北京郵電大學
Beijing University of Posts and Telecommunications

操作系统内核
-基于Linux

第10讲
设备驱动

主讲：杨文川

内容

- 1 设备驱动概述
- 2 I/O空间管理
- 3 设备驱动模型
- 4 字符设备驱动程序简介
- 5 块设备驱动程序简介
- 6 动手实践-编写字符设备驱动程序
- 7 实践-块设备驱动程序实现

内容导航：

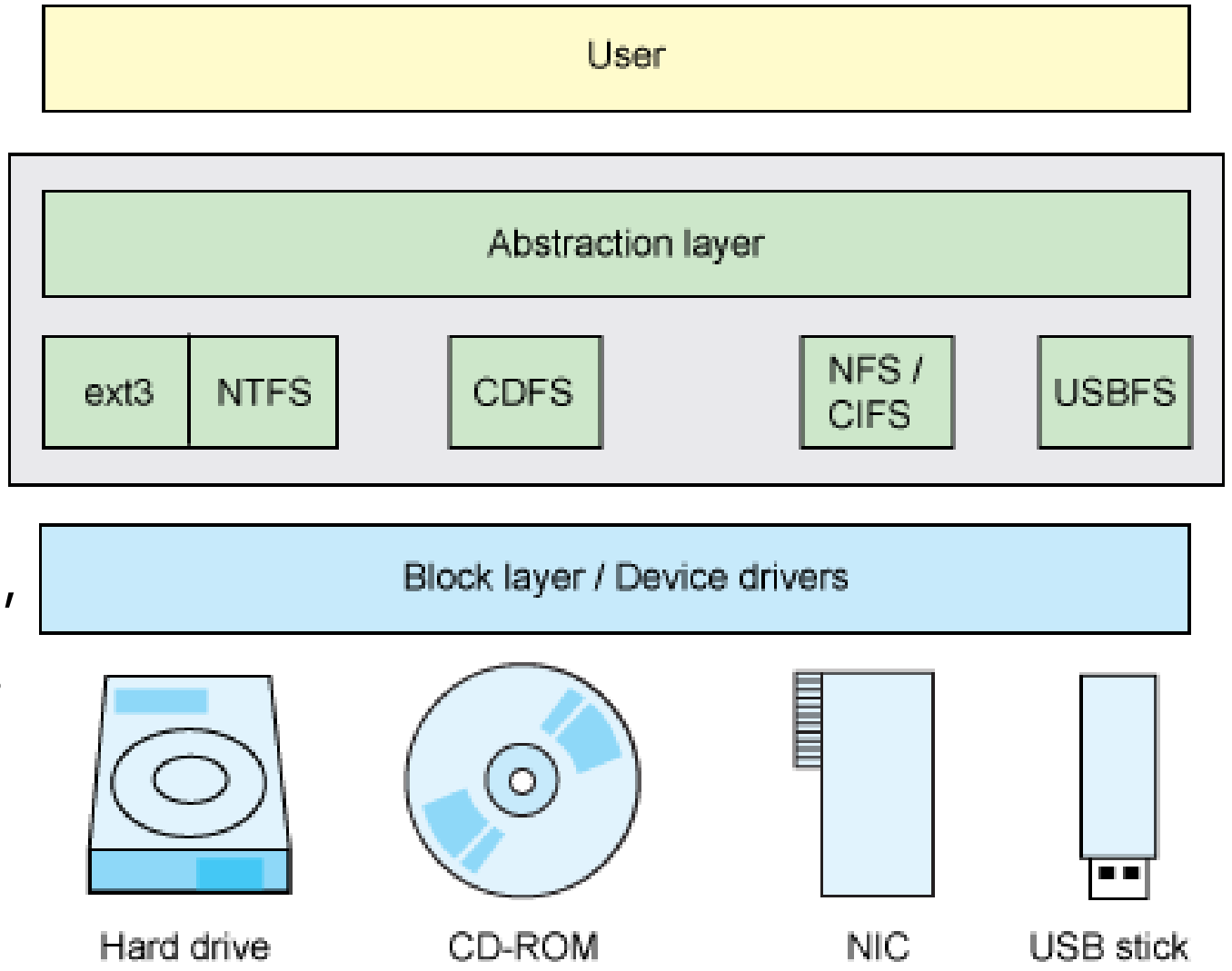
1 设备驱动概述

为什么引入设备驱动程序

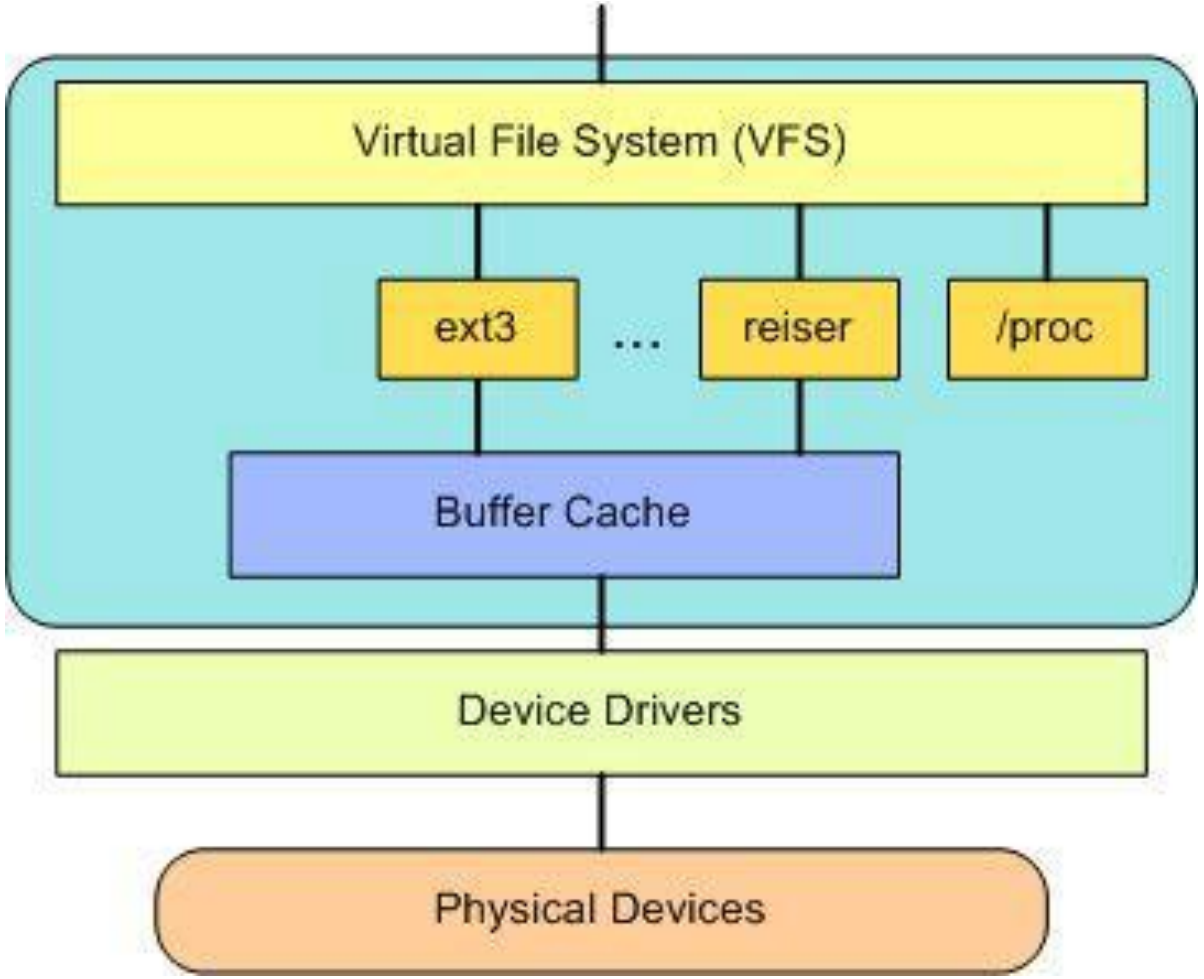
- 计算机中三个最基本的物质基础是
- **CPU、内存和输入输出(I/O)设备。**

- 与I/O等硬件设备相比，文件系统是一种逻辑意义上的存在，它对设备的操作更方便有效、组织更接近人类的思维方式。
- 文件操作是对设备操作的组织和抽象，
- 设备操作则是对文件操作的最终实现。

- 设备驱动的引入，可以使得，控制设备，像操作普通文件一样方便有效。



设备驱动程序-隐藏设备的细节



- 设备驱动程序在Linux内核中扮演着特殊的的角色，它们是一个个独立的“黑盒子”
- 使某个硬件响应一个定义良好的内部编程接口，这些接口完全隐藏了设备的工作细节。
- 用户的操作，通过一组标准化的调用执行，而这些**调用**独立于特定的驱动程序任务。
- **驱动程序**的任务，是将这些调用映射到作用于实际硬件的设备特有操作上。

设备纳入文件管理体系下

- Linux操作系统把设备纳入 文件系统 的范畴来管理
- 每个设备都对应一个文件名，在内核中也就对应一个索引节点
- 对文件操作的系统调用，大都适用于设备文件
- 从应用程序的角度看，设备文件 逻辑上的空间，是一个线性空间(起始地址为0，每读取一个字节加1)。
- 从这个 逻辑空间 到具体 设备物理空间(如磁盘的磁道、扇区)的映射，由内核提供，并被划分为 **文件操作** 和 **设备驱动** 两个层次
- 对一个具体设备而言，这是一个事物的不同层次。
- 因此可以从概念上，把一个系统划分为三个层次：
- **应用、文件系统** 和 **设备驱动**

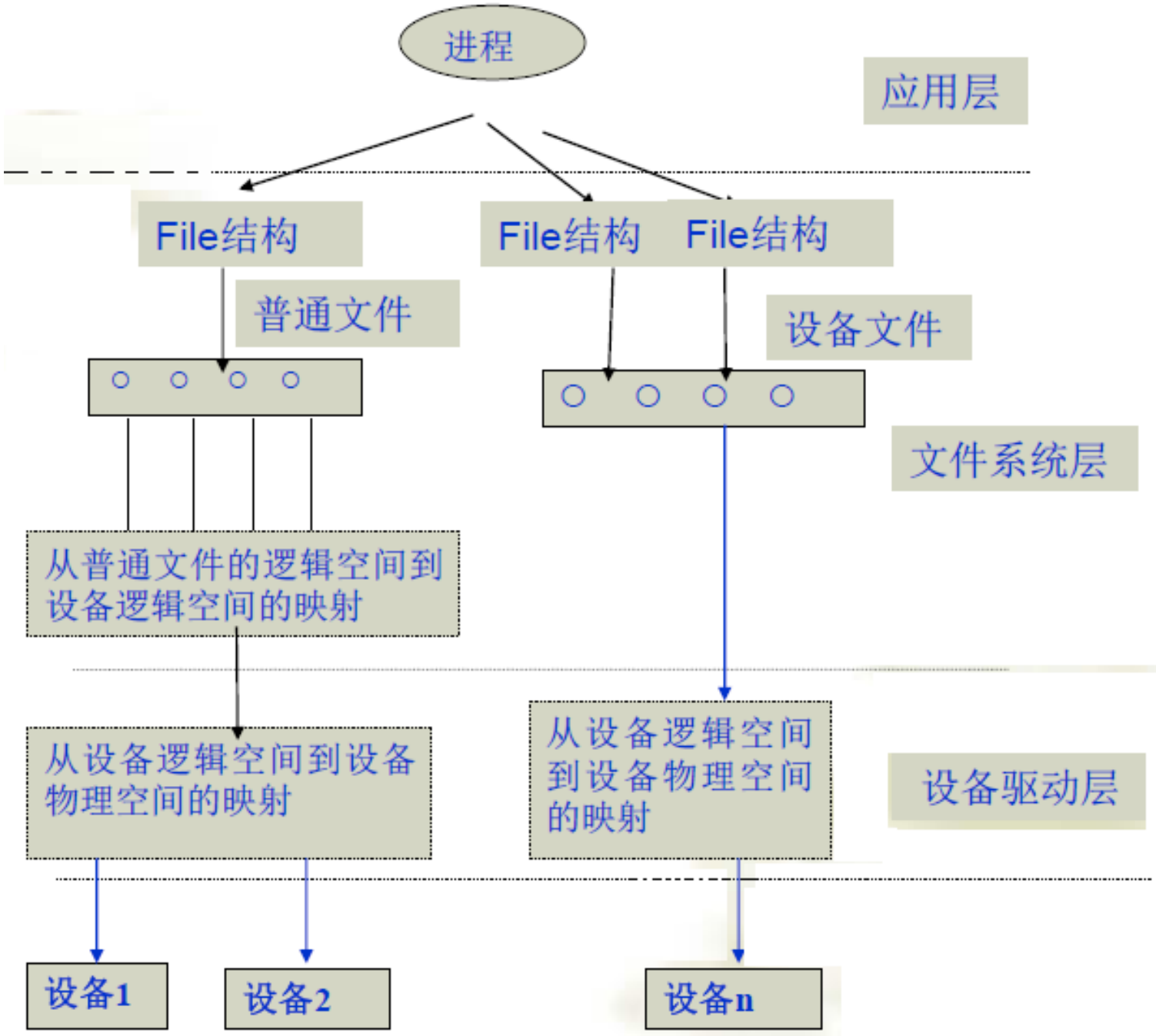
层次和厚度

- 不同的设备，其文件系统层的“厚度”不同。
- 像字符终端这样的**字符设备**，其文件系统就比较“薄”，其设备驱动层也比较简单。

- 像**磁盘**这样结构性很强，并且内容需要进一步组织和抽象的设备来说，其文件系统就很“厚重”，这是取决于磁盘设备的复杂性。
- 一方面是对磁盘物理空间的立体描述，如柱面、磁道、扇区；
- 另一方面是从物理空间到逻辑空间的抽象，如第一层抽象，即线性地址空间中的“块”，
- 以及在块之上又一层组织和抽象，即“磁盘文件”。

设备驱动的层次观

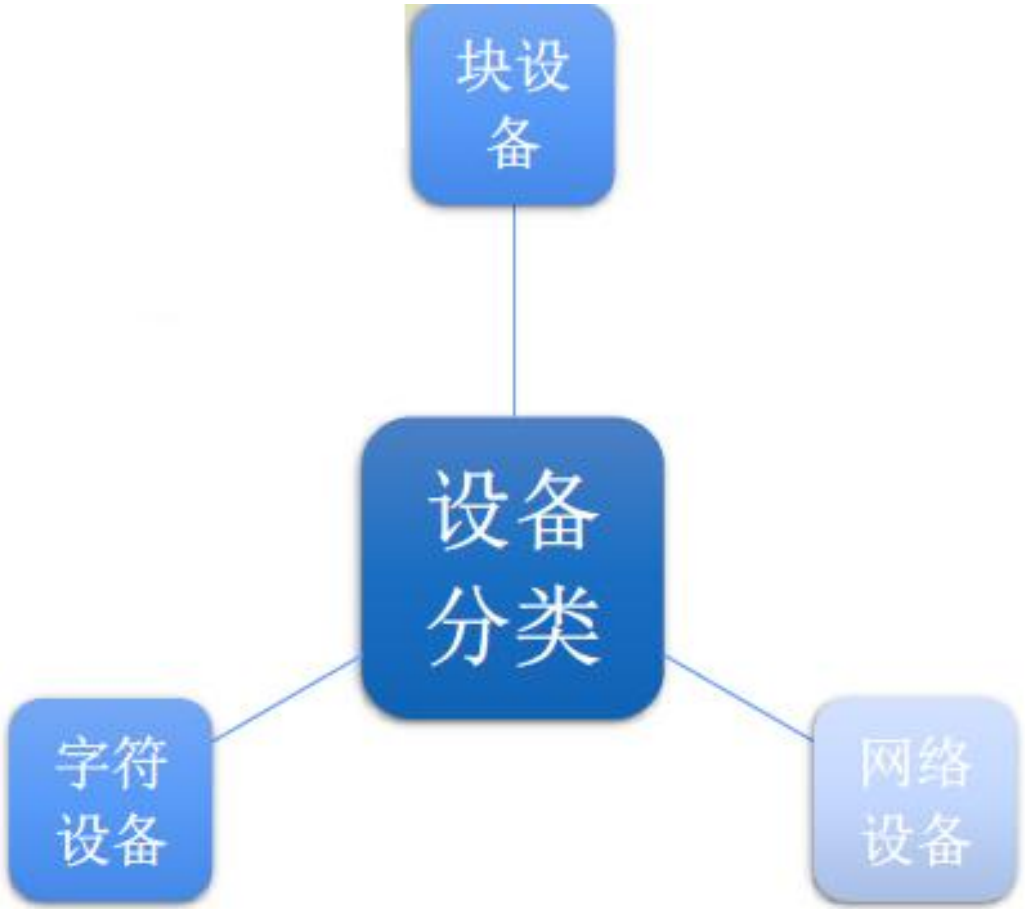
- 分层后，在物理介质上的第一层抽象，使操作者不必关心读/写的物理位置究竟在哪一个磁道，哪一个扇区；
- 而第二层抽象则使操作者不必关心读/写的内容在哪一个逻辑“块”中。
- 如图所示，可以把第一层抽象归为设备驱动，而把第二层抽象归为文件系统。



如何识别设备

- 与文件用唯一的索引结点标识相似，一个物理设备也用唯一的索引节点标识，索引节点中记载着与特定设备建立连接所需的信息。
- 这种信息由三部分组成包括：设备的类型、主设备号和次设备号。
- 其中设备类型和主设备号，结合在一起唯一地确定了设备的驱动程序及其接口，而次设备号则说明目标设备是同类设备中的第几个。
- 如图所示，通过索引节点，主设备号+次设备号来标识一个设备。

设备的分类



- Linux将设备分成三大类：
- 1.一类是 **块设备**，像磁盘那样以块或扇区为单位，成块进行输入 / 输出的设备；
- 2.一类是 **字符设备**，像键盘那样以字符为单位，逐个字符进行输入 / 输出的设备。
- 3.一类是 **网络设备**。

不同设备类型的差异性

- Linux网络设备驱动，与字符设备和块设备有很大的不同。
- 字符设备和块设备对应/dev下的一个设备文件。而网络设备不存在这样的设备文件。网络设备使用套接字socket访问，虽然也使用read,write系统调用，但这些调用只作用于软件对象。
- 块设备只响应来自内核的请求，而网络驱动程序异步接收来自外部世界的数据包，并发送到内核。
- 文件系统通常都建立在块设备上，也有很多文件系统放在内存，比如/proc，不需要驱动程序。

如何建立设备文件

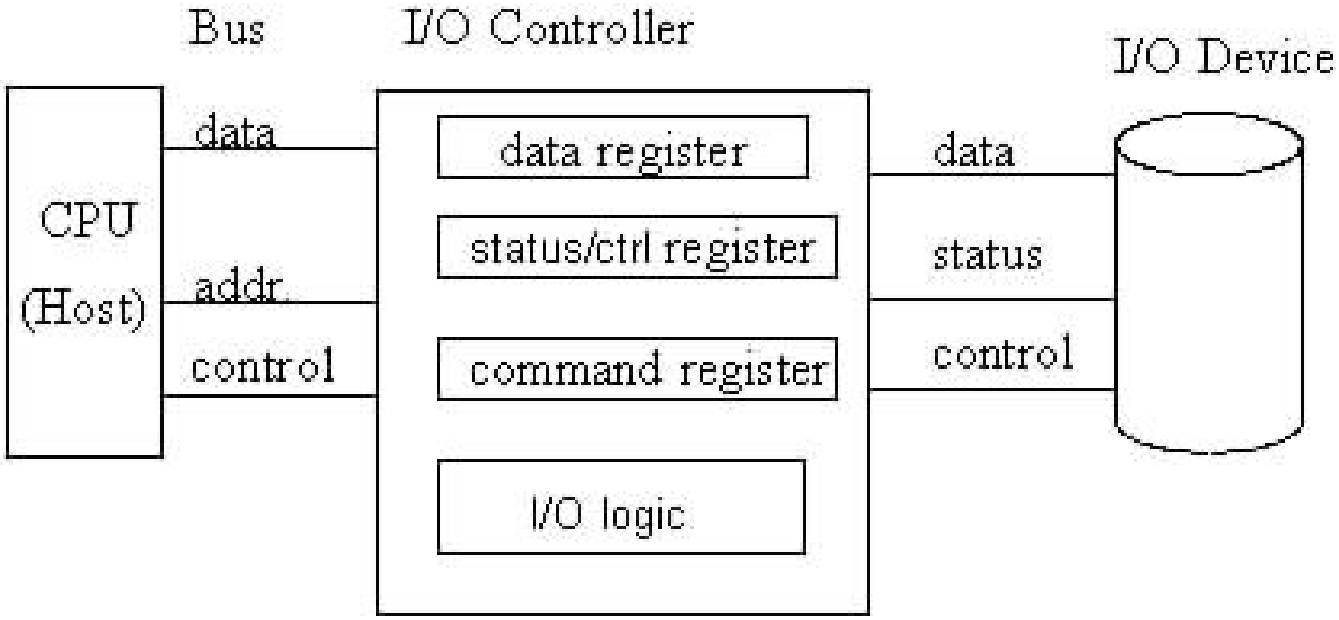
- 如图所示，建立设备文件有两种方式，
- 一是通过系统调用mknode()
- 一是通过命令mknod。

内容导航：

2 I/O空间的管理

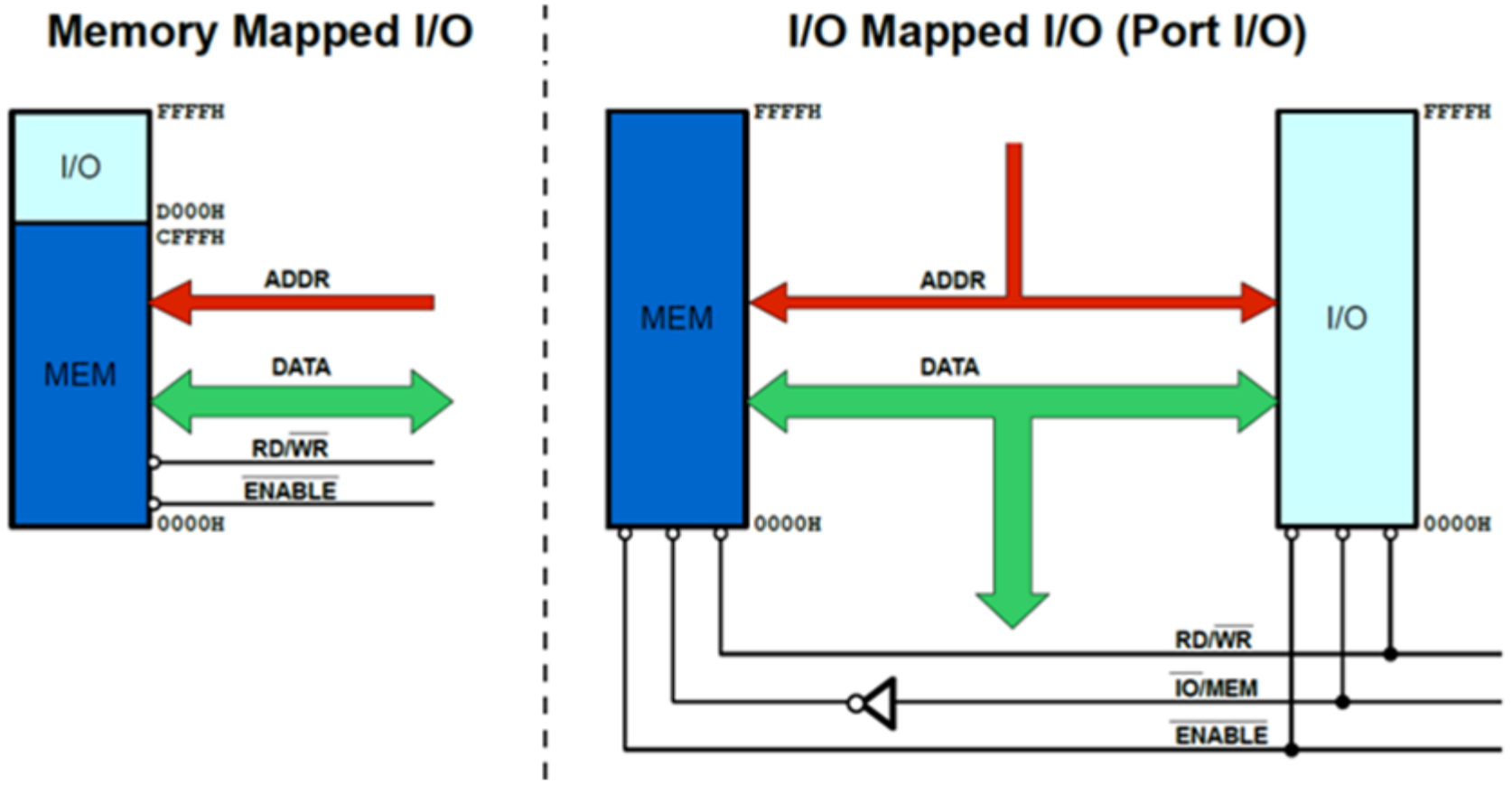
设备控制器

- 设备控制器是计算机中的一个实体，其主要职责是控制一个或多个I/O设备，以实现I/O设备和处理器之间的数据交换。
- 它是CPU与I/O设备之间的接口，它通过控制总线接收从CPU发来的命令，并去控制I/O设备工作，
- 控制器相当于CPU与外设打交道的助理，以使处理机从繁杂的设备控制事务中解脱出来。

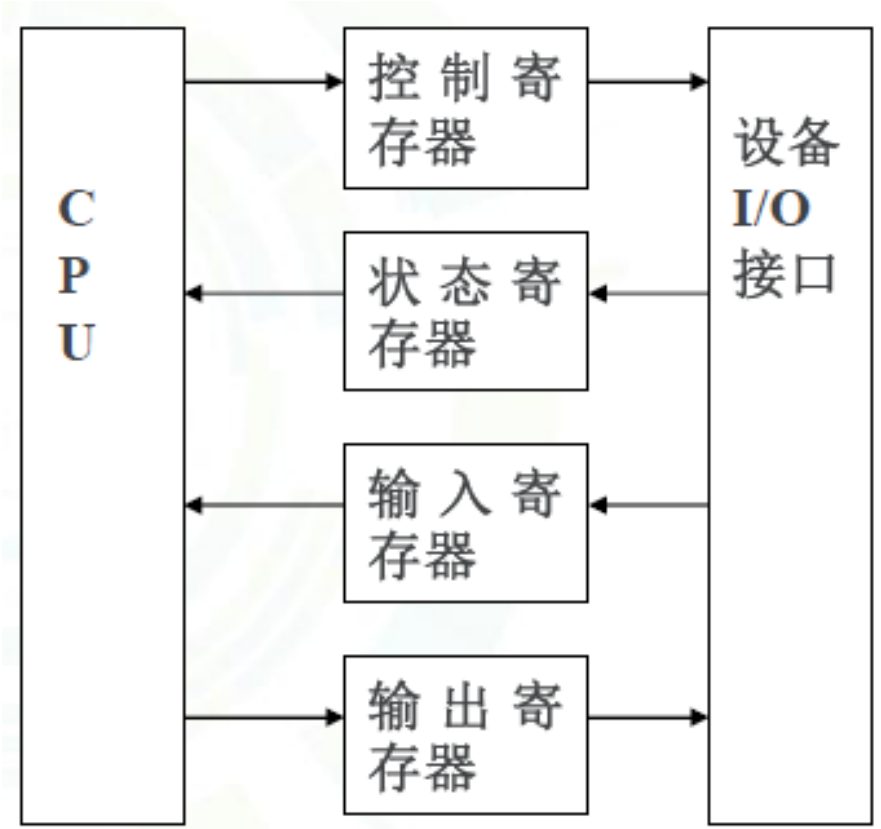


I/O 空间的管理

- 设备通常会提供一组寄存器来控制设备、读写设备以及获取设备的状态。
- 这些寄存器就是：控制寄存器、数据寄存器和状态寄存器，它们位于控制器中。
- 从编址方式来说，若I/O空间与内存一起编址，对应的内存空间被称为**I/O内存**（图左）。若I/O空间单独编址，就位于I/O空间，通常被称为**I/O端口**（图右）。



I/O 端口



专用I/O端口

- 设备驱动程序要直接访问外设或其接口卡上的物理电路，通常以寄存器的形式出现访问；
- 外设寄存器也称为I/O端口，通常包括三类：
- 控制寄存器、状态寄存器 和 数据寄存器 。

如何访问I/O 内存和I/O 端口

- 访问“I/O 内存”方式：寄存器参与内存统一编址，访问寄存器，通过访问一般的内存指令，来访问寄存器。
- 访问“I/O端口”方式：将外设的寄存器看成一个独立的地址空间，对外设寄存器的读/写设置专用指令。

访问I/O内存资源

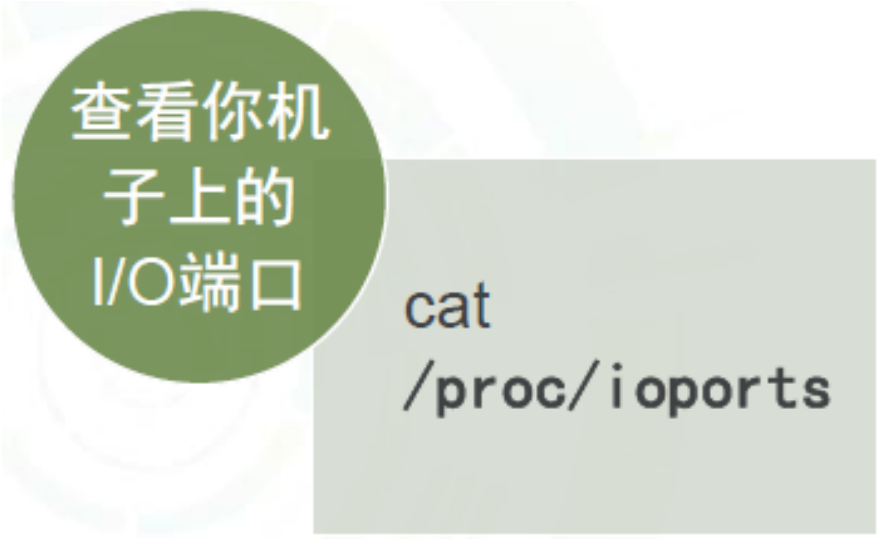
- 用于I/O指令的“地址空间”相对来说是很小的。
- 自从PCI总线出现后，无论CPU的设计采用I/O 端口方式，还是I/O内存方式，都必须将外设卡上的存储器，映射到内存空间，
- 实际上是采用了虚拟内存的手段，这样的映射是通过ioremap()来建立的，
- 该函数及其中参数的含义如左图所示

访问I/O端口空间

- 在驱动程序请求了I/O端口空间中的端口资源后，它就可以通过CPU的I/O指令来读写这些I/O端口。
- 注意：在读写I/O端口时，大多数平台都区分8位、16位和32位的端口。
- port参数指定I/O端口空间中的端口地址。在大多数平台上(如x86)它都是unsigned short类型的，其它的一些平台上则是unsigned int类型的。
- 显然，端口地址的类型，是由I/O端口空间的大小来决定的。

查看你机子上的I/O端口

- 可通过 cat /proc/ioports 来获取设备当前的I/O 端口号




```
[clj@localhost ~]$ cat /proc/ioports
0000-0cf7 : PCI Bus 0000:00
  0000-001f : dma1
  0020-0021 : pic1
  0040-0043 : timer0
  0050-0053 : timer1
  0060-0060 : keyboard
  0064-0064 : keyboard
  0070-0071 : rtc0
  0080-008f : dma page reg
  00a0-00a1 : pic2
  00c0-00df : dma2
  00f0-00ff : fpu
  0170-0177 : 0000:00:01.1
    0170-0177 : ata_piix
  01f0-01f7 : 0000:00:01.1
```

I/O 资源管理

- Linux将基于I/O端口和I/O内存的映射方式通称为“I/O区域”(I/O region)。
- Linux 设计了一个通用的数据结构resource ，来描述各种I/O 资源。该结构定义在：include/linux/ioport.h

```
struct resource {
resource_size_t start; //资源范围的开始
    resource_size_t_end; //资源范围的结束
    const char * name; //资源拥有者的名字
    unsigned long flags; //各种标志
   struct resource * parent, * sibling, * child; //指向资源树中父、兄以及孩子的指针
}
```

I/O 资源管理

request_resource()	• 把一个给定范围分配给一个 I/O 设备
alloate_resource()	• 在资源树中寻找一个给定大小和排列方式可用的范围
release_resource()	• 释放以前分配给 I/O 设备的给定范围

- 任何设备驱动程序都可以使用左边三个函数，来申请、分配和释放资源，
- 传递给它们的参数为资源树的根节点，和要插入的新资源数据结构的地址。

管理I/O端口资源

- 采用I/O端口的X86处理器，为外设实现了一个单独的地址空间，也即“I/O空间”或称为“I/O端口空间”，其大小是64KB(0x0000-0xffff)
- Linux在其所支持的所有平台上，都实现了“I/O端口空间”这一概念
- Linux是基于“I/O 区域”，来实现对I/O端口资源的管理。
- Linux在include/linux/ioport.h头文件中定义了三个对I/O端口空间进行操作的接口函数

管理I/O区域资源

- Linux将基于I/O端口和基于I/O内存的资源统称为“I/O区域”。
- I/O 区域仍然是一种I/O资源，因此它仍然可以用resource结构类型来描述。
- Linux在include/linux/ioport.h头文件中定义了三个对I/O区域进行操作的接口函数

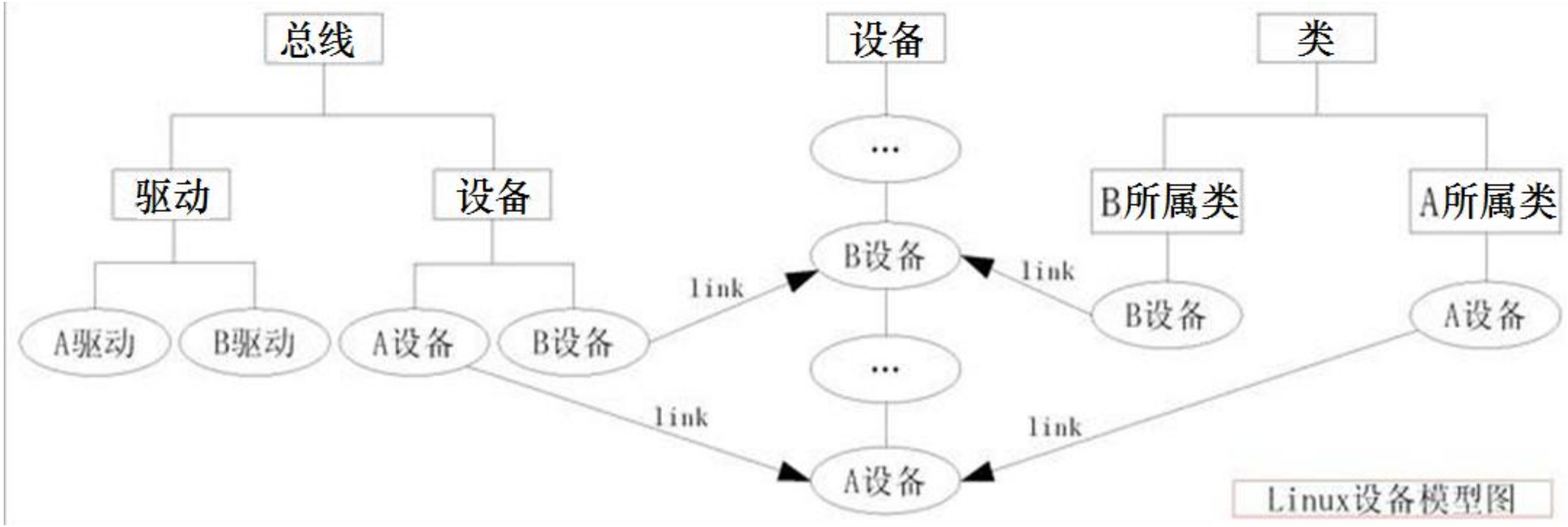
管理I/O内存资源

- 基于I/O区域的操作函数__xxx_region()，
- Linux在头文件include/linux/ioport.h中定义了三个对I/O内存资源进行操作的接口

内容导航：

3 设备驱动模型

设备驱动模型的引入


总线
驱动
设备
A驱动
B驱动
A设备
B设备
设备
...
B设备
...
A设备
link
link
link
link
类
B所属类
A所属类
B设备
A设备
Linux设备模型图

设备驱动模型的引入

- 由于Linux支持世界上几乎所有的、不同功能的硬件设备，导致Linux内核中有一半的代码是设备驱动，而且随着硬件的快速升级换代，设备驱动的代码量也在快速增长。
- 为了降低设备多样性带来的Linux驱动开发的复杂度，以及设备热拔插处理、电源管理等，Linux内核提出了设备模型(也称作Driver Model)的概念。
- 设备模型将硬件设备归纳、分类，然后抽象出一套标准的数据结构和接口。驱动的开发，就简化为对内核所规定的数据结构的填充和实现。
- 因此，Linux设备驱动模型是一种抽象，为内核建立起统一的设备模型。其目的是：提供一个对系统结构的一般性抽象描述。
- Linux设备模型跟踪所有系统所知道的设备，以便让设备驱动模型的核心程序协调驱动与新设备之间的关系。

Linux设备驱动模型引入的目的

功能	描述
电源管理和系统关机	设备之间大多情况下有依赖、耦合，因此要实现电源管理就必须对系统的设备结构有清楚的理解，应知道先关哪个然后才能再关哪个。
与用户空间通信	sys虚拟文件系统的实现与设备模型密切相关，并且向外界展示了它所表述的结构。向用户空间所提供的系统信息，以及改变操作参数的接口，都要通过/sys文件系统实现，即通过设备模型实现。
热插拔设备	处理与用户空间进行热插拔设备的通信是通过设备模型管理的。
设备分类机制	设备模型包括了对设备分类的机制，它会在更高的功能层上描述这些设备，并使得这些设备对用户空间可见。尤其是将命名设备的功能从内核层转移到用户层，大大提高了设备管理的灵活性。
对象生命周期管理	设备模型实现一系列机制以处理对象的生命周期、对象之间的关系，以及这些对象在用户空间中的表示。简化编程人员创建和管理对象的工作。

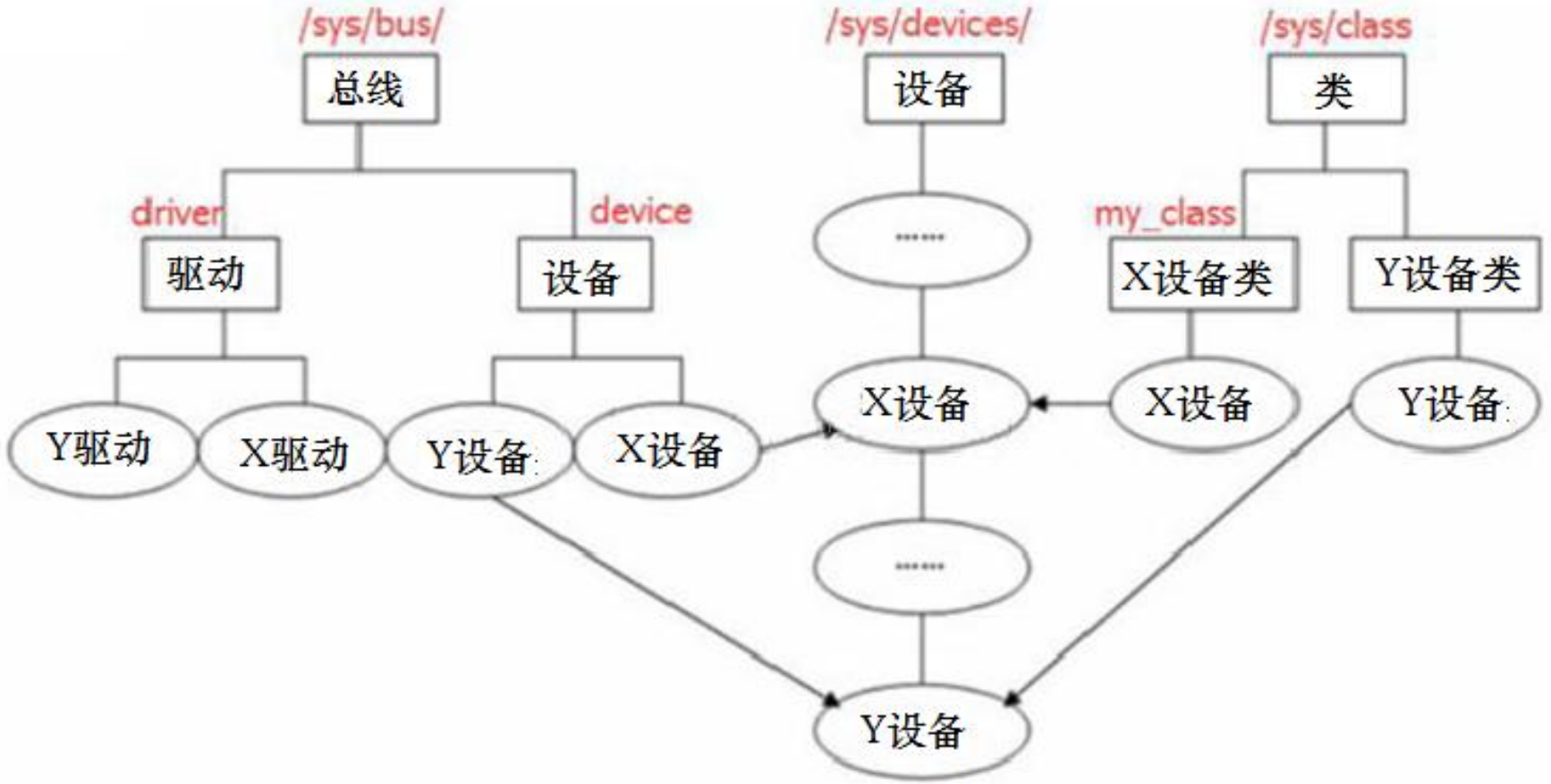
sys文件系统

- sys文件系统是一个类似于proc文件系统的特殊文件系统，用于将系统中的设备组织成层次结构，并向用户程序提供详细的内核数据信息。
- 也就是说，在用户态可以通过对sys文件系统的访问，来看内核态的一些驱动或者设备等信息，如图为sys文件系统目录。

```
[clj@localhost ~]$ cd /sys
[clj@localhost sys]$ ls
block  class  devices   fs          kernel  power
bus    dev    firmware  hypervisor  module
[clj@localhost sys]$ cd bus
[clj@localhost bus]$ ls
acpi         edac          ishtp         nvmem        scsi          usb-serial
clockevents  event_source  machinecheck  pci          serio         virtio
clocksource  hid           mdio_bus      pci_express  spi           workqueue
container    i2c           memory        platform     thunderbolt   xen
cpu          iio           node          pnp          usb
[clj@localhost bus]$ cd ../devices/
[clj@localhost devices]$ ls
breakpoint   msr         platform  software  tracepoint
LNXSYSTM:00  pci0000:00  pnp0      system    virtual
```

Linux设备模型

- Linux设备驱动模型，使用一系列抽象(面向对象里的类)，提供统一的设备管理视图
- 这些抽象包括：**总线、类、设备**和**设备驱动。**



Linux设备模型

- **Bus(总线)：**总线是CPU和一个或多个设备之间信息交互的通道。为了方便设备模型的抽象，所有的设备都应连接到总线上。

- **Class(分类)：**在Linux设备模型中，Class的概念类似面向对象中的类，它主要是集合具有相似功能或属性的设备，这样就可以抽象出一套可以在多个设备之间共用的数据结构和接口函数。从属于相同Class的设备的驱动程序，就无需重复定义这些公共资源，直接从Class中继承即可。

- **Device(设备)：**抽象系统中所有的硬件设备，描述它的名字、属性、从属的Bus、从属的Class等信息。

- **Device Driver(设备驱动)：**Linux设备模型用Driver，抽象硬件设备的驱动程序，它包含设备初始化、电源管理相关的接口实现。而Linux内核中的驱动开发，基本都围绕该抽象进行(实现所规定的接口函数)。

核心对象之kobject

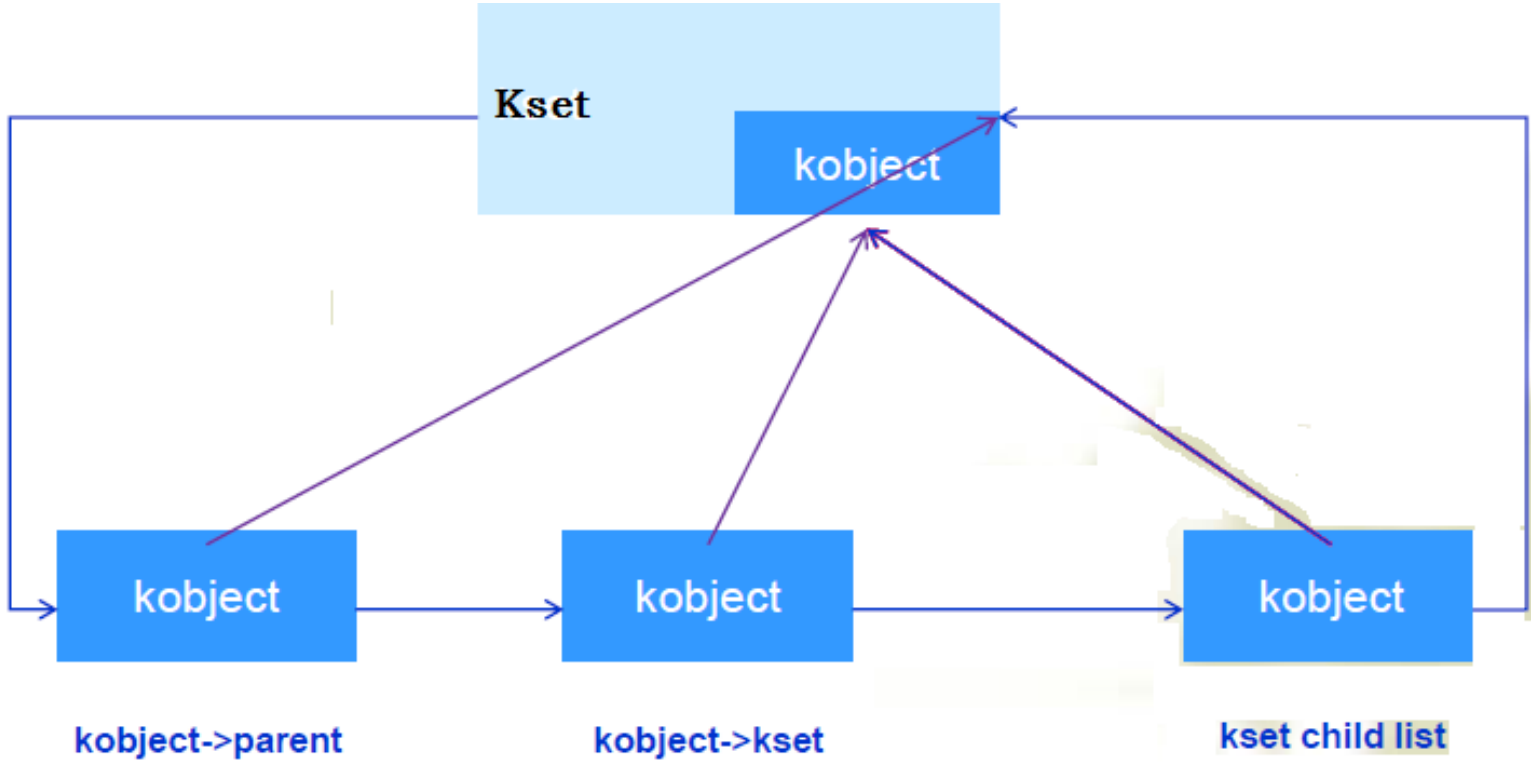
- kobject结构体是设备驱动模型底层的一个结构体，它是设备驱动模型下的所有对象的一个基本单元，是对设备驱动模型下所有对象抽象出来的共有的部分；
- kobject结构体提供了一些公共型的服务：对象引用计数、维护对象链表、对象上锁、对用户空间的表示。
- 设备驱动模型中的各种对象，其内部都会包含一个kobject，相当于面向对象中的总基类。

```
struct kobject {
  const char          *name;   //   对象的名字
  struct list_head     entry;
  struct kobject       *parent;
  struct kset          *kset;  //  用来指向父类对象的kset
  struct kobj_type     *ktype; //   指向一个kobj_type对象
  struct sysfs_dirent     *sd;
  struct kref          kref;    //   kobject的引用计数
  unsigned int state_initialized:; //   该对象是否被初始化了的状态标志位
... ...
};
```

定义在文件 include/linux/kobject.h中

对象集合体kset

- kset是嵌入相同类型结构的kobject的集合，可以把它看成是一个容器，可将所有相关的kobject对象聚集起来，
- 比如“全部的块设备”就是一个ksetkset结构关心的是对象的聚集与集合。
- 它与kobject的关系如图所示

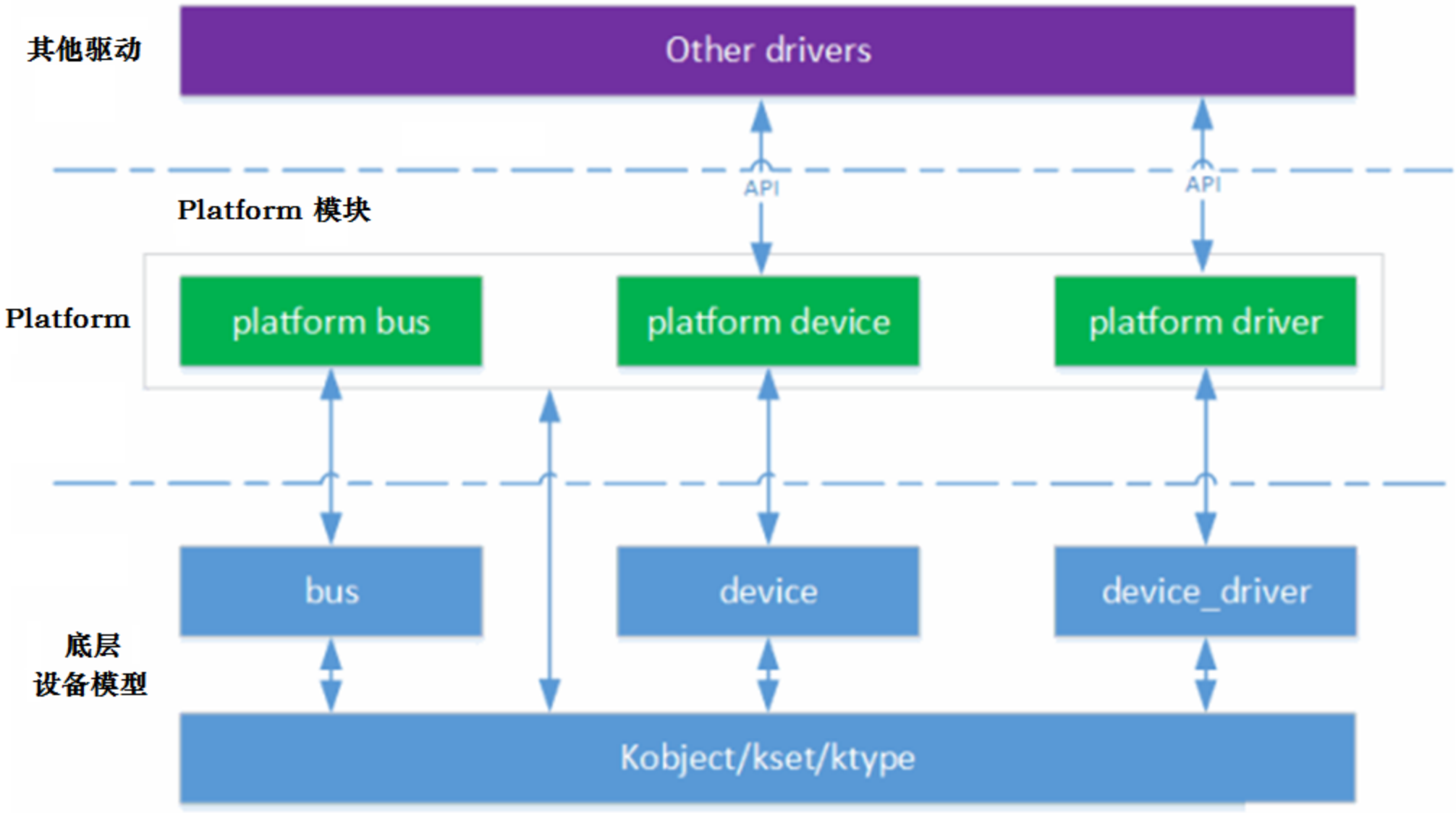



```
struct kset {
    struct list_head list;    //用来链接该目录下的所有kobject对象
    spinlock_t list_lock;
    struct kobject kobj;    //该kobject就是本目录对应的对象结构体
    const struct kset_uevent_ops *uevent_ops; // 指向一个用于处理集合中kobject
对象的热插拔操作的结构体
};
```

定义在文件 include/linux/kobject.h中

platform平台总线驱动模型

- 为解决驱动代码和设备信息耦合问题，
- Linux提出了platform bus (平台总线)的概念，
- 即使用虚拟总线将设备信息和驱动程序进行分离。

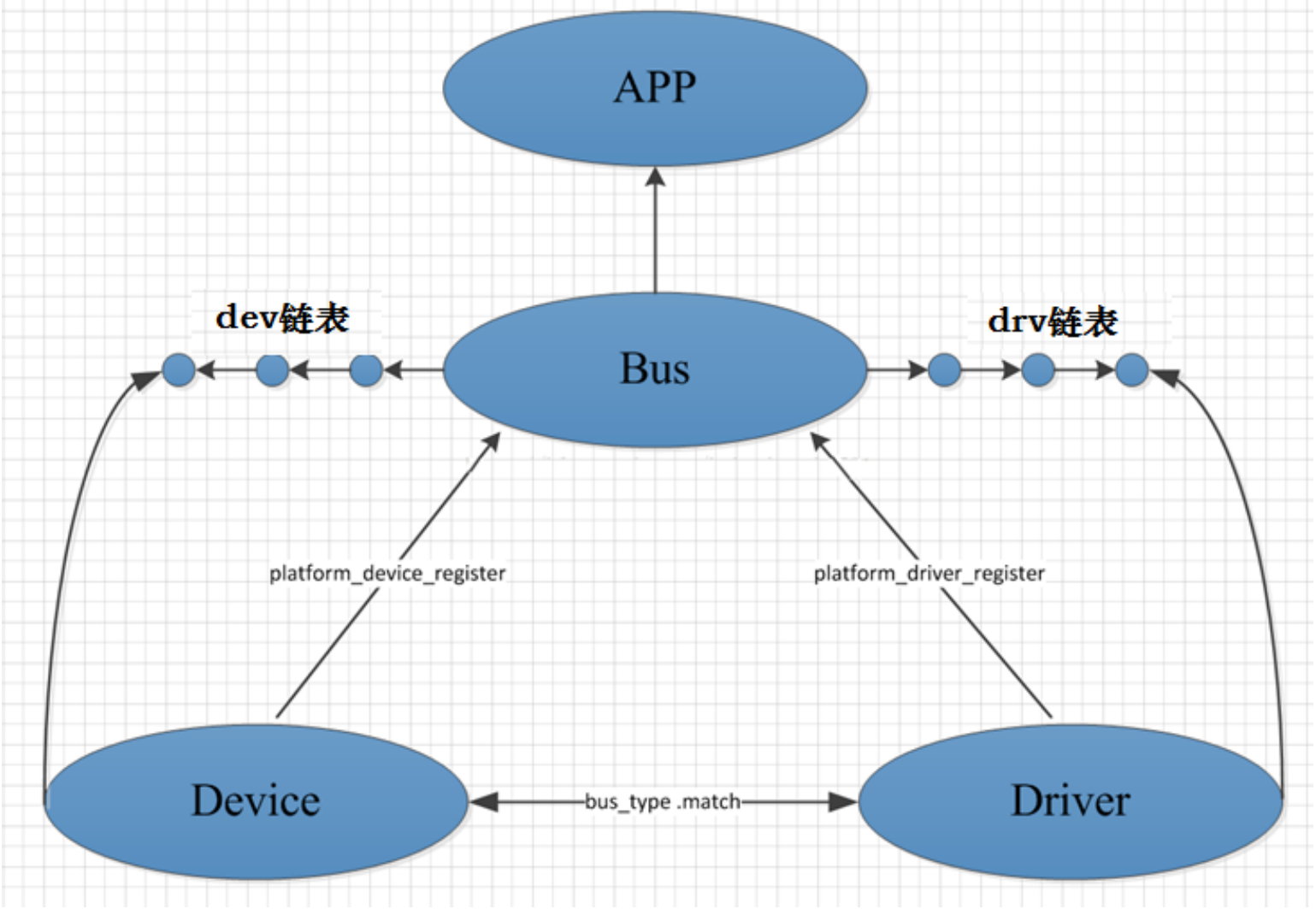


platform平台总线驱动模型

- 平台总线 会维护两条链表，分别管理设备和驱动，
- 　当一个设备被注册到总线上的时候，总线会根据其名字搜索对应的驱动，如果找到就将设备信息导入驱动程序并执行驱动；
- 　当一个驱动被注册到平台总线的时候，总线也会搜索设备。
- 总之，平台总线 负责将 设备信息和驱动代码 匹配，这样就可以做到驱动和设备信息的分离。
- 与传统的bus/device/driver机制相比，platform由内核进行统一管理，在驱动中使用资源，提高了代码的安全性和可移植性。
- 当硬件部分的时序变了或者芯片替换了，我们只需要修改硬件部分的代码，还有一部分代码是属于内核的稳定部分，是不用修改的，这就是一种通用的接口。
- 下面重点讲解platform模型。

platform平台总线驱动模型

- platform平台总线是一条虚拟总线。
- 其中platform_device为相应的**设备**，platform_driver为相应的**驱动**。



描述平台的结构platform_driver

- 可以看到platform_driver结构体中包含了probe和remove等相关操作，同时还内嵌了device_driver结构体。

```
1  struct platform_driver {
2          int (*probe)(struct platform_device *);
3          int (*remove)(struct platform_device *);
4          void (*shutdown)(struct platform_device *);
5          int (*suspend)(struct platform_device *, pm_message_t state);
6          int (*resume)(struct platform_device *);
7          struct device_driver driver;
8          const struct platform_device_id *id_table;
9          bool prevent_deferred_probe;
10 };
```

在include/linux/platform_device.h

描述设备驱动的结构device_driver

- 从中看到最后一个域指针p指向driver_private，
- 这是一个描述驱动私有数据的结构，

```
1  struct device_driver {
2          const char              *name;          // 驱动的名字
3          struct bus_type         *bus;           // 所属总线
4
5          struct module           *owner;
6          const char              *mod_name;      /* used for built-in modules */
7
8          bool suppress_bind_attrs;       /* disables bind/unbind via sysfs */
9
10         const struct of_device_id       *of_match_table;
11         const struct acpi_device_id     *acpi_match_table;
12
13         int (*probe) (struct device *dev);      // 驱动挂载的时候调用
14         int (*remove) (struct device *dev);     // 驱动卸载的时候调用
15         void (*shutdown) (struct device *dev);
16         int (*suspend) (struct device *dev, pm_message_t state);
17         int (*resume) (struct device *dev);
18         const struct attribute_group **groups;
19
20         const struct dev_pm_ops *pm;
21
22         struct driver_private *p;
23 };
```

在include/linux/device.h

描述设备的结构platform_device

- 其中struct resource在前面的I/O空间管理一讲已经讲过。

```
struct platform_device {
        const char      *name;          // 设备的名字这将代替device->dev_id，用作sys/device下显示的目录
        int             id;             // 设备id，用于给插入给该总线并且具有相同name的设备编号，如果只有一
        bool            id_auto;
        struct device   dev;            // 内嵌device结构
        u32             num_resources;  // 资源的数目
        struct resource *resource;      // 资源

        const struct platform_device_id *id_entry;

        /* MFD cell pointer */
        struct mfd_cell *mfd_cell;

        /* arch specific additions */
        struct pdev_archdata    archdata;
};
```

设备与驱动匹配的过程

- 设备与驱动如何匹配，打个比喻：
- a --红娘(总线)负责男方(设备)和女方(驱动)的撮合；
- b --男方找到红娘登记一下，看有没有合适的女方——设备或驱动的注册；
- c --红娘这时候就需要看看有没有匹配的女方—match 函数进行匹配，看name是否相同；
- d --如果没有找到匹配的，就告诉男方没有合适的对象，先等着，别急——设备和驱动会等待，直到匹配成功；
- e --终于遇到匹配的结婚后，男方就向女方交代，他有那些资源(如存款，房子等)(struct resource *resource)，
- 女方于是去拿钱买车，买衣服 (int(*probe) (struct platform_device*) 匹配成功后驱动执行的第一个函数)。

4 字符设备驱动程序

字符设备基础

- 字符设备是指只能一个字节一个字节进行读写操作的设备，不能随机读取设备中的某一数据、读取数据要按照先后数据。
- 字符设备是面向流的设备，常见的字符设备有：鼠标、键盘、串口、控制台和LED等。
- 一般每个字符设备或者块设备都会在/dev目录下对应一个设备文件。
- Linux用户层程序通过设备文件，来使用驱动程序操作字符设备或块设备。

如何描述字符设备

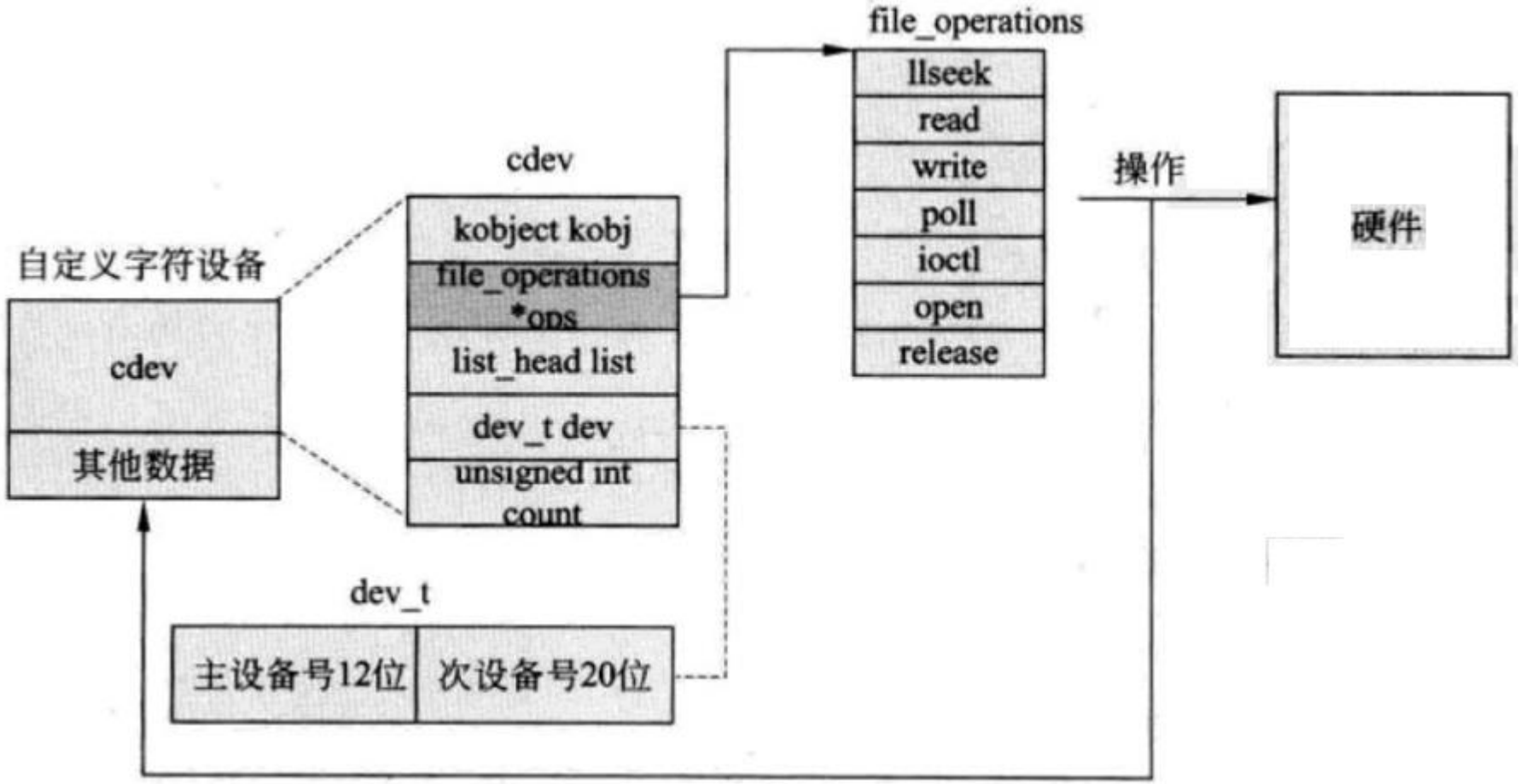
- Linux内核中使用struct cdev来表示一个字符设备
- 其中最关键的是file_operations结构，它是实现字符设备的操作集。

```
<include/linux/cdev.h>

struct cdev {
    struct kobject kobj;                    //内嵌的内核对象.
    struct module *owner;                   //该字符设备所在的内核模块（所有者）的对象指针，一般为THIS_MODULE主要用于模块计数
    const struct file_operations *ops;      //该结构描述了字符设备所能实现的操作集（打开、关闭、读/写、...），是极为关键的一个结构体
    struct list_head list;                  //用来将已经向内核注册的所有字符设备形成链表
    dev_t dev;                              //字符设备的设备号，由主设备号和次设备号构成（如果是一次申请多个设备号，此设备号为第一
个）
    unsigned int count;                     //隶属于同一主设备号的次设备号的个数
    ...
```

cdev与file_operations的关系图

- 每个字符设备，都有一个描述字符设备操作集的file_operations数据结构，
- 它与cdev的关系图如图所示
- 那么file_operations的原型到底是什么样的



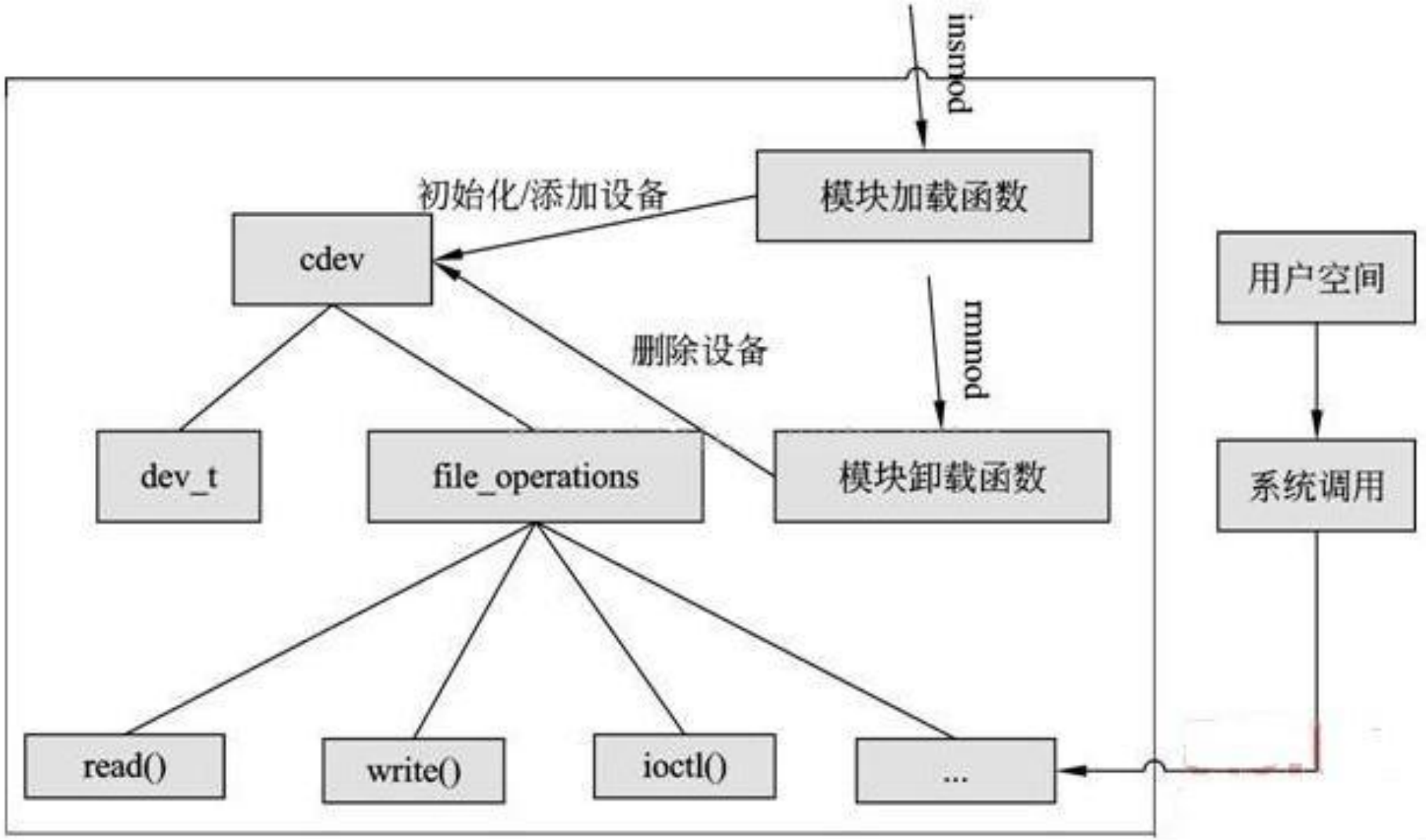
file_operations原型

- file_operations数据结构位于include/linux/fs.h中，其原型为：

```
struct file_operations {
        struct module *owner;
        loff_t (*llseek) (struct file *, loff_t, int);
        ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
        ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
        ssize_t (*read_iter) (struct kiocb *, struct iov_iter *);
        ssize_t (*write_iter) (struct kiocb *, struct iov_iter *);
        int (*iterate) (struct file *, struct dir_context *);
        int (*iterate_shared) (struct file *, struct dir_context *);
        __poll_t (*poll) (struct file *, struct poll_table_struct *);
        long (*unlocked_ioctl) (struct file *, unsigned int, unsigned long);
        long (*compat_ioctl) (struct file *, unsigned int, unsigned long);
        int (*mmap) (struct file *, struct vm_area_struct *);
        unsigned long mmap_supported_flags;
        int (*open) (struct inode *, struct file *);
        int (*flush) (struct file *, fl_owner_t id);
        int (*release) (struct inode *, struct file *);
        int (*fsync) (struct file *, loff_t, loff_t, int datasync);
        int (*fasync) (int, struct file *, int);
        int (*lock) (struct file *, int, struct file_lock *);
        ssize_t (*sendpage) (struct file *, struct page *, int, size_t, loff_t *, int);
```

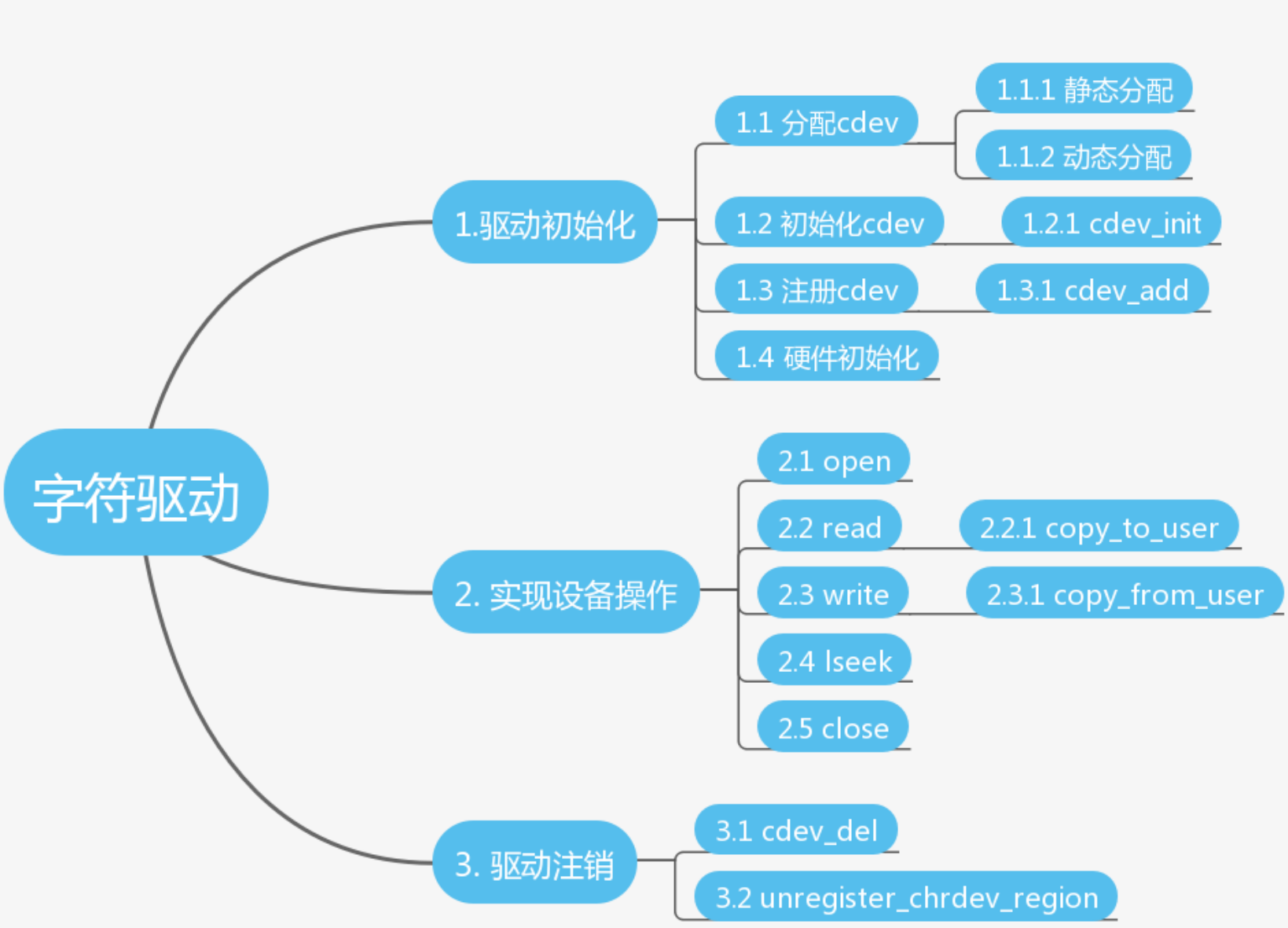
字符设备驱动框架

- 在Linux内核代码中：
- 1.使用 cdev结构体，来抽象一个字符设备；
- 2.通过一个 dev_t类型的设备号，来确定字符设备唯一性；
- 3通过 file_operations类型的操作方法集，来定义字符设备提供给VFS的接口函数。
- 如图所示。



编写字符设备驱动的步骤

- 如图所示
- 编写字符设备驱动分为三大步骤：
- 1.驱动的初始化
- 2.实现设备的操作
- 3.驱动的注销



字符设备驱动接口函数

- 其中调用的接口函数功能如下：

函数名	功能
cdev_alloc（）	动态申请（构造）cdev内存（设备对象）
cdev_init（）	初始化cdev的成员，并建立cdev和file_operations之间关联起来
cdev_add（）	注册cdev设备对象，也就是添加到系统字符设备列表中
cdev_del（）	将cdev对象从系统中移除（注销）
cdev_put（）	释放cdev内存

设备号的申请和释放

- 一个字符设备或块设备都有一个主设备号和一个次设备号。
- 主设备号用来标识与设备文件相连的驱动程序，用来反映设备类型。
- 次设备号被驱动程序用来辨别操作的是哪个设备，用来区分同类型的设备。

宏或者函数名	功能
MAJOR宏	从设备号中提取主设备号
MINOR宏	从设备号中提取次设备号
MKDEV宏	将主、次设备号拼凑为设备号
register_chrdev_region（）函数	静态申请设备号
alloc_chrdev_region（）	动态申请设备号
unregister_chrdev_region()	释放设备号

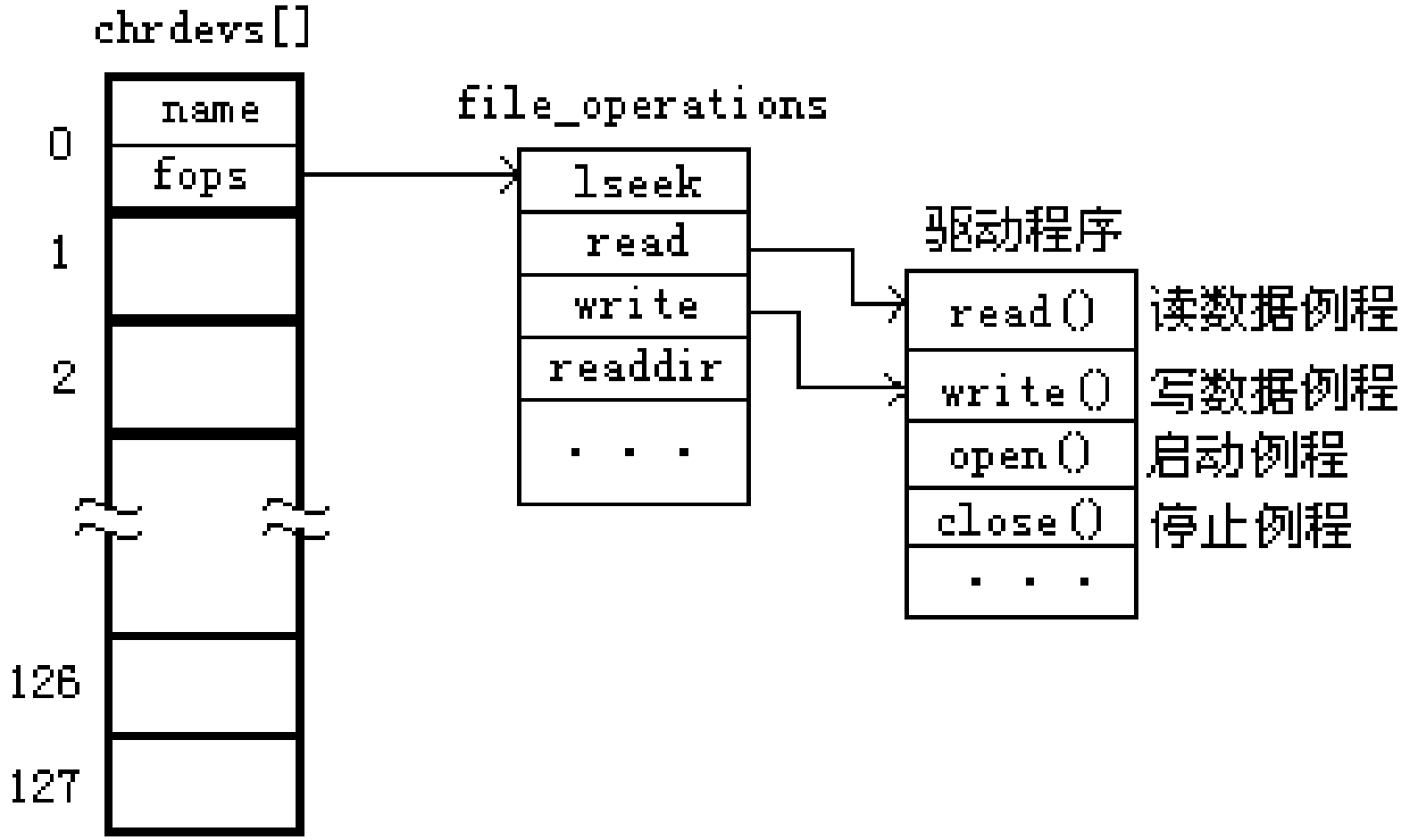
字符设备结构

- 在Linux中，字符设备是用 字符设备结构 char_device_struct来描述的。
- 为方便管理上，系统维护了一个数组chrdevs[]，数组每一项都代表一个字符设备。
- 在文件linux/fs/char_dev.c中定义的 char_device_struct的数据结构及数组chrdevs [ ]

```
static struct char_device_struct {
        struct char_device_struct *next;
        unsigned int major;
        unsigned int baseminor;
        int minorct;
        char name[64];
        struct cdev *cdev;              /* will die */
} *chrdevs[CHRDEV_MAJOR_HASH_SIZE];
```

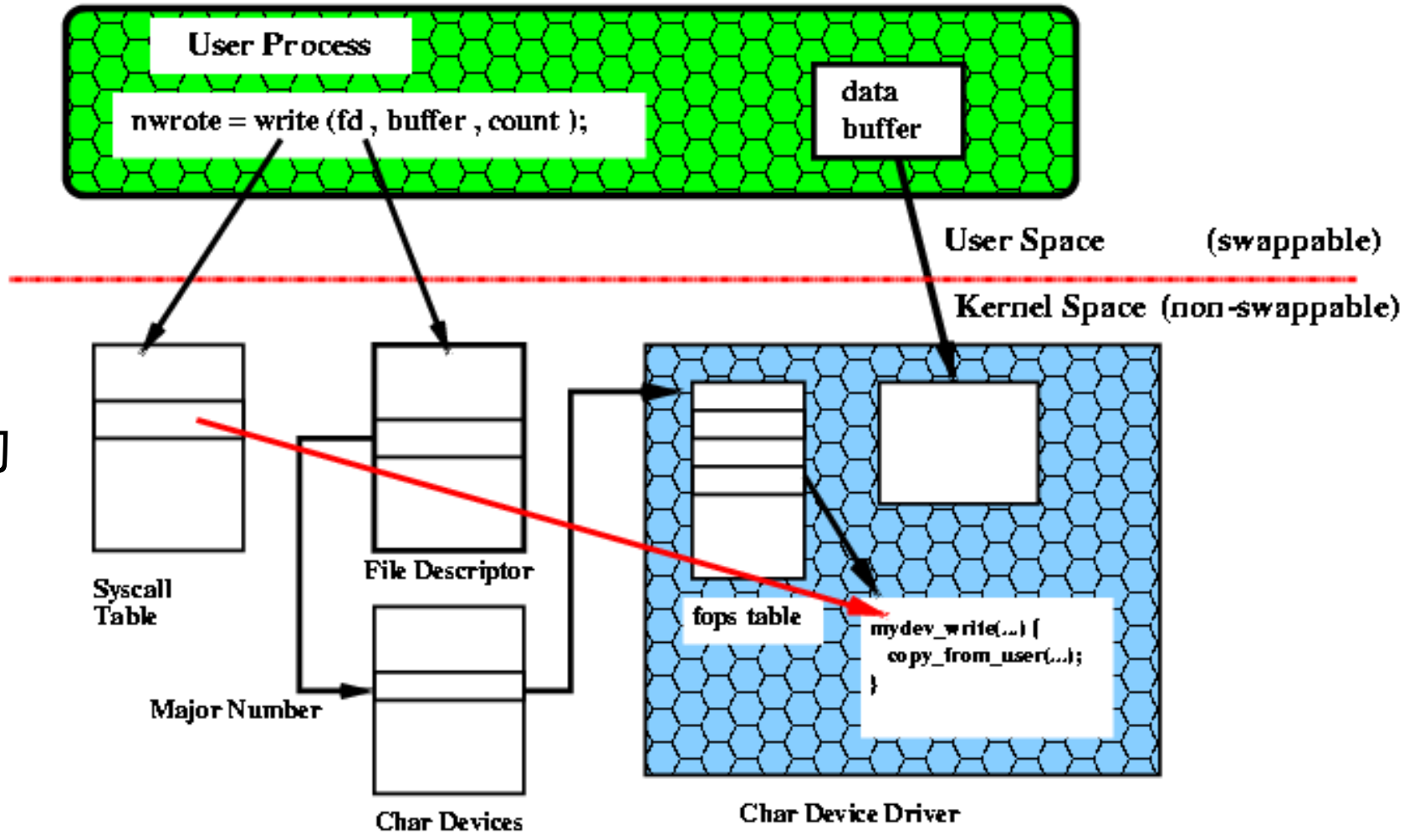
字符设备驱动程序的注册

- char_device_struct结构中的域cdev中的fops是指向文件操作函数集结构的指针。
- 每个注册的驱动的程序在chrdevs表中都有一项。



从系统调用到驱动程序

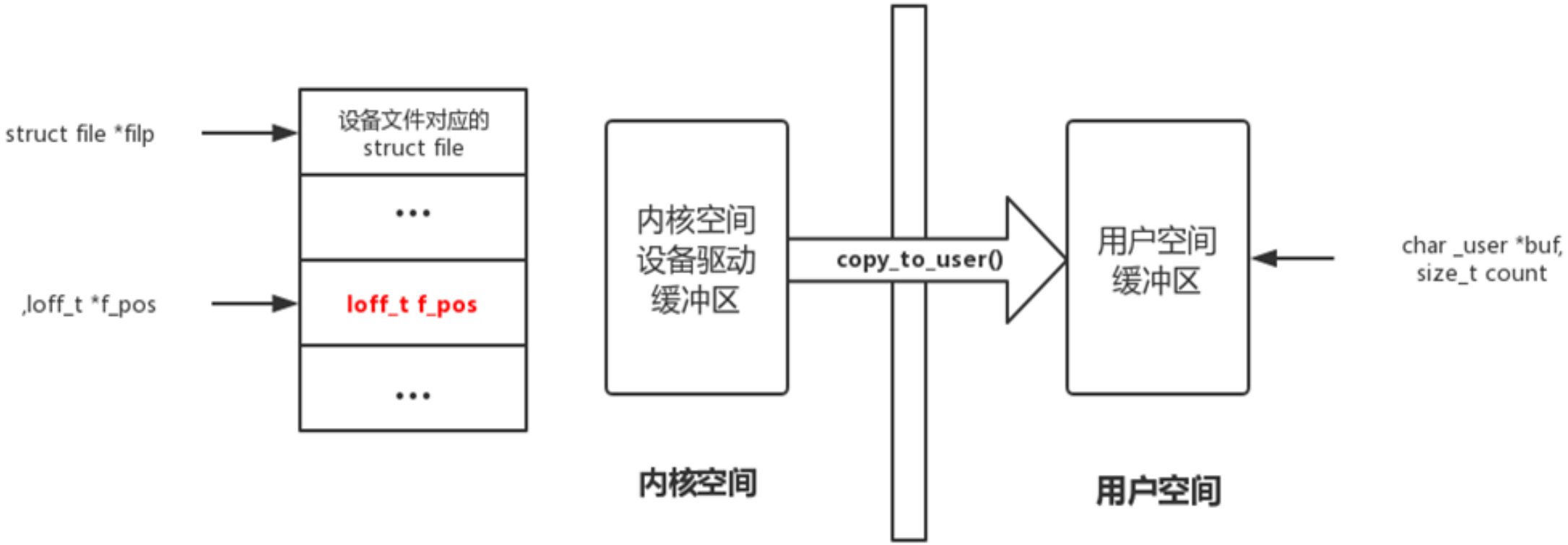
- 如图所示，当用户进程在调用write系统调用时，则陷入内核，首先查系统调用表，找到write系统调用的服务例程总入口。
- 如何找到你自己写的驱动程序mydev_write呢。
- 当我们打开文件时，open()的第一个参数是设备文件名，文件描述符fd就与这个设备文件关联起来了。
- 因此，通过主设备号，在字符设备表中就可以找到对应的驱动程序的入口函数了。



用户空间与内核空间数据的传送

- 当调用读函数时，通过内核的copy_to_user()函数把内核空间缓冲区中的数据拷贝到用户空间的缓冲区。

(*read)(struct file *filp, char _user *buf,size_t count,loff_t *f_pos);

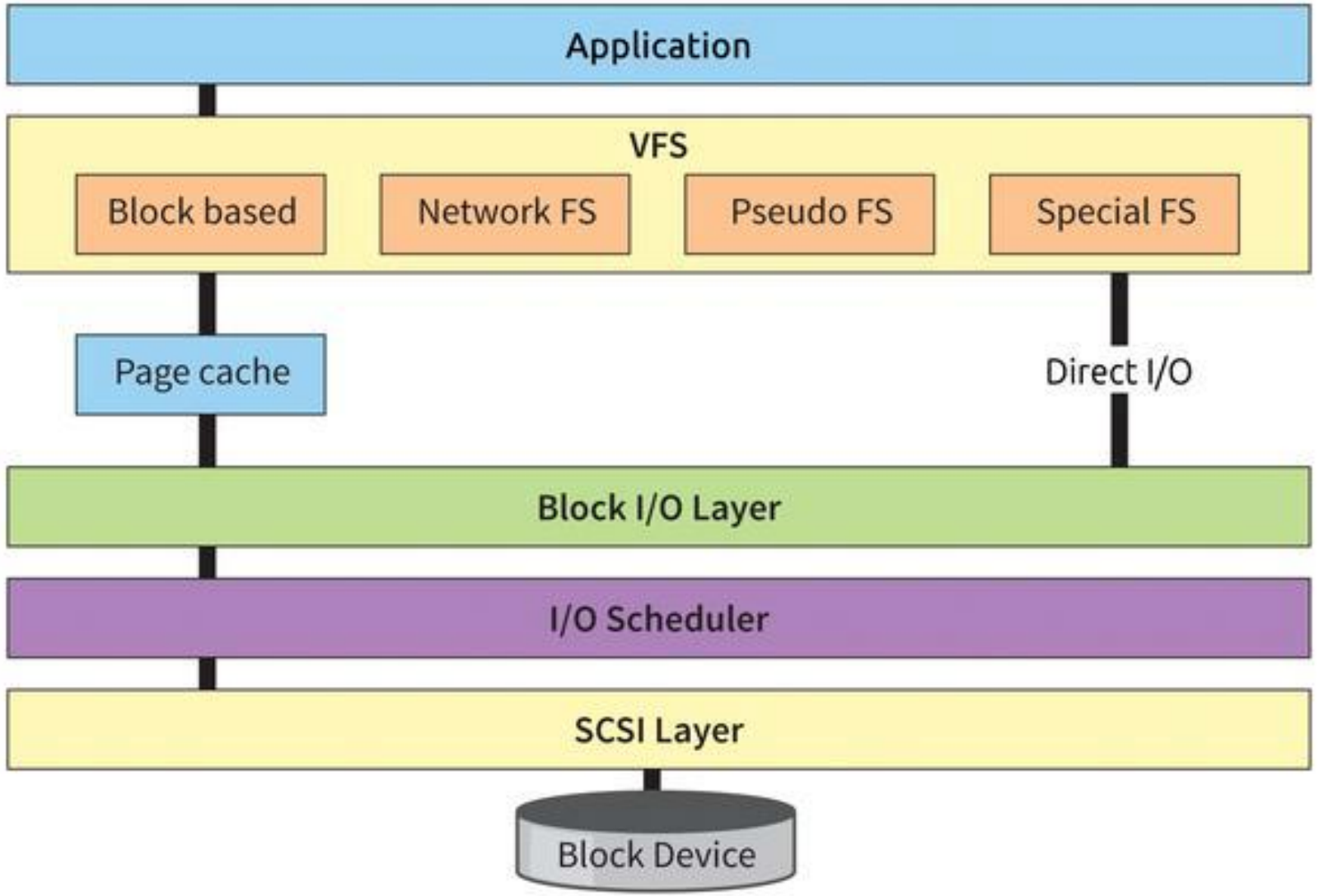


内容导航：

5 块设备驱动程序

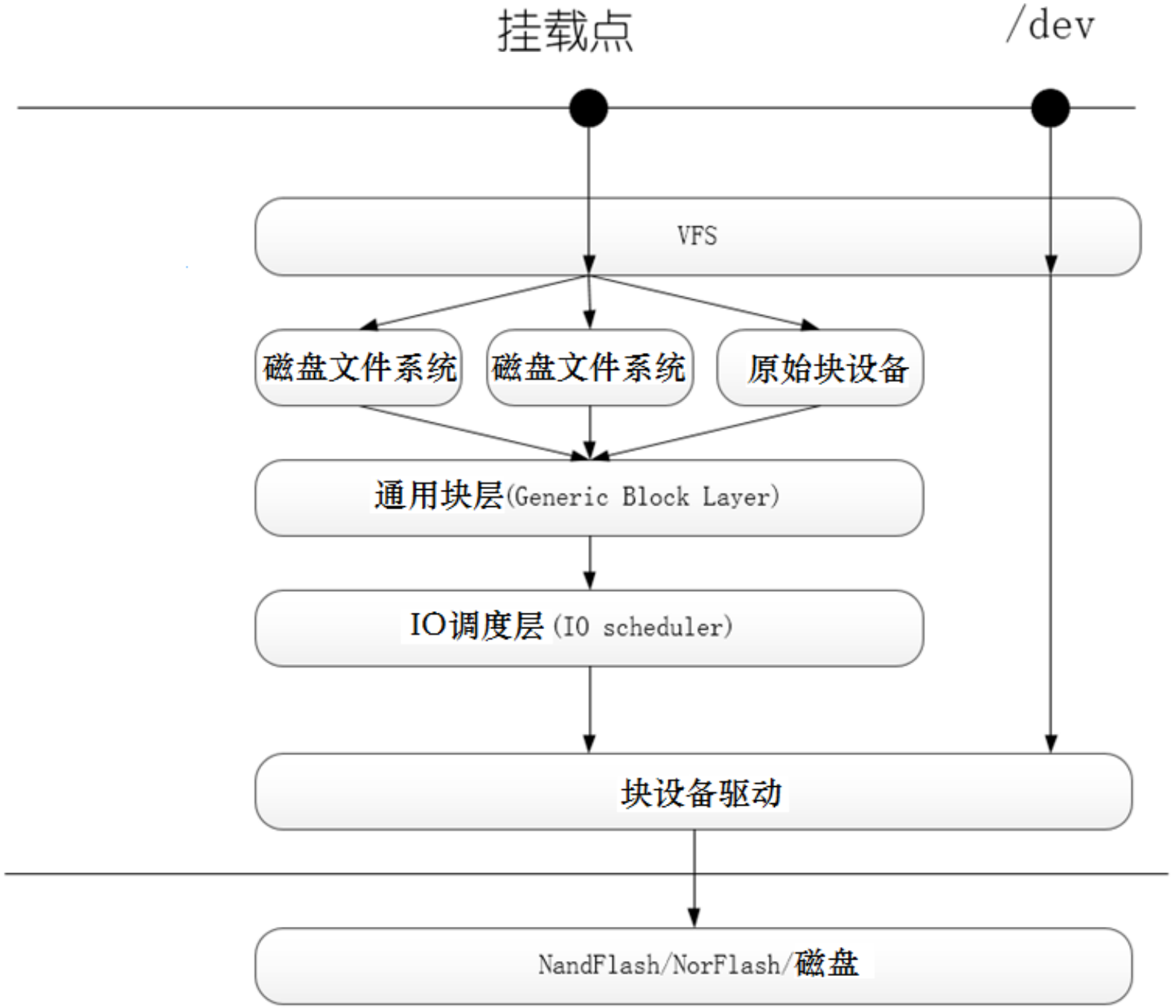
块驱动程序

- 如图所示，块驱动程序提供了对面向块的设备的访问，这种设备以随机访问的方式传输数据，并且数据总是具有固定大小的块。
- 块设备和字符设备的区别：
- 　块设备上可以mount文件系统，而字符设备是不可以的；
- 　相比操作字符设备，数据经过块设备，需要多经历一个数据缓冲层(buffer cache)机制。



块驱动模型

- 如图所示 Linux中的块设备模型示意图，
- 应用层程序有两种方式访问一个块设备：
- **通过/dev目录** 和 **文件系统挂载点**，
- 前者和字符设备一样，通常用于配置，
- 后者就是我们mount之后，通过文件系统直接访问一个块设备了。



块驱动模型

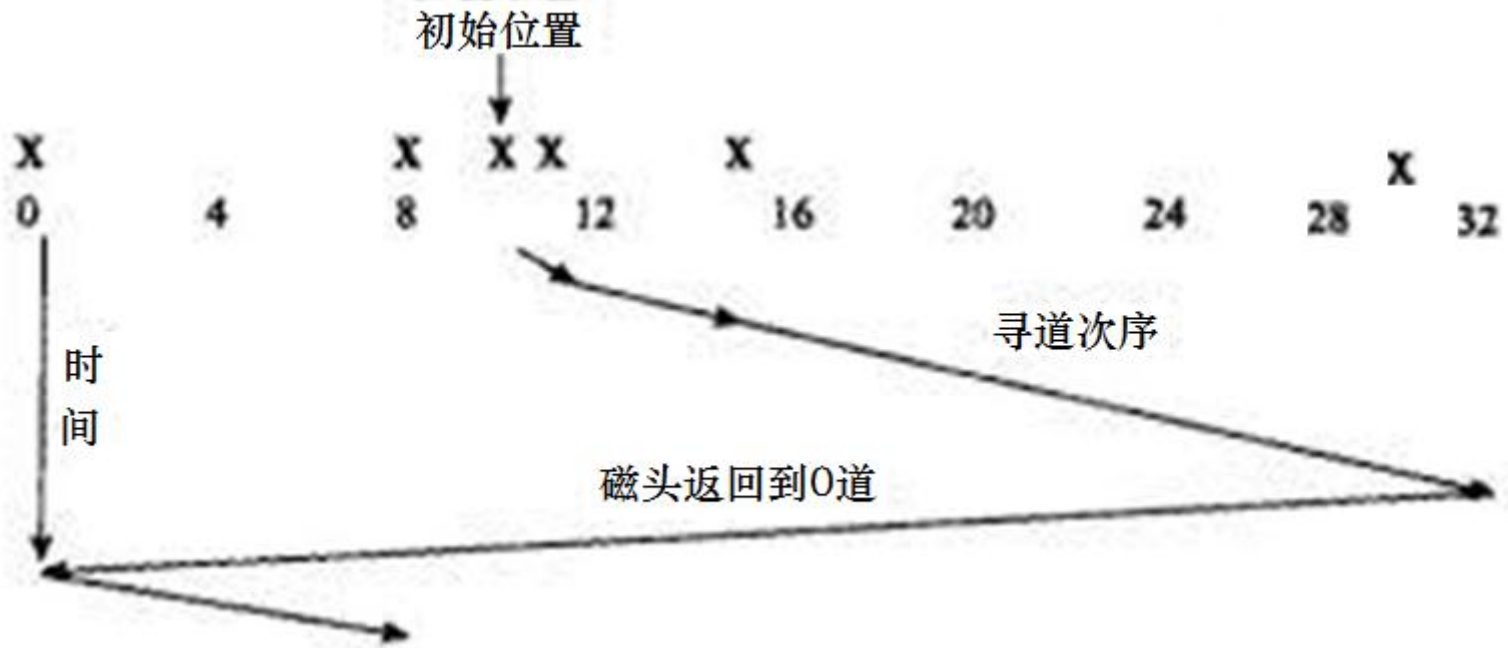
- 1.read()系统调用最终会调用一个适当的VFS函数(read()-->sys_read()-->vfs_read())，将文件描述符fd和文件内的偏移量offset传递给它。
- 2.VFS会判断这个系统调用的处理方式，如果访问的内容已经被缓存在缓冲区中，就直接访问，否则从磁盘中读取。

- 3.为了从物理磁盘中读取，内核依赖映射层mapping layer，即前图中的磁盘文件系统。
- 1)确定该文件所在文件系统的块的大小，并根据文件块的大小计算所请求数据的长度。本质上，文件被拆成很多块，因此内核需要确定请求数据所在的块
- 2)映射层调用一个具体的文件系统的函数，这个层的函数会访问文件的磁盘节点，然后根据逻辑块号确定所请求数据在磁盘上的位置。

- 4.内核利用通用块层(generic block layer)启动I/O操作，来传达所请求的数据，通常，一个I/O操作只针对磁盘上一组连续的块。
- 5.I/O调度程序根据预先定义的内核策略，将待处理的I/O进行重排和合并
- 6.块设备驱动程序向磁盘控制器硬件接口发送适当的指令，进行实际的数据操作

I/O调度

- I/O调度器的总体目标，是希望让磁头能够总是往一个方向移动，移动到底了再往反方向走。
- 这恰恰就是现实生活中的电梯模型，所以I/O调度器也被叫做电梯(elevator)调度，而相应的算法也就被叫做电梯算法。



- 我们知道，磁盘的读写是通过机械性的移动磁头来实现的，理论上磁盘设备满足块设备的随机读写的要求，
- 但是出于节约磁盘，提高效率的考虑，我们希望当磁头处于某一个位置的时候，一起将最近需要写在附近的数据写入，而不是这写一下，那写一下然后再回来，
- I/O调度就是将上层发下来的I/O请求的顺序，进行重新排序以及对多个请求进行合并，这样就可以实现上述的提高效率、节约磁盘的目的。
- Linux内核中提供了下面的几种电梯算法来实现I/O调度。

I/O调度算法

算法名	描述
No-op I/O scheduler	先来先服务算法：只实现了简单的FIFO，只进行最简单的合并，比较适合基于Flash的存储
Anticipatory I/O scheduler	预测调度算法：推迟IO请求(大约几个微秒)，以期能对他们进行排序，获得更高效率
Deadline I/O scheduler	最后期限调度算法：试图把每次请求的延迟降到最低，同时也会对BIO重新排序，特别适用于读取较多的场合，比如数据库
CFQ I/O scheduler	公平调度算法：为系统内所有的任务分配均匀的IO带宽，提供一个公平的工作环境，在多媒体环境中，能保证音视频及时从磁盘中读取数据，是当前内核默认的调度器。

如何指定或者改变调度算法

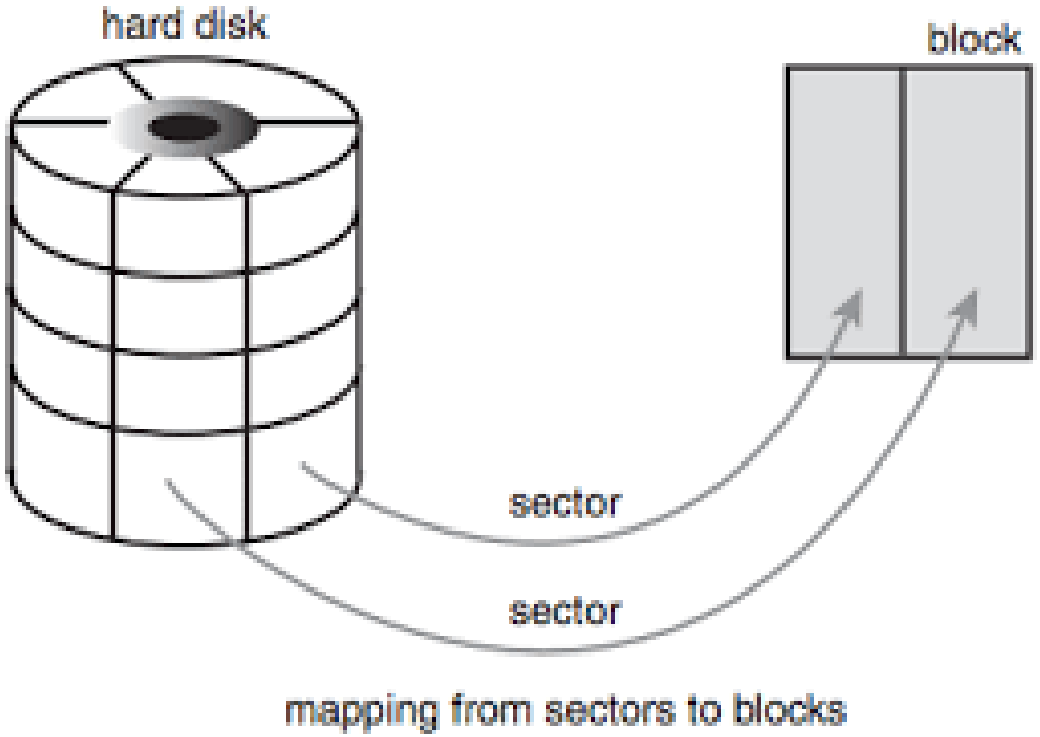
内核传参的方式指定使用的调度算法

kernel elevator=deadline

使用命令改变内核调度算法

echo SCHEDULER >/sys/block/DEVICE/queue/scheduler

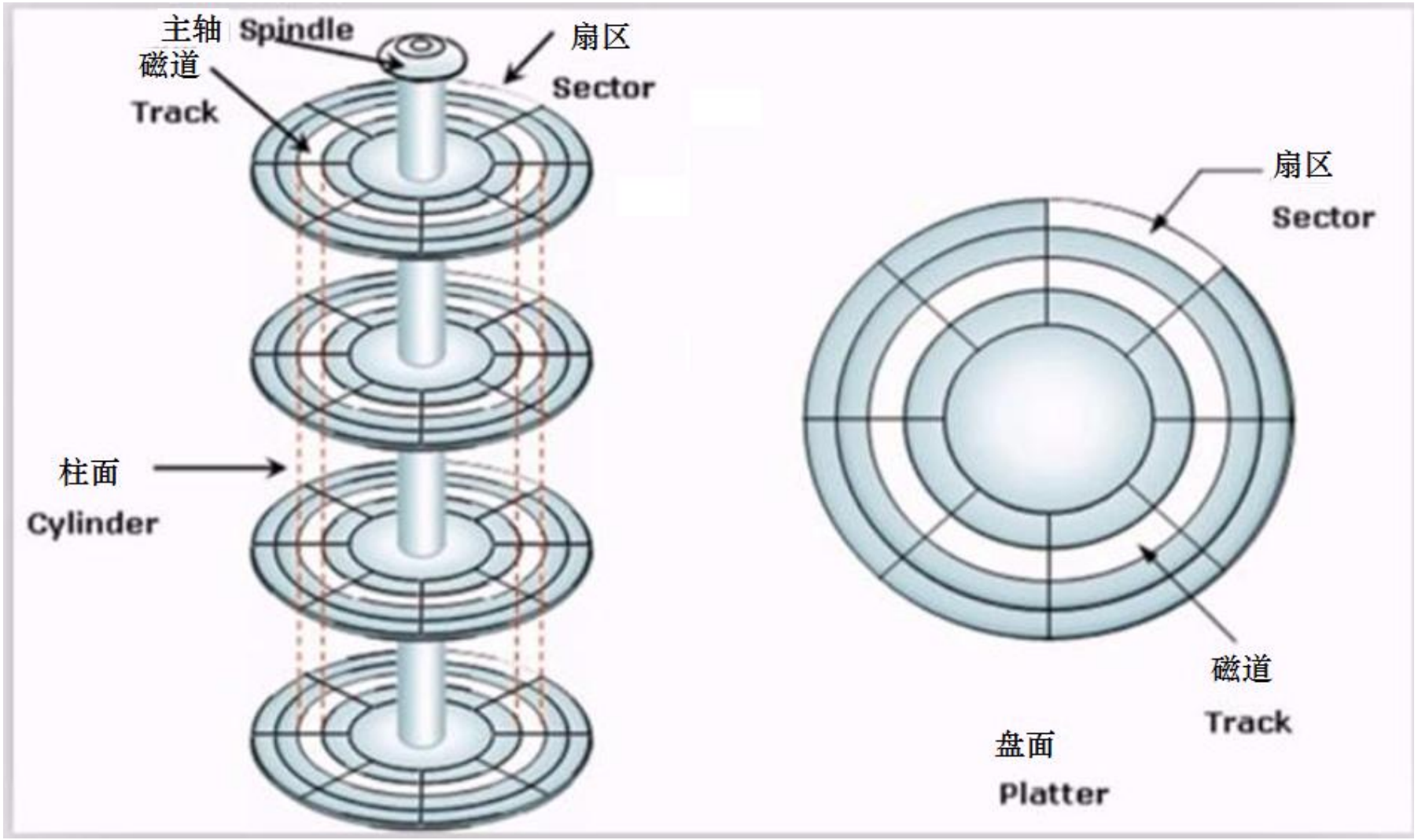
块-文件系统的读写单位



- 文件系统的读写单位是块，
- 一个块的大小是2的n次方个扇区，比如1k，2k, 4k, 4M 等，如ext4文件系统的block缺省是4k。
- block是VFS和文件系统传送数据的基本单位。

扇区-磁盘读写单位

- 扇区(sector):扇区是硬件(磁盘)上的最小的操作单位，是文件系统和块设备(硬件、磁盘)之间传送数据的单位。
- 一般一个扇区的大小是512字节。如果实际的设备的扇区不是512字节，而是4096字节(比如SSD)，
- 那么只需要将多个内核扇区，对应一个设备扇区即可。

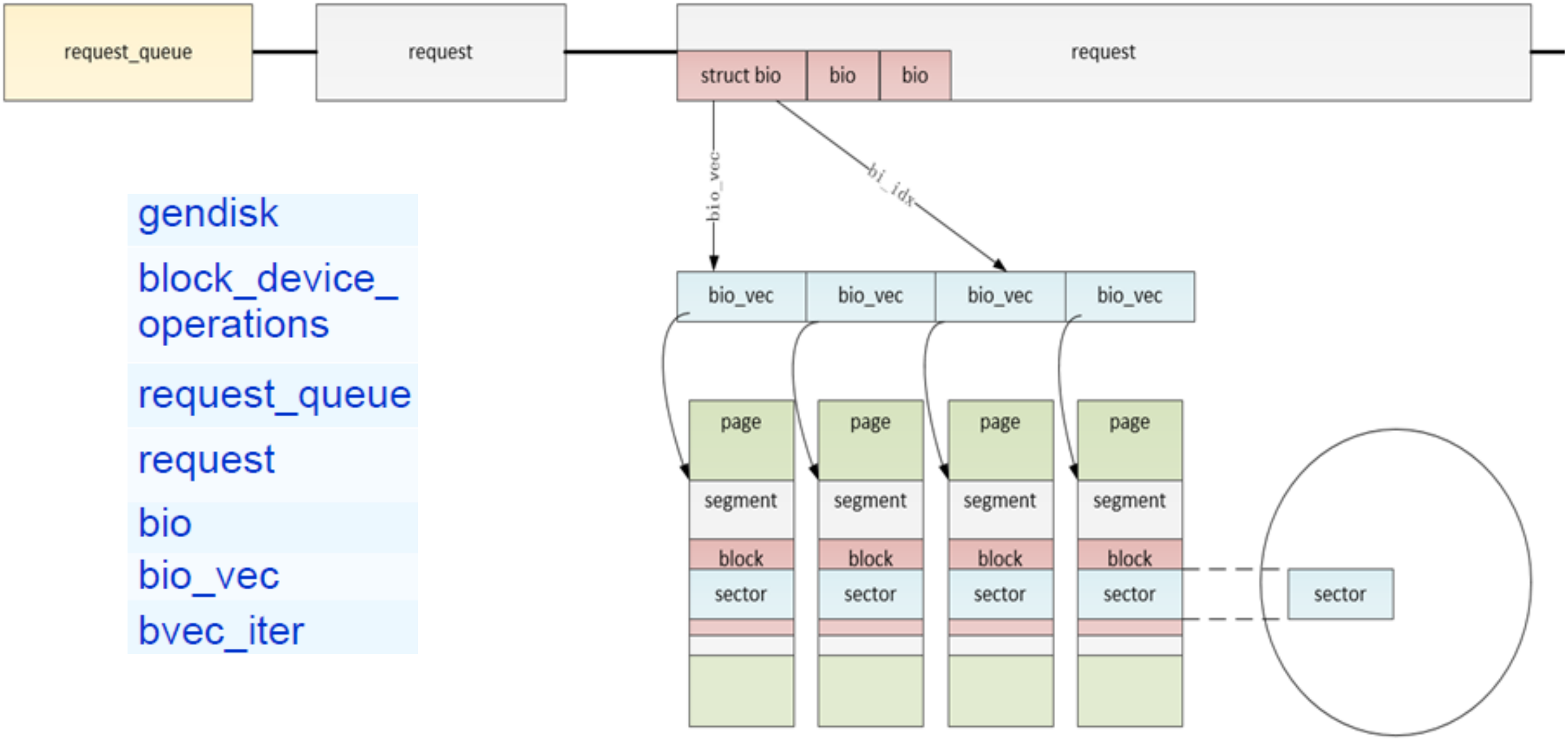


块设备驱动的核心结构

核心结构	描述
gendisk	磁盘描述符，是对一个物理磁盘或分区的描述
block_device_operations	描述磁盘的操作方法集，它之于gendisk，类似于file_operations之于cdev
request_queue	针对一个gendisk（磁盘）对象的所有请求的队列，是相应gendisk对象的一个域
request	表示经过IO调度之后的针对一个gendisk(磁盘)的一个"请求"，是request_queue的一个节点。多个request构成了一个request_queue
bio	表示应用程序对一个gendisk(磁盘)原始的访问请求，一个bio由多个bio_vec组成，多个bio经过IO调度和合并之后可以形成一个request。
bio_vec	描述的应用层准备读写一个gendisk(磁盘)时需要使用的内存页page的一部分，也就是"段segment"，多个bio_vec和bio_iter形成一个bio
bvec_iter	用于记录当前bio_vec被处理的情况，用于遍历bio。

核心结构之间的关系图

- 上面描述的这些核心数据结构，如图所示：



gendisk

block_device_operations

request_queue

request

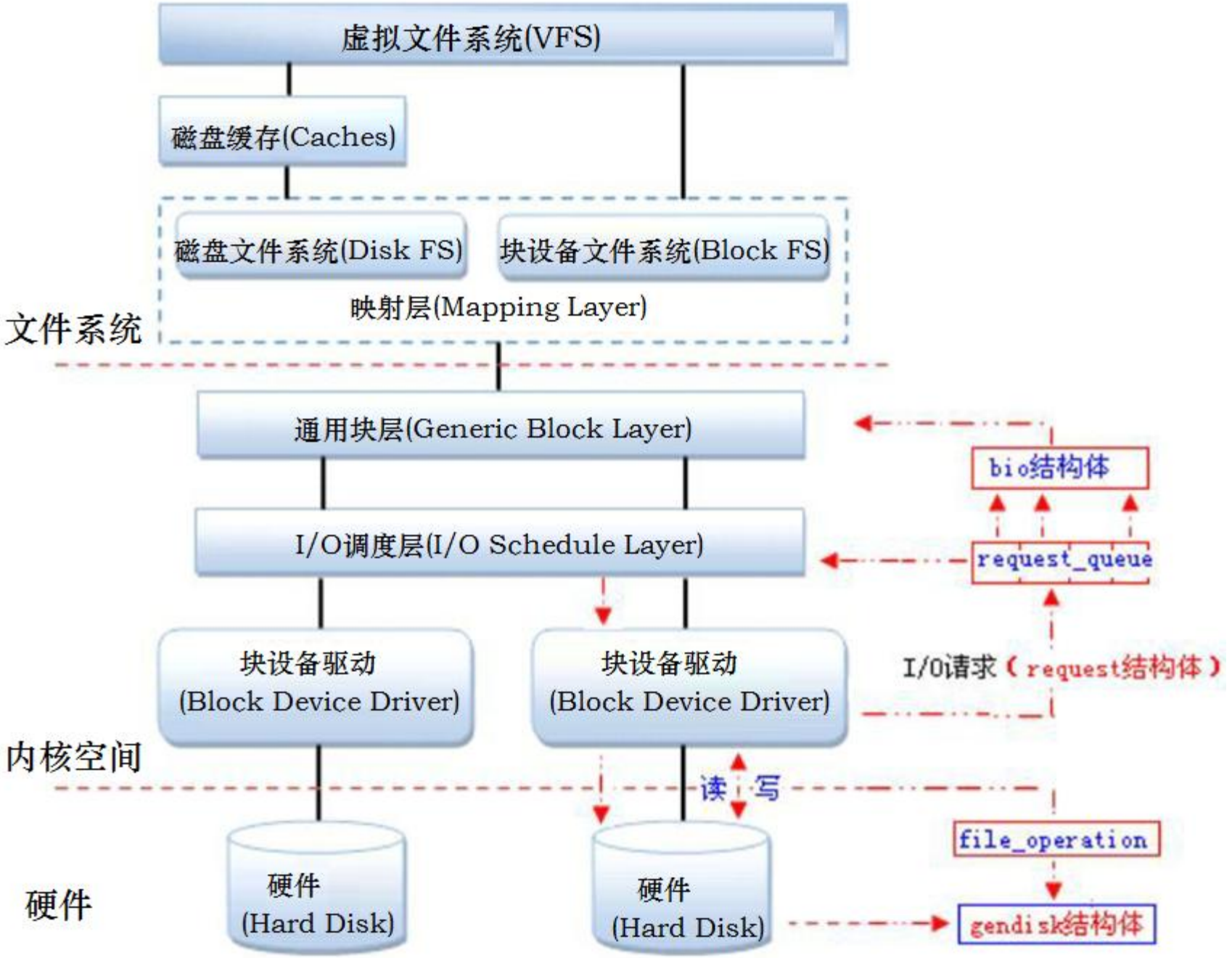
bio

bio_vec

bvec_iter

块设备驱动程序小结

- 1.VFS 层
- 2.缓存层
- 3.映射层
- 4.通用块层
- 5.I/O调度层
- 6.块设备驱动



课程思政

课程思政

富有远见，着眼未来，时时刻刻做好应对变化的准备

国内Linux系统的发展与国际上Linux系统的市场占有情况密切相关。

随着大数据与云计算等前沿技术的快速发展，越来越多的互联网公司开始构建自主控制与维护的云计算平台。具有开源与跨平台等属性的Linux系统，搭配采用Arm64芯片的计算平台，成为了这些互联网公司的首选技术方案。与此同时，Linux服务器端解决方案通过互联网企业迅速应用到了大数据与云计算的市场环境中。

但是，互联网企业使用Linux服务器端时，并未因采用Linux系统而形成典型的操作系统销售市场，专业的Linux系统厂商在服务器市场中还未形成较大的市场影响力。目前，国内海量的应用软件都是基于Windows系统的，因为该系统用户学习成本低、熟练程度高；而针对Linux系统，存在用户熟练程度低、对专业技术支持团队的依赖程度高、使用和维护成本高等问题。

富有远见，着眼未来，时时刻刻做好应对变化的准备

为了更好、更快地解决上述问题，亟须建立顺畅的产品服务情况与用户使用预期的沟通渠道，通过了解并满足用户针对操作系统在使用、维护等方面的多种需求，提升国产（基于Linux系统进行二次开发的）操作系统的整体性能。同时，亟须确定一个兼具稳定性和一致性的开发接口，以使开发Linux系统应用软件的代码可以跨平台落地，进而减少因操作系统不同而导致的应用软件重复开发与测试工作，最终形成基于较为成熟的国产操作系统的产业氛围。

我们应该富有远见，着眼未来，时时刻刻做好应对变化的准备，积极推进操作系统产品化坚决构建基于国产操作系统的产业氛围，围绕如何建立产品服务情况与用户使用预期的沟通渠道，以及如何确定兼具稳定性和一致性的开发接口做一些踏踏实实的工作。

课后练习题

1、什么是Linux设备驱动？请简述设备驱动程序在Linux内核中的作用。
2、如何理解Linux操作系统把设备纳入文件系统的范畴来管理？
3、请介绍Linux字符设备驱动是什么，有哪些常见的字符设备？
4、Linux块设备驱动是什么，请介绍常见的块设备。
5、Linux网络设备驱动有什么特点？
6、请对访问I/O内存方式和访问I/O端口方式做一个介绍。
7、请简述platform平台总线驱动模型。

谢 谢！

THANKS