Linux Device Drivers

Jie Liao
April 7, 2016
A device driver is a piece of software that allows the operating system to communicate with the devices.
Roadmap

- Fundamentals
- Basic device types
- Bus controller/device drivers
- Linux graphic stack
- Universal drivers
Physical device topology

- CPU
  - PCI bus
    - ALSA device
    - GPU device
  - USB bus
    - USB Net device
    - I2C bus
      - EEPROM device
Linux device model

• Unified model to represent devices and describe device topology
• Minimize code duplication
• Device drivers separated from controller drivers
• Hardware description separated from drivers themselves
• Device and driver binding
struct device
{
    struct device  *parent;
    struct device_private  *p;

    struct kobject kobj;

    struct bus_type *bus;  /* type of bus device is on */
    struct device_driver *driver;  /* which driver has allocated this device */

    void   *driver_data;  /* Driver data, set and get with dev_set/get_drvdata */

    u64   *dma_mask;
    u64   coherent_dma_mask;

    void   (*release)(struct device *dev);
    /* some fields are evicted */
...
}
struct device_driver
{
    const char *name;
    struct bus_type *bus;
    struct module *owner;
    const char *mod_name; /* used for built-in modules */
    bool suppress_bind_attrs; /* disables bind/unbind via sysfs */

    const struct of_device_id *of_match_table;
    const struct acpi_device_id *acpi_match_table;

    int (*probe)(struct device *dev);
    int (*remove)(struct device *dev);
    void (*shutdown)(struct device *dev);
    int (*suspend)(struct device *dev, pm_message_t state);
    int (*resume)(struct device *dev);
    struct attribute_group **groups;

    const struct dev_pm_ops *pm;
    struct driver_private *p;
}

struct bus_type
{
    const char *name;
    const char *dev_name;
    struct device *dev_root;

    int (*match)(struct device *dev, struct device_driver *drv);
    int (*uevent)(struct device *dev, struct kobj_uevent_env *env);
    int (*probe)(struct device *dev);
    int (*remove)(struct device *dev);
    void (*shutdown)(struct device *dev);

    int (*suspend)(struct device *dev, pm_message_t state);
    int (*resume)(struct device *dev);

    const struct dev_pm_ops *pm;
    /* some fields are evicted */
    ...
}
struct class
{
    const char *name;

    int (*dev_uevent)(struct device *dev, struct kobj_uevent_env *env);
    char *(*devnode)(struct device *dev, umode_t *mode);

    void (*class_release)(struct class *class);
    void (*dev_release)(struct device *dev);

    int (*suspend)(struct device *dev, pm_message_t state);
    int (*resume)(struct device *dev);

    const struct dev_pm_ops *pm;

    struct subsys_private *p;...
    /* some fields are evicted */
    ...
Kobject & sysfs

{  
  const char *name;
  struct list_head entry;
  struct kobject *parent;
  struct kset *kset;
  struct kobj_type *ktype;
  struct kernfs_node *sd;
  struct kref kref;
  /* some fields are evicted */
  ...
}

- Reference counting
- Sysfs representation
- Data structure glue
- Hotplug event handling
Subsystem

• A top-level view to the system’s structure as a whole

• A view to the device model data structure rather than physical connection
Not a complete model

• Devices are wrapped into subsystem’s data structures
• One device/driver could be registered to multiple subsystems
• Multiple layers of frameworks between the driver and user space
• Common frameworks extract common driver functionalities
• etc.
Memory-mapped I/O and port I/O

Physical address space, accessed using normal load/store instructions

- RAM
- MMIO Registers

Separate address space, accessed using specific instructions

- PIO Registers
Memory-mapped I/O and port I/O

```
struct resource * request_mem_region (start, len, name)

void __iomem * ioremap(phy_addr, size)

read[bwl]/write[bwl]

MMIO

struct resource * request_region (start, len, name)

in[bwl]/out[bwl]

PIO
```
DMA

[Diagram showing DMA controller, RAM, Descriptor, Buffer, FIFO, SPI Controller, Audio Interface, Network Controller, Request Lines]

DMA memory mapping

Coherent DMA mapping

- Exists for the whole module life time
- Kernel allocates the buffer
- CPU can access the memory while the mapping is valid
- Set up could be expensive

```c
void *dma_alloc_coherent(
    dev, size, handle, gfp)
```

Streaming DMA mapping

- Usually used for one DMA transfer
- Buffer already allocated by the driver
- CPU cannot access the memory while the mapping is valid
- Recommended solution

```c
void *dma_map_single(
    dev, handle, size, dir)
```

DMA requires physical memory to be contiguous!
Initiate a DMA transfer

1. Allocate a DMA channel
2. Set slave and controller config
3. Get a DMA descriptor
4. Submit the transaction
5. Issue pending DMA request

```c
struct dma_chan *dma_request_chan(dev, name)

int dmaengine_slave_config(chan, config)

struct dma_async_tx_descriptor *dmaengine_prep_slave_sg(
  chan, sgl, sg_len, direction, flags);

dma_cookie_t dmaengine_submit(desc)

void dma_async_issue_pending(chan);
```
Interrupt

int devm_request_irq(
    struct device *dev,
    unsigned int irq,
    irq_handler_t handler,
    unsigned long irq_flags,
    const char *devname,
    void *dev_id);

void devm_free_irq(
    struct device *dev,
    unsigned int irq,
    void *dev_id);
Interrupt Handling

• Top half
  • Should complete as soon as possible
  • Interrupt disabled
  • Take data out and invoke the bottom half

• Bottom half
  • Does the rest of the interrupt handling at a postponed time
  • Interrupt enabled
  • Implemented as softirqs, tasklets or workqueues
Hotplug with udev

- **Allocate resources**
  - Match driver and probe if possible

- **uevent**
  - Kobject created

- **uevent**
  - Driver core notified by interrupt

- **Rules in */lib/udev/rules.d/**
  - Can override

- **Set rules in */etc/udev/rules.d/**

- **Udevd listening**
  - Create a file in */dev*
  - Load modules when necessary

- **Plug/remove device**
## Basic device types

<table>
<thead>
<tr>
<th>Type</th>
<th>Char devices</th>
<th>Block devices</th>
<th>Network Devices</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transfer format</td>
<td>Transfer data in characters</td>
<td>Transfer data in blocks</td>
<td>Transfer data in packets</td>
</tr>
<tr>
<td>User space control</td>
<td>Standard system call via /dev/</td>
<td>Standard system call via /dev/</td>
<td>Socket API and ifconfig</td>
</tr>
<tr>
<td>Kernel space represent</td>
<td>cdev/ char_device Struct ¹</td>
<td>gendisk</td>
<td>net_device</td>
</tr>
<tr>
<td>Operation connection</td>
<td>struct file_operations</td>
<td>block_device_operations</td>
<td>net_device_ops</td>
</tr>
<tr>
<td>Registration API</td>
<td>cdev_init()/register_chrdev()</td>
<td>register_blkdev()</td>
<td>register_netdev()</td>
</tr>
</tbody>
</table>
Block devices

Network devices

Network stack layer
struct sk_buff

struct net_device

NIC driver

Bus driver instantiation

Bus infrastructure (PCI, USB...)

** struct net_device – The DEVICE structure.
Actually, this whole structure is a big mistake. It mixes I/O
data with strictly "high-level" data, and it has to know about
almost every data structure used in the INET module.
USB communication

USB device driver creates URB and populates the fields

Submitted URB could be canceled

USB host controller driver does the URB transfer and notifies USB core

USB core notifies USB device driver of the transfer status

USB core submits the URB to the USB host controller driver

USB device driver submits the URB to USB core
I2C

- Processor
- I2C controller (master)
- I2C device (slave)
- I2C controller driver
- I2C device driver
- Upper layer software
- Bus infrastructure (PCI, platform)

```
struct i2c_client ->
struct device

struct i2c_driver ->
struct device_driver
```
I2C data transfer

• Only master can initiate transaction, slaves only respond to the master

• i2c_master_send/recv(i2c_client, buf, count)

• i2c_transfer
Platform bus

• Peripheral devices are usually connected to CPU on SoCs

• Legacy peripherals assume to be attached to a bus

• Pseudo bus to glue attached peripherals
PCI

• Most common bus in PCs
• Support various devices including graphics card, sound card, network card and bus controllers
• Automatic device configuration through BIOS-type boot firmware or kernel configuration

```c
def struct pci_dev -> struct device dev
def Struct pci_driver -> struct device_driver driver
```
PCI device has 256-byte configuration register memory region

00: de 10 80 12 07 00 10 00 a1 00 00 03 10 00 80 00
10: 00 00 00 f2 0c 00 00 e8 00 00 00 00 0c 00 00 f0
20: 00 00 00 00 01 e0 00 00 00 00 00 00 00 00 00
30: 00 00 00 f3 60 00 00 00 00 00 00 00 00 00 00

pci_[read | write]_config_[byte | word | dword]

https://en.wikipedia.org/wiki/PCI_configuration_space
PCI device has up to 6 MMIO or PIO regions

unsigned long pci_resource_[start|len|end|flags](
  struct pci_dev *pdev, int bar)

MMIO mapping or PIO mapping

For generic steps to initialize PCI devices, refer to: http://free-electrons.com/kerneldoc/latest/PCI/pci.txt
Linux graphic stack

- X11 application
- OpenGL application
- Framebuffer application
- 2D driver
- 3D driver
- libdrm
- DRM/KMS
- FBDEV
- Graphic card driver
- Bus infrastructure

user space

kernel space
Linux Framebuffer

DRM/KMS

Universal drivers

• Linux kernel has already extracted common code into frameworks for each class and bus
  • PCI core, USB core, I2C core, serial core etc.
  • RPM framework, pctl framework, clock framework, mfd framework

• We need a data structure to describe the flow (bring up data-drien again)

• Driver diversity and complexity are the challenges
Device users

Driver frameworks

Device drivers

- Abstraction
- Computation
- Management

Internal states

Devices

\[ \rightarrow \]

Universal driver

Device users

Driver frameworks

Device drivers

- Abstraction
- Computation
- Device knowledge

Internal states

Devices
Universal Driver

universal drv_data list

universal device list

Universal Driver registration

match

universal driver registration

universal device registration

- universal probe
- universal reg read/write
- universal suspend/resume

driver A

driver B

driver C

device a

device b

device c

Bus cores
Backup slides start from this slide...
Frameworks work with Linux device model

Physical / virtual memory mapping

Kernel

Process n

0xFFFFFFFF

0xC0000000

0x00000000

RAM

ZONE_HIGH

ZONE_NORMAL

ZONE_DMA

I/O Memory
64-bit x86 machine memory mapping

• Virtual memory map with 4 level page tables:

• 0000000000000000 - 00007fffffffffff (=47 bits) user space, different per mm
• hole caused by [48:63] sign extension
• ffff800000000000 - ffff8fffffffffff (=43 bits) guard hole, reserved for hypervisor
• ffff880000000000 - ffffc7fffffffffff (=64 TB) direct mapping of all phys. memory
• ffffc800000000000 - ffffc8fffffffffff (=40 bits) hole
• ffffc900000000000 - ffffe8fffffffffff (=45 bits) vmalloc/ioemap space
• ffffe900000000000 - ffffe9fffffffffff (=40 bits) hole
• ffffea000000000000 - ffffeafffffffffffff (=40 bits) virtual memory map (1TB)
• ... unused hole ...
• ffffecc0000000000 - ffffecc0000000000 (=44 bits) kasan shadow memory (16TB)
• ... unused hole ...
• fffffff0000000000 - fffffff7fffffff (=39 bits) %esp fixup stacks
• ... unused hole ...
• fffffff8000000000 - ffffffffa00000000 (=512 MB) kernel text mapping, from phys 0
• ffffffffa00000000 - ffffffffa00000000 (=1525 MB) module mapping space
• fffffffff600000 - fffffffff60000000 (=8 MB) vsyscalls
• ffffffffffe00000 - ffffffffffe000000 (=2 MB) unused hole
mmap

• Map (device) file content to part of the virtual address space of the process

• Avoid expensive read/write system calls to access I/O memory or ports

• Supported by the MMU hardware (device driver must have mmap operation defined)
struct dma_slave_config {
    enum dma_transfer_direction direction;
    dma_addr_t src_addr;
    dma_addr_t dst_addr;
    enum dma_slave_buswidth src_addr_width;
    enum dma_slave_buswidth dst_addr_width;
    u32 src_maxburst;
    u32 dst_maxburst;
    bool device_fc;
    unsigned int slave_id;
}

struct dma_async_tx_descriptor {
    dma_cookie_t cookie;
    enum dma_ctrl_flags flags; /* not a 'long' to pack with cookie */
    dma_addr_t phys;
    struct dma_chan *chan;
    dma_cookie_t (*tx_submit)(struct dma_async_tx_descriptor *tx);
    dma_async_tx_callback callback;
    void *callback_param;
#if defined CONFIG_ASYNC_TX_ENABLE_CHANNEL_SWITCH
    struct dma_async_tx_descriptor *next;
    struct dma_async_tx_descriptor *parent;
    spinlock_t lock;
#endif
}

44
Windows device driver architecture

• Plug and Play (PnP) Device Tree
• Device object and driver pairs compose device stack
• PnP manager notify a device or driver attachment
Windows device driver architecture

Applications access devices in Mac OS X

- I/O Kit family APIs
- POSIX APIs, which provides access to storage, network and serial
- I/O Registry, which is a database for device properties