Linux Device Drivers
Modules

- A piece of code that can be added to the kernel at runtime is called a “module”
- A device driver is one kind of module
- Each module is made up of object code that can be dynamically linked to the running kernel
  - Dynamic linking done using *insmod* program
  - Unlinking done using *rmmod* program
- Keep kernel small
Character Devices

• Char device drivers
  – stream of bytes (sequential access)
  – open, close, read, write
  – E.g. console, serial ports

• Block device drivers
  – buffering
Character Device Drivers

• Char devices are accessed through nodes of the filesystem tree located in the /dev directory

• Special files for char drivers are identified by “c” in the first character of the ls -l listing in /dev

• crw--w---- 1 root tty 4, ... tty40
Example Character Device

• Scull:
  – Simple Character Utility for Loading Localities (scull)
  – A memory based device
  – Does not connect to any real device
Character Device Driver: Scull

Scull devices are persistent; can be shared
Device Numbers

• Major number
  – Identifies the driver associated with the device
  – Available in: /proc/devices

• Minor number
  – Used by the Kernel to determine exactly which device
    is being referred to

• Idea: many devices can share the same driver
  – e.g. many terminals might share the same driver
Device Numbers

• dev_t type
  – Used to hold device numbers
  – Major and minor parts
  – 32 bit (12 bits for major number, 20 bits for minor number)

• Macros
  – To obtain the major or minor parts of a dev_t
    • MAJOR(dev_t dev);
    • MINOR(dev_t dev);
  – To convert major and minor numbers into dev_t
    • MKDEV(int major, int minor);
int register_chrdev_region (dev_t first, unsigned int count, char *name)

– first: beginning device number of the range you would like to allocate
– count: total device numbers (minor) you are requesting (will be 1 for us)
– name: name of the device that should be associated with this range
Device Major Number: Dynamic Allocation **

int alloc_chrdev_region (dev_t *dev, unsigned int firstminor, unsigned int count, char *name)

– dev: output parameter; on successful completion, holds the first number in your allocated range

– firstminor: requested first minor number to use; usually 0

– count: total number of contiguous device numbers (minor) you are requesting

– name: name of the device that should be associated with this number range
Example of Device Number Allocation

extern int scull_major; // auto allocation => 0
extern int scull_minor; // assume this is 0

if (scull_major) {
    dev = MKDEV(scull_major, scull_minor);
    result = register_chrdev_region (dev, scull_nr_devs, "scull");
}
else {
    result = alloc_chrdev_region (&dev, scull_minor, 
                                scull_nr_devs, "scull");
    scull_major = MAJOR(dev);
}
Device Driver Life-cycle

• Stage 1: Registration and Initialization
  – `module_init` (called when `insmod` is invoked)

• Stage 2: Serving requests from user-space programs
  – `open`, `read`, `write`, `close`, `lseek`

• Stage 3: De-registration and clean-up
  – `module_exit` (called when `rmmod` is invoked)

Hello World
#include <linux/init.h>
#include <linux/module.h>
static char *charArg = "foo";
static int intArg = 25;

/* declare that intArg and charArg are args to the module and list their types and permissions */
module_param (intArg, int, S_IRUGO);
module_param (charArg, charp, S_IRUGO);

/* module initialize function */
static int hello_init(void)
{
    printk(KERN_INFO "HelloWorld: You passed: %d and %s\n", intArg, charArg);
}

/* module remove function */
static void hello_exit(void)
{
    printk(KERN_INFO "HelloWorld: So long and thanks for all the fish..\n");
}

/* specify the module init and remove functions */
module_init(hello_init);
module_exit(hello_exit);

root# insmod ./hello.ko HelloWorld: You passed 25 and foo
root# rmmod hello HelloWorld: So long and thanks for all the fish..
Important Data Structures

- **struct file**
  - This structure is created every time a file/dev is opened. It is maintained while the file is open

- **struct inode**
  - An inode is maintained for each file/dev; contains pointers to the device structure (cdev)

- **struct cdev**
  - the char device; contains a pointer to the file operations structure

- **struct file_operations**
  - contains pointers to functions for device interface functions

- **struct your_device**
  - contains state, storage, ... and cdev
struct file_operations scull_fops = {
  .llseek = scull_llseek,
  .read = scull_read,
  .write = scull_write,
  .ioctl = scull_ioctl,
  .open = scull_open,
  .release = scull_release,
}

User code:
fd = open("/dev/scull0", ...);
read (fd, ...);
...

A structure `struct file` contains:

- Some important fields: open file
  - `struct file_operations *fops` — The operations associated with the file
  - `void *private_data` (~ device-specific data) — Useful resource for preserving state information across system calls
Scull Device

struct scull_dev {  // up to you (i.e.. struct your_device)
    ... data, bookkeeping, buffers, ...  
struct semaphore sem;
struct cdev cdev;
}

• **struct cdev** is Kernel’s internal structure that represents char devices

• The scull device driver needs to initialize this structure, initialize the cdev structure and register cdev with the Kernel
struct inode

• Passed to open function

• Some important fields
  – dev_t i_rdev
    • For inodes of device files, this field contains the actual device number
  – struct cdev *i_cdev
    • struct cdev is Kernel’s internal structure that represents char devices

  – container_of: from i_cdev => *struct your_device
Char Device Registration

• Kernel uses structures of type `struct cdev` to represent char devices internally

• Before Kernel can invoke device’s operations, we must do the following
  – 1. Set the `file_operations` pointer inside this structure
  – 2. Allocate and register one or more such structures
Char Device Registration

```c
void cdev_init (struct cdev *cdev, struct file_operations *fops)

int cdev_add (struct cdev *dev, dev_t num, unsigned int count);

count: #of device numbers (usually, this is 1)

Device is now “live”
```
Status after Char Device Registration

Kernel Space

User Space

File System

struct inode

Scull Device Driver

cdev_add

scull_dev

cdev
Handling of open call

Kernel Space

User Space

open

File System

struct inode

open

scull_open (struct inode *inode, struct file *filp)

Scull Device Driver

scull_dev
cdev
Conceptual View

Process A

after registration

scull module
scull_devices

inode
inode
inode

cdev
cdev
cdev

Buffer
Mutex
Semaphores
Buffer
Mutex
Semaphores
Buffer
Mutex
Semaphores

cdev

cdev

cdev

file_operations

file

open

sets
private_data

after registration
Open and release

• open (*inode, *filp)
  – setup filp->private_data for subsequent methods
  – device-specific initialization

• release (*inode, *filp) // close
  – device-specific dealloc / release resources
Read and write

- \texttt{read} (*filp, *buff, count, *offp)
- \texttt{write} (*filp, *buff, count, *offp)

- returns: <0 on error; >= 0 is bytes transferred
- buff – user space pointer

- \texttt{copy\_to\_user} (toAddr, fromAddr)
- \texttt{copy\_from\_user} (toAddr, fromAddr)
Closer look at read ...

```c
ssize_t dev_read(struct file *file, char *buf, size_t count, loff_t *pos);
```
Allocating memory in the kernel

- `kmalloc (size, GFP_KERNEL)`
  - similar to `malloc`
  - memory is not cleared
- `kfree (memPtr)`
- allocate buffers within your device
Synchronization

• Block processes calling your device

• Semaphores
  – sema_init (*sem, val)
  – down (*sem), down_interruptible, down_trylock
  – up (*sem)

• WaitQueues
  – init_waitqueue_head()
  – wait_event(), wait_event_interruptible() ...
  – wake_up(), wake_up_interruptible()