **Workqueues are created using:**

```
struct workqueue_struct *create_workqueue(const char *name);
```

where **name** is the name of the workqueue.

- The task needs to be packaged into a structure called the **work_struct** structure.
- **A workqueue task can be initialized at compile time using**

```
DECLARE_WORK(name, void (*function)(void *), void *data);
```

where **name** is the **work_struct** name, **function** is the function to be invoked when the task is scheduled, and **data** is the pointer to that function.

- **A workqueue task can be initialized at run time by using:**

```
INIT_WORK(struct work_struct *work, void (*function)(void *), void *data);
```

Queuing

- **Queuing** a job into the workqueue is done with the following function calls:

```
int queue_work(struct workqueue_struct *queue,
            struct work_struct *work);

int queue_delayed_work(struct workqueue_struct
 *queue, struct work_struct *work, unsigned long
                      delay);
```

- The **`delay`** in **`queue_delayed_work()`** is given to ensure that at least a minimum delay in **`jiffies`** is given before the workqueue entry actually starts executing.

Cancel, flush, destroy

- Entries in the workqueue are executed by the associated worker thread at an unspecified time (depending on load, interrupts, etc.) or after a delay time. Any workqueue entry that takes an inordinately long time to run can be **cancelled** with:

```
int cancel_delayed_work(struct work_struct *work);
```

  - If the entry is actually executing when the cancellation call returns, the entry continues to execute, but will not be added to the queue again.

- The workqueue is **flushed of entries** with:

```
void flush_workqueue(struct workqueue_struct
                     *queue);
```

- and is **destroyed** with:

```
void destroy_workqueue(struct workqueue_struct
                       *queue);
```

Schedule work in the global queue

- It is not necessary for all drivers to have their own custom workqueue. Drivers can use the default workqueue provided by the kernel. Since this workqueue is shared by many drivers, the entries may take a long time to execute. To alleviate this, delays in worker functions should be kept to a minimum or avoided.
- An important note is that the **default queue is available to all drivers**, but only GPL licensed drivers can use their own custom-defined workqueues:

`int schedule_work(struct work_struct *work);` // add an entry to the workqueue

`int schedule_delayed_work(struct work_struct *work, unsigned long delay);` // add a workqueue entry and delay execution

- There is also a **`flush_scheduled_work()`** function that waits for everything in the queue to be executed, and that should be called by modules upon unloading.

sched.c

- AT every timer interrupt increments a counter
- Reading **`/proc/sched`** shows the content of the counter
- 100 jiffies delay -> 25 interrupts every 10 seconds
- 50 jiffies delay -> 50 interrupts every 10 seconds

```
while true; do cat /proc/sched;sleep 10; done
```