《openEuler内核编程》

课程讲稿

第八章 第2讲

字符设备

软件所制

第八章 第2讲 字符设备

**学时：1**学时

**教学目的：**系统学习Linux设备驱动和字符设备驱动程序，深入了解Linux设备驱动和字符设备驱动程序原理。

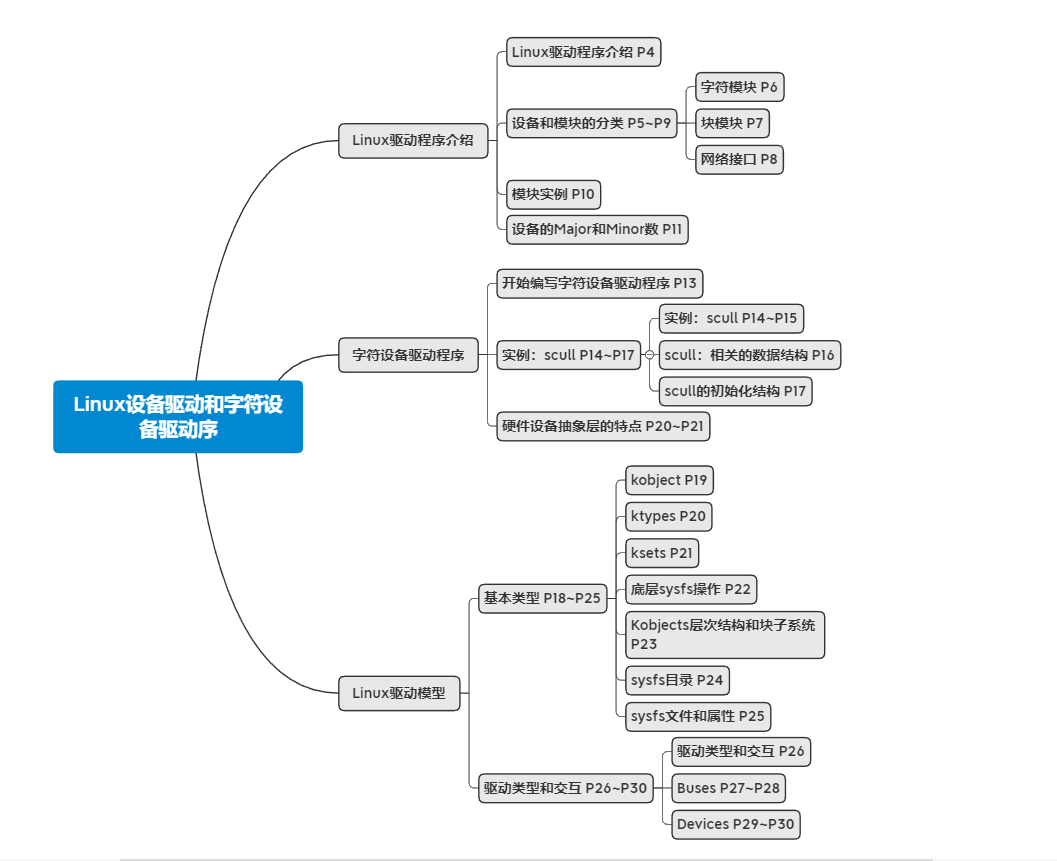
**课程时间线：**



**课外参考读物：**

《LINUX设备驱动程序（第三版）》

**知识框图：**



**PPT讲稿：**

1. 接下来是我们的第八章的第二节内容，在这一节中我们将会对Linux设备驱动进行分类，并学习其中的第一类设备驱动：字符设备驱动程序。

其实这里标题有点剧透对吧，分类中的第一类是字符设备驱动程序这一点就透露出来了。

2. 这里是本章的内容分节，我们这里是属于第二节的内容。

3. OK，这里是本节内容的一个划分。我们分为了三部分内容来讲解本节内容：Linux驱动程序介绍、字符设备驱动程序和驱动类型和交互。

那么我们先来看第一部分内容，Linux驱动程序的一个介绍。

4. 这里是一个抽象图，我们来看一下最上面是System call interface，操作系统调用接口，也就是上面是一个操作系统层面的调用；

最下面是各式各样的硬件设备，包括cpu，内存，磁盘，网络接口等等。

而中间就是我们要介绍的驱动程序，可以看到抽象化出来的驱动程序是各种各样的，分为不同的模块进行工作。那么我们现在来看一下其中的模块都分为什么类别。

5. 在知道驱动程序分类之前，我们先来看一下设备的分类，因为驱动程序的不同类型主要是根据不同的设备来进行区分的。Linux系统将设备分成三种基本类型，每个模块通常实现为其中某一类：字符模块、块模块或网络模块。然而这种将模块分成不同类型或类的分类方式并不是非常严格，程序员可以构造一个大的模块，在其中实现不同类型的设备驱动程序。然而，优秀的程序员通常还是为每个新功能创建一个不同的模块，从而实现良好的伸缩性和扩展性。接下来我们来依次介绍一下三种设备：

6. 字符（char）设备是个能够像字节流（类似文件）一样被访问的设备，由字符设备驱动程序来实现这种特性。字符设备驱动程序通常至少要实现open、 close、read和 write系统调用。字符终端（ //console）和串口（ dev/ttys0以及类似设备）就是两个字符设备，它们能够很好地说明“流”这种抽象概念。字符设备可以通过文件系统节点来访问，比如 dev/tty和/ devil0等。这些设备文件和普通文件之间的唯一差别在于对普通文件的访问可以前后移动访问位置，而大多数字符设备是一个个只能顺序访问的数据通道。然而，也存在具有数据区特性的字符设备，访问它们时可前后移动访问位置。例如，帧抓取器就是这样一个设备，应用程序可以用mmap或 seek访问抓取的整个图像。

7. 块设备和字符设备类似，块设备也是通过/dev目录下的文件系统节点来访问。块设备（例如磁盘）上能够容纳文件系统。在大多数Unix系统中，进行IO操作时块设备每次只能传输一个或多个完整的块，而每块包含512字节（或2的更高次幂字节的数据）. Linux可以让应用程序像字符设备一样地读写块设备，允许一次传递任意多字节的数据。因而，块设备和字符设备的区别仅仅在于内核内部管理数据的方式，也就是内核及驱动程序之间的软件接口，而这些不同对用户来讲是透明的。在内核中，和字符驱动程序相比，块驱动程序具有完全不同的接口。

8. 网络接口任何网络事务都经过一个网络接口形成，即一个能够和其他主机交换数据的设备。通常，接口是个硬件设备，但也可能是个纯软件设备，比如回环（loopback）接口。网络接口由内核中的网络子系统驱动，负责发送和接收数据包，但它不需要了解每项事务如何映射到实际传送的数据包。许多网络连接（尤其是使用TCP协议的连接）是面向流的，但网络设备却围绕数据包的传输和接收而设计。网络驱动程序不需要知道各个连接的相关信息，它只要处理数据包即可。由于不是面向流的设备，因此将网络接口映射到文件系统中的节点（比如/dev/ttyl）比较困难。Unx访问网络接口的方法仍然是给它们分配一个唯一的名字（比如eth0），但这个名字在文件系统中不存在对应的节点。内核和网络设备驱动程序的通信，完全不同于内核和字符以及块驱动程序之间的通信，内核调用一套和数据包传输相关的函数而不是read、 write等。

9. 这里我们介绍完了第一种设备分类方式：分为字符模块、块模块和网络模块。

除此之外，还有另外一种划分驱动程序模块类型的方法。一般而言，某些驱动程序类型同内核用来支持某种给定类型设备的附加层一起工作。比如，通用串行总线（USB）模块、串行模块、SCSI模块，等等。每个USB设备由一个USB模块驱动，而该USB模块和USB子系统一同工作，但设备本身在系统中表现为一个字符设备（比如USB串口）、一个块设备（比如USB存储卡读取器），或者一个网络设备（比如USB以太网接口）.

而在这里，我们介绍驱动程序按照两种分类的混搭：我们之后会分别介绍字符设备驱动程序、块设备驱动及USB模块。我们在本课之中接下来介绍完驱动模块的基础知识之后就会开始讲解字符设备驱动程序相关的内容。

10. 接下来我们介绍的是无论是什么模块都需要完成的一些函数：

有两个关键的函数就是“\_\_init”和“\_\_exit”标识的函数，它们分别代表了加载模块和卸载模块时所进行的调用，在这些函数里一般进行申请空间、预赋值等操作，然后在退出函数中进行申请空间的释放。代码中例子则是以显示加载信息与显示退出信息来标志这些内容。

11. 由于不同的设备可以使用同一个驱动程序，于是我们需要一个命令来查看。这个命令就是ls –l，可以使用此命令来查看驱动模块负责的设备列表，同时获取这些设备的详细信息。

12. 接下来是本节课的第二部分内容：字符设备驱动程序。之前我们介绍了驱动程序的分类，其中第一类就是字符设备驱动程序，我们现在来详细看一下相关的内容。

13. 之前我们介绍了字符驱动外部表现出来的实现方式，那么现在我们从编写一个设备驱动程序的角度，从内部来看一下相关的内容：

作为一个字符设备驱动程序，想要让其发挥作用，首先我们需要根据设备号来注册设备，然后进行编写内容时首先定义、声明设备驱动程序特定数据结构，主要包括内存指针、嵌入字符设备类型和同步互斥变量。然后编写设备初始化代码，然后编写设备驱动程的其他操作，最后编写设备初始化/释放代码。

14. 在这里我们介绍一个scull的实例，首先我们先介绍一些scull是什么：

这里是scull 设备的结构体的一个定义，这里的scull的含义是指scull（simple character utility for loading localities，区域装载的简单字符工具）是一个操作内存区域的字符设备驱动程序，这片内存区域就相当于一个字符设备。

15. 首先是有基本的数据单元，然后用链表连接起来这些基本的单元就组成了一个集合。主要是用于方便简单的内存分配。

16. 这里是前面我们所说过的一些设备驱动程序特定数据结构，这里定义了scull\_qset和scull\_dev两个数据结构，申请了一系列会用到的空间，然后最后定义了一个信号量用于读写互斥。

17. 在这里初始化file\_operations 结构scull\_fops，实现申请空间后的数据的初始化赋值操作。下面的两个函数就是我们之前提到的模块加载入口和卸载模块出口。

18. 接下来我们介绍本节课的第三部分Linux驱动模型，在这一部分内容中我们会介绍基本类型和驱动类型和交互。

基本类型分为Kobjects, Ksets, and Subsystems三种，我们接下来依次介绍这三种。

19. 首先是kobject，struct kobject 主要用于:引用计数、表达sysfs虚拟文件系统中各模块、表达设备间关系、类似OO编程模式、热插拔事件处理支持。

Kobject的核心功能是：保持一个引用计数，当该计数减为0时，自动释放（由本文所讲的kobject模块负责） Kobject所占用的meomry空间。这就决定了Kobject必须是动态分配的（只有这样才能动态释放）。

而Kobject大多数的使用场景，是内嵌在大型的数据结构中（如Kset、device\_driver等），因此这些大型的数据结构，也必须是动态分配、动态释放的。那么释放的时机是什么呢？是内嵌的Kobject释放时。但是Kobject的释放是由Kobject模块自动完成的（在引用计数为0时），那么怎么一并释放包含自己的大型数据结构呢？

20. 这时Ktype就派上用场了。我们知道，Ktype中的release回调函数负责释放Kobject（甚至是包含Kobject的数据结构）的内存空间，那么Ktype及其内部函数，是由谁实现呢？是由上层数据结构所在的模块！因为只有它，才清楚Kobject嵌在哪个数据结构中，并通过Kobject指针以及自身的数据结构类型，找到需要释放的上层数据结构的指针，然后释放它。

讲到这里，就清晰多了。所以，每一个内嵌Kobject的数据结构，例如kset、device、device\_driver等等，都要实现一个Ktype，并定义其中的回调函数。同理，sysfs相关的操作也一样，必须经过ktype的中转，因为sysfs看到的是Kobject，而真正的文件操作的主体，是内嵌Kobject的上层数据结构。

21. ksets 像是 kobj\_type 结构的扩展; 一个 kset 是嵌入到相同类型结构的 kobject 的集合。但 struct kobj\_type 关注的是对象的类型，而struct ksets 关心的是对象的聚合和集合，其主要功能是包容，可认为是kobjects 的顶层容器类。每个 kset 在内部包含自己的 kobject, 并可以用多种处理kobject 的方法处理kset。 ksets 总是在 sysfs 中出现; 一旦设置了 kset 并把它添加到系统中, 将在 sysfs 中创建一个目录；kobjects 不必在 sysfs 中表示, 但kset中的每一个 kobject 成员都在sysfs中得到表述。

增加 kobject 到 kset 中去，通常是在kobject 创建时完成，其过程分为2步：

1）完成kobject的初始化，特别注意mane和parent和初始化。

2）把kobject 的 kset 成员指向目标kset。

3）将kobject 传递给下面的函数

22. 这里介绍一些底层sysfs操作：kobject and sysfs

kobject\_add为对象在sysfs中创建一个入口，每个对象对应一个目录，目录中包含对象的属性文件，对文件的操作，即为对kobject对应的设备属性的设置或获取。

kobject\_set\_name() 名字必须在本目录中唯一，如果对象有null parent，kobject\_add为其设置为指向kset，如果两者都为null，则kobject为sysfs根目录中的一项。

23. 这里是Kobjects层次结构和块子系统。基本上是解释刚才我们所说的各种类型的数据结构在整体中的位置以及相互的关系。

24. sysfs目录：sysfs 是 Linux 内核中设计较新的一种虚拟的基于内存的文件系统。

/sys 下的目录结构是经过精心设计的：在 /sys/devices 下是所有设备的真实对象，包括如视频卡和以太网卡等真实的设备，也包括 ACPI 等不那么显而易见的真实设备、还有 tty, bonding 等纯粹虚拟的设备；在其它目录如 class, bus 等中则在分类的目录中含有大量对 devices 中真实对象引用的符号链接文件。

也就是说，在文件系统之中，devices目录是非常重要的，这是内核对系统中所有设备的分层次表达模型，也是 /sys 文件系统管理设备的最重要的目录结构，下文会对它的内部结构作进一步分析。

25. 这里介绍了sysfs的一些文件以及属性。

26. 这里我们开始介绍驱动的交互：主要是基于Buses, devices, and drivers之间的交互。

Buses是总线，devices是设备，drivers是驱动，那么实际上我们这里进行的操作就是：

首先driver从bus获取driver core和kobject core（也就是我们之前所说的，用于管控的数据结构）的控制信息。

然后driver根据获取的命令信息产生相对应的操作，通过bus传递到device中。

因此我们可以看出来其实bus与在操作系统的其他地方起到的功能一样，起到了数据传递的作用。但是我们为了实现这样的功能也是需要进行一些对应的注册的。

27. 首先为了实现在bus上传递信息，我们需要注册总线信息。这一步操作是需要在初始化函数与退出函数中设定的：在初始化函数中注册总线，在退出函数中取消注册。

28. 在注册好总线后，我们就可以进行总线操作了，这里列举了一些总线操作方法，我们可以通过这些方法进行总线的消息传递，信息的获取以及匹配。

29. 这里是device的一些信息，在此页中我们首先先看一下device的内容，然后我们需要进行外部的处理。

这一页我们需要注意下第一项：struct device \*parent;，我们可以注意一下device->kobj->parent其实是完全等同于&device->parent->kobj。

30. 设备我们也是需要注册的。通过device\_register和device\_unregister两个函数就是设备的注册函数与退出函数。

值得注意的是我们是在bus中注册设备信息，但是bus本身也是一种设备（理解为一种硬件通道），那么我们同样是需要为BUS注册设备信息的（这一点很重要）。