Linux Device Drivers & Project3 preview

CSC345
Project 3 Preview

• Write a device driver for a pseudo stack device
• Idea from http://www.cs.swarthmore.edu/~newhall/cs45/f01/proj5.html
• Linux character device type supports the following operations
  – Open: only one is allowed.
  – Write: writes an char string to top of the device stack. Error if stack is empty
  – Read: reads an item from top of the device stack. Error if stack is empty
  – Release: release the device
• Install with LKM.
• Test: It will be a dedicated standalone machine in the lab. Root password may be given out. If you mess up, you will re-install the
User program & Kernel interface

Note: This picture is excerpted from Write a Linux Hardware Device Driver, Andrew O'Shaughnessy, Unix world
Loadable Kernel Module (LKM)

- A new kernel module can be added on the fly (while OS is still running)
- LKMs are often called “kernel modules”
- They are not user program
Types of LKM

• Device drivers
• Filesystem driver (one for ext2, MSDOS FAT16, 32, NFS)
• System calls
• Network Drivers
• TTY line disciplines. special terminal devices.
• Executable interpreters.
Basic LKM (program)

• Every LKM consist of two basic functions (minimum):

  int init_module(void) /*used for all initialization stuff*/
  {
    ...
  }
  void cleanup_module(void) /*used for a clean shutdown*/
  {
    ...
  }

• Loading a module - normally restricted to root - is managed by issuing the following command: # insmod module.o
LKM Utilities

• insmod
  – Insert an LKM into the kernel.
• rmmod
  – Remove an LKM from the kernel.
• depmod
  – Determine interdependencies between LKMs.
• kerneld
  – Kerneld daemon program
• ksym
  – Display symbols that are exported by the kernel for use by new LKMs.
• Ismod
  – List currently loaded LKMs.
• modinfo
  – Display contents of .modinfo section in an LKM object file.
• modprobe
  – Insert or remove an LKM or set of LKMs intelligently. For example, if you must load A before loading B, Modprobe will automatically load A when you tell it to load B.
Common LKM util

• Create a special device file
  % mknod /dev/driver c 40 0

• Insert a new module
  % insmod modname

• Remove a module
  % rmmod modname

• List module
  % lsmod
  Or % more /proc/modules

  audio          37840  0
  cmpci          24544  0
  soundcore      4208   4 [audio cmpci]
  nfsd           70464  8 (autoclean)
Linux Device Drivers

- A set of API subroutines (typically system calls) interface to hardware
- Hide implementation and hardware-specific details from a user program
- Typically use a file interface metaphor
- Device is a special file
Linux Device Drivers (continued)

- Manage data flow between a user program and devices
- A self-contained component (add/remove from kernel)
- A user can access the device via file name in /dev, e.g. /dev/lp0
General implementation steps

- Understand the device characteristic and supported commands.
- Map device specific operations to unix file operation.
- Select the device name (user interface)
  - Namespace (2-3 characters, /dev/lp0)
- (optional) select a major number and minor (a device special file creation) for VFS interface
  - Mapping the number to right device sub-routines
- Implement file interface subroutines
- Compile the device driver
- Install the device driver module with loadable kernel module (LKM)
- or Rebuild (compile) the kernel
Read/write (I/O)

- Pooling (or synchronous)
- Interrupt based
Device Driver interface

Note: This picture is excerpted from Write a Linux Hardware Device Driver, Andrew O'Shaughnessy, Unix world
VSF & Major number

- principal interface between a device driver and Linux kernel
File operation structure

• struct file_operations Fops = {
    NULL, /* seek */
    xxx_read,
    xxx_write,
    NULL, /* readdir */
    NULL, /* select */
    NULL, /* ioctl */
    NULL, /* mmap */
    xxx_open,
    NULL, /* flush */
    xxx_release /* a.k.a. close */
};
Device special file

- Device number
  - Major (used to VFS mapping to right functions)
  - Minor (sub-devices)
- mknod /dev/stk c 38 0
- ls -l /dev/tty
  - crw-rw-rw- 1 root root 5, 0 Apr 21 18:33 /dev/tty
Register and unregister device

```c
int init_module(void) /*used for all initialition stuff*/
{
    /* Register the character device (atleast try) */
    Major = register_chrdev(0,
                            DEVICE_NAME,
                            &Fops);

    :
}

void cleanup_module(void) /*used for a clean shutdown*/
{
    ret = unregister_chrdev(Major, DEVICE_NAME);
    ...
}
```
Register and unregister device

- **compile**
  
  -Wall -DMODULE -D__KERNEL__ -DLINUX -DDEBUG -I
  /usr/include/linux/version.h

- **Install the module**
  
  %insmod module.o

- **List the module**
  
  %lsmod

- **If you let the system pick Major number, you can find the major number (for special creation) by**
  
  % more /proc/devices

- **Make a special file**
  
  % mknod /dev/device_name c major minor
Device Driver Types

• A character device driver (c)
  – Most devices are this type (e.g. Modem, Ip, USB
  – No buffer.

• A block device driver (b)
  – through a system buffer that acts as a data cache.
  – Hard drive controller and HDs
Implementation

• Assuming that your device name is Xxx
• Xxx_init() initialize the device when OS is booted
• Xxx_open() open a device
• Xxx_read() read from kernel memory
• Xxx_write() write
• Xxx_release() clean-up (close)
• init_module()
• cleanup_module()
Supported functions

- `add_timer()`
  - Causes a function to be executed when a given amount of time has passed
- `cli()`
  - Prevents interrupts from being acknowledged
- `end_request()`
  - Called when a request has been satisfied or aborted
- `free_irq()`
  - Frees an IRQ previously acquired with `request_irq()` or `irqaction()`
- `get_user*()`
  - Allows a driver to access data in user space, a memory area distinct from the kernel
- `inb(), inb_p()`
  - Reads a byte from a port. Here, `inb()` goes as fast as it can, while `inb_p()` pauses before returning.
- `irqaction()`
  - Registers an interrupt like a signal.
- `IS_*(inode)`
  - Tests if inode is on a file system mounted with the corresponding flag.
- `kfree*()`
  - Frees memory previously allocated with `kmalloc()`
- `kmalloc()`
  - Allocates a chunk of memory no larger than 4096 bytes.
- `MAJOR()`
  - Reports the major device number for a device.
- `MINOR()`
  - Reports the minor device number for a device.
Supported functions

- `memcpy_*fs()`
  - Copies chunks of memory between user space and kernel space
- `outb(), outb_p()`
  - Writes a byte to a port. Here, `outb()` goes as fast as it can, while `outb_p()` pauses before returning.
- `printk()`
  - A version of `printf()` for the kernel.
- `put_user*()`
  - Allows a driver to write data in user space.
- `register_*dev()`
  - Registers a device with the kernel.
- `request_irq()`
  - Requests an IRQ from the kernel, and, if successful, installs an IRQ interrupt handler.
- `select_wait()`
  - Adds a process to the proper `select_wait` queue.
- `*sleep_on()`
  - Sleeps on an event, puts a `wait_queue` entry in the list so that the process can be awakened on that event.
- `sti()`
  - Allows interrupts to be acknowledged.
- `sys_get*()`
  - System calls used to get information regarding the process, user, or group.
- `wake_up*()`
  - Wakes up a process that has been put to sleep by the matching `*sleep_on()` function.
Pitfalls

1. **Using standard libraries**: can only use kernel functions, which are the functions you can see in `/proc/ksyms`.

2. **Disabling interrupts** You might need to do this for a short time and that is OK, but if you don't enable them afterwards, your system will be stuck.
Resources

- **Linux Kernel API**: http://kernelnewbies.org/documents/kdoc/kernel-api/linuxkernelapi.html
- **Kernel development tool** http://www.jungo.com/products.html