Operating Systems II

Device Drivers

Device Driver Motivation

- Multiple processes trying to access the same hardware.
  - Need to ensure integrity of the device.
- Quote: “Device drivers are a collection of routines which write magical numbers to magical places in the hardware”.
  - But, how does one integrate these magical routines into the OS?

OS Responsibilities

- When it comes to devices, your OS has several responsibilities:
  - has to provide full access to the features of the hardware device
  - needs to coordinate hardware resources.
    - Only one task can manipulate a device at a time.
  - needs to protect users from the device.
    - Incorrect actions by the device driver can crash the system.

Driver Overview

- Device drivers are executed as part of the system kernel.
- Drivers are either part of the kernel image, or loaded as a module (statically or dynamically).
- In either case it must provide a **standard** interface by which ALL devices can be accessed.
  - Necessary for kernel flexibility.

Everything is a File

- You have probably heard the saying that in UNIX, everything is a file.
- This is true, because ALL devices are provided as a file in the UNIX OS.
- Special entry points are provided to the driver through virtual files.
- The user can then perform basic file operations on the device
  - read, write, open, close, …

Major & Minor Numbers

- Devices require unique identification in the system
  - Major and Minor numbers?
- The major number identifies the type of device
- The minor number identifies any mode or subunit of the device.
  - Check out Linux's `Documentation/devices.txt`

Blocks & Characters

- Devices within the kernel can be separated between character and block devices.
  - Block devices work on the basis of blocks of memory or the management of block sizes.
  - Character devices work on devices character by character or in sequential byte streams.

Example from devices.txt

1 char	Memory devices		
	1 = /dev/mem	Physical memory access	
	2 = /dev/kmem	Kernel virtual memory access	
	3 = /dev/null	Null device	
	4 = /dev/port	I/O port access	
	5 = /dev/zero	Null byte source	
	6 = /dev/core	OBSOLETE - replaced by /proc/kcore	
	7 = /dev/full	Returns ENOSPC on write	
	8 = /dev/random	Nondeterministic random number gen.	
	9 = /dev/urandom	Faster, less secure random number gen.	
	10 = /dev/aio	Asyncronous I/O notification interface	
block	RAM disk		
	0 = /dev/ram0	First RAM disk	
	1 = /dev/ram1	Second RAM disk	
	...		
	250 = /dev/initrd	Initial RAM disk {2.6}	

Creating a Virtual Driver File

- One can create a virtual driver file in Linux by using the mknod command.
  - `mknod /dev/name type major minor`
  - `/dev/name` the virtual file for the device.
  - `type` indicates whether it is a block or character device.
  - `major/minor` the unique identifier for device.

Device Drivers as a Module

- If a device is provided as a module, then it must be registered and unregistered with the kernel.
- To register a device driver
  - `register_chrdev()` or `register_blkdev()`
- To unregister a device driver
  - `unregister_chrdev()` or `unregister_blkdev()`

Accessing the Device

- Once created, the device is accessible through a standard interface.
  - Function pointers in a structure.
- Two types of structures typically exist, one for character devices and one for block devices.

file_char_op Structure

- Contains the mapped functions to use when manipulating a character device.
- The user will fill the structure with mapped functions and then pass it as a parameter in registering the device.
- Internally the kernel keeps this structure in a hash table for character devices that can be quickly accessed by major and minor numbers.

file_char_op (cont.)

- owner - reference to the registered module.
- llseek - updates the file pointer.
- read - read n bytes starting at given offset.
- write - writes n bytes starting at given offset.
- readdir - returns the next directory entry.
- ioctl - sends a command to underlying hardware device.

`file_char_op` (cont.)

- `poll` - checks if activity, sleeps until something happens.
- `mmap` - maps the file into a process address space.
- `open` - creates a file object linked to object.
- `close` - closes file linked to object.
- `flush` - flushes file (filesystem dependent).

file_char_op (cont.)

- release - releases the file object.
- fsync - writes all cached data of file to disk.
- fasync - enables/disables asynchronous I/O notification.
- lock - applies a lock to the file.
- readv - reads bytes from file and puts data into vectors.

file_char_op (cont.)

- writev - writes bytes from vectors and puts into file.
- sendpage - transfers data from one file to another (used by sockets).
- get_unmapped_area - gets unused address range to map the file (frame buffer memory mappings).

`file_block_op` Structure

- Contains the mapped functions to use when manipulating a block device.
- The user will fill the structure with mapped functions and then pass it as a parameter in registering the device.
- Internally the kernel keeps this structure in a unique hash table for block devices that can be quickly accessed by major and minor numbers.

file_block_op Structure

- open - open the block device file.
- release - close the last reference to block device file.
- ioctl - issue a i/o control system call on block device.

file_block_op Structure

- check_media_change - check if the media has been changed (e.g. floppy drive)
- revalidate - check if the block device holds valid data.
- owner - reference to the registered module.
- Other basic operations ...

Linux Example: /dev/random

- You can check out devices/char/random.c for more info on this example.
- The random device simply returns a random number to the reading process.
- So it only registers the `read`, `write`, `poll`, and `ioctl` functions.

Polling and Interrupts

- Devices are slower than the processor.
- Waiting for devices to perform operations is a waste of processing time.
- Device drivers typically call schedule to move the processor over to another task.
- But how does the processor know the device has completed?
  - Several solutions including interrupts and polling.

Polling

- Certain devices can work under the conditions that the processor can ask the device if it has completed the operation.
- In this case, the processor has to regularly check the device for completion.
- The task calls schedule() to preempt itself, but does not put itself to sleep!
  - The device will be polled when the task is next scheduled.

Interrupt Mode

- In interrupt mode the device informs the processor that it is completed its operation via an IRQ (Interrupt Request).
- This relies on the physical hardware device being capable of generating an interrupt.
- When an IRQ is generated, the processor switches to execute an interrupt service routine (ISR).

Interrupt Mode (cont.)

- Using interrupts, the device driver operation invokes `interruptible_sleep_on()` putting the task to sleep.
- The ISR upon receiving the IRQ, performs any additional work and signals the original task to resume.
- Example - mouse.
  - Every movement causes an IRQ
  - ISR reads the port and passes it along to the application.

Allocating IRQ’s

- IRQ’s have to be requested. Not hard coded to each device.
  - Impossible to predict the different combinations of hardware equipment in a computer.
- Allocated through the use of:
  - `int request_irq(unsigned int irq, void (*handler_func)(), unsigned long irqflags, char* devname, void* dev_id)`

Registering Interrupts

- The ISR routines associated with an IRQ can be set to be either interruptible or non-interruptible when requesting the IRQ.
  - This can be conveyed through the `irqflags`.
- `dev_name` is a historical field that is still used so system can show what IRQs are in use by what device.

Sharing IRQ's

- Unfortunately, there are a limited number of IRQ's in a machine.
- So how does the operating system provide a less limited number of IRQ's?
- It shares the same IRQ between different devices.
  - *RESTRICTION:* the device must support interrogation by the processor.

Sharing IRQ's (cont.)

- To share an IRQ, both device drivers when registering must indicate through the irqflags that they are capable of sharing.
  - If so, for the IRQ entry, a chain is built of ISRs for each device that is sharing the IRQ.
- When the IRQ occurs, each ISR in the chain is invoked.
- Each ISR then interrogates its hardware device to determine if it generated the interrupt.

Dead Hardware?

- There is a possibility that a device can not respond when expected by the device driver.
- Process requests to read floppy disk, device driver issues read operation, but the device never responds.
  - i.e. No floppy/CD in drive
- The operating system deals with this problem through the use of timers.

Timers

- Before the device driver releases control, it registers a **timer**.
  - As discussed previously...
- If no interrupt from the device is received, the **timer** will expire and a registered function of the devices choice will be executed.
- If the interrupt occurs then the ISR must either restart or delete the **timer** to prevent over service.