《openEuler内核编程》

课程讲稿

第八章 第4讲

USB设备

软件所制

第八章 第4讲 USB设备

**学时：1**学时

**教学目的：**系统学习USB设备，深入了解USB设备原理.

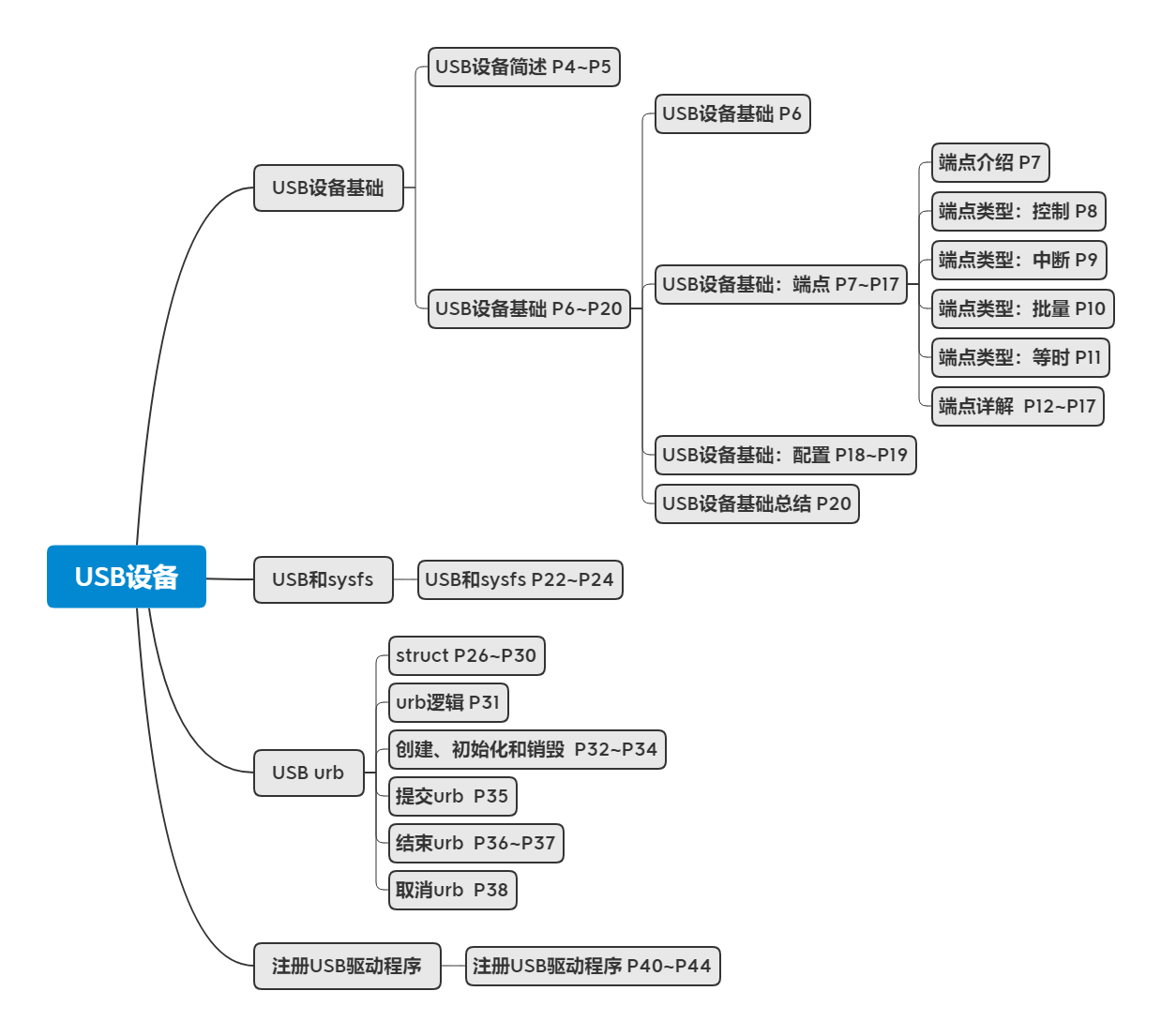
**课程时间线：**



**课外参考读物：**

《Linux驱动程序（第三版）》

**知识框图：**

****

**PPT讲稿：**

1. 今天是操作系统第八章的最后一节内容，USB设备。在这一节中我们将介绍usb设备及usb设备驱动的相关内容。

2. USB设备是我们操作系统课程，第六章设备驱动的最后一节内容。

3. 我们首先介绍一下本节的第一部分内容：USB设备基础。在这一部分我们主要介绍USB的一些性质和在软硬件层面的一些知识铺垫。

4. USB大家肯定用过，包括手机的充电线、数据线在内，我们平常生活中是经常会接触到USB的，然而同学们用的多不代表对其理解有多深，因此我们现在需要详细的学习一下它。

通用串行总线（USB）是主机和外围设备之间的一种连接。

从拓扑上来看，一个USB子系统并不是以总线的方式来布置的；它是一棵由几个点对点的连接构建而成的树。

USB主控制器（ host controller）负责询问每一个USB设备是否有数据需要发送。

由于拓扑结构的原因，一个USB设备在没有主控制器要求的情况下是不能发送数据的。

USB总线在技术层面上是非常简单的，因为它是一个单主方式的实现，在此方式下，主机轮询各种不同的外围设备。尽管存在这种内在的局限性，USB总线有一些吸引人的特性，例如设备具有要求一个固定的数据传输带宽的能力，以可靠地支持视频和音频I/O，USB另一个重要的特性是它只担当设备和主控制器之间通信通道的角色，对它所发送的数据没有任何特殊的内容和结构上的要求.

USB协议规范定义了一套任何特定类型的设备都可以遵循的标准。如果一个设备遵循该标准，就不需要一个特殊的驱动程序。

这些特性，加上设计上与生俱来的热插拔能力，使得USB成为一个便利和低成本的机制，它可以连接多个设备到计算机，而不需要关闭系统、打开机箱、拧螺丝钉和插拔电线。

5. 如图所示，USB驱动程序存在于不同的内核子系统（块设备、网络设备，字符设备等等）和USB硬件控制器之中，USB核心为USB驱动程序提供了一个用于访间和控制USB硬件的接口，而不必考虑系统当前存在的各种不同类型的USB硬件控制器

6. USB设备是一个非常复杂的东西，官方USB文档中有详细的描述。幸运的是， Linux内核提供了一个称为USB核心（ USB core）的子系统来处理大部分的复杂性。本章描述驱动程序和USB核心之间的接口。右图展示了USB设备的构成，包括配置、接口和端点，以及USB驱动程序如何绑定到USB接口上，而不是整个USB设备。

7. USB通信最基本的形式是通过一个名为端点（endpoint）的东西。USB端点只能往一个方向传送数据，从主机到设备（称为输出端点）或者从设备到主机（称为输入端点）。端点可以看作是单向的管道。

8. 控制端点用来控制对USB设备不同部分的访问。它们通常用于配置设备、获取设备信息、发送命令到设备，或者获取设备的状态报告。这些端点一般体积较小。每个USB设备都有一个名为“端点0”的控制端点，USB核心使用该端点在插入时进行设备的配置，USB协议保证这些传输始终有足够的保留带宽以传送数据到设备。

9. 每当USB宿主要求设备传输数据时，中断端点就以一个固定的速率来传送少量的数据。这些端点是USB键盘和鼠标所使用的主要传输方式。它们通常还用于发送数据到USB设备以控制设备，不过一般不用来传输大量的数据，USB协议保证这些传输始终有足够的保留带宽以传送数据。

10. 批量（bulk）端点传输大批量的数据，这些端点通常比中断端点大得多（它们可以一次持有更多的字符）.它们常见于需要确保没有数据丢失的传输的设备。USB协议不保证这些传输始终可以在特定的时间内完成。如果总线上的空间不足以发送整个批量包，它将被分割为多个包进行传输。这些端点通常出现在打印机、存储设备和网络设备上。

11. 等时（isochronous）端点同样可以传送大批量的数据，但数据是否到达是没有保证的。这些端点用于可以应付数据丢失情况的设备，这类设备更注重于保持一个恒定的数据流。实时的数据收集（例如音频和视频设备）几乎毫无例外都使用这类端点。

12. 然后我们来看一下在软件层面，关于端点的描述。端点的描述，涉及到了两个结构体：usb\_host\_endpoint和usb\_endpoint\_descriptor，内核中使用（1）结构体来描述USB端点，该结构体在（2）这个结构体中包含了真正的端点信息。（2）结构体包含了所有的USB特定的数据，大部分数据格式是由设备定义的，但是需包含如下一些字段：

bEndpointAddress

这是特定端点的USB地址。这个8位的值中还包含了端点的方向。该字段可以结合位掩码USB\_DIR\_OUT和USB\_DIR\_IN来使用，以确定该端点的数据是传向设备还是主机。

13.

bmAttributes

这是端点的类型。该值可以结合位掩码USB\_ ENDPOINT\_ XFERTYPE MASK来使用，以确定此端点的类型是USB\_ENDPOINT\_XFER\_ISOC、 USB\_ENDPOINT\_XFER\_BULK还是USB\_ENDPOINT\_XFER\_INT.

这些宏分别表示等时、批量和中断端点

wMaxPacketSize

这是该端点一次可以处理的最大字节数。注意，驱动程序可以发送数量大于此值的数据到端点，但是在实际传输到设备的时候，数据将被分割为 wMaxpacketsize大小的块，对于高速设备，通过使用高位中一些额外的位，该字段可以用来支持端点的高带宽模式。

14. bInterval

如果端点是中断类型，该值是端点的间隔设置——也就是说，端点的中断请求间隔时间。该值以毫秒为单位。

以上所述的几个字段并没有采用“传统的” Linux内核命名方案，这是因为这些字段直接对应于USB规范中的字段名字，USB内核程序员认为使用规范指定的名字比使用 Linux程序员熟悉的变量命名方式更加重要，因为这样便于规范的阅读。

15. 由端点所引申出来的是接口，USB端点被捆绑为接口。USB接口只处理一种USB逻辑连接，例如鼠标、键盘或者音频流。一些USB设备具有多个接口，例如USB扬声器可以包括两个接口：一个USB键盘用于按键和一个USB音频流。因为一个USB接口代表了一个基本功能，而每个USB驱动程序控制一个接口，因此，以扬声器为例， Linux需要两个不同的驱动程序来处理一个硬件设备。

16. 内核使用 struct usb\_interface结构体来描述USB接口。USB核心把该结构体传递给USB驱动程序，之后由USB驱动程序来负责控制该结构体。

该结构体中的一些重要字段：

struct usb\_host\_interface \*altsetting

一个接口结构体数组，包含了所有可能用于该接口的可选设置。每个 struct usb\_host\_ interface结构体包含一套由上述 struct usb\_host\_endpoint结构体定义的端点配置。注意，这些接口结构体没有特定的次序。

unsigned num\_altsetting

altsetting指针所指的可选设置的数量。

struct usb\_host\_interface \*cur\_altsetting  
指向altsetting数组内部的指针，表示该接口的当前活动设置

17. int minor

如果捆绑到该接口的USB驱动程序使用USB主设备号，这个变量包含USB核心分配给该接口的次设备号。这仅在一个成功的 usb\_register\_dev调用之后才有效。

与之前描述端点一样，我们在这里列举出来的字段都是驱动程序需要得字段，除此之外结构体之中还有很多其他字段。

18. 由接口引申出来配置，USB接口本身被捆绑为配置。一个USB设备可以有多个配置，而且可以在配置之间切换以改变设备的状态。

19. Linux使用struct usb\_host\_config结构体来描述USB配置，使用 struct usb\_device结构体来描述整个USB设备。

然而在驱动程序中，并不关心该结构体的任何字段。因此我们在结构体这里并不需要了解什么字段信息。

20. 我们在这里做一个总结：

概言之，USB设备是非常复杂的，它由许多不同的逻辑单元组成。

这些逻辑单元之间的关系：

·设备通常具有一个或者更多的配置

·配置经常具有一个或者更多的接口

·接口通常具有一个或者更多的设置

·接口没有或者具有一个以上的端点

21. 接下来我们来看一下USB与sysfs的关系。

22. 由于单个USB物理设备的复杂性，在 sysfs中表示该设备也相当复杂。无论是物理USB设备（用 struct usb\_device表示）还是单独的USB接口（用 struct usb\_interface表示），在 sysfs中均表示为单独的设备（这是因为这些结构体都包含一个 struct device结构体）.

23. struct usb device表示为目录树中的

sys/ devices/pei0000：00/0000：00：09.0/ub2/2-1（第一行）

而鼠标的USB接口（USB鼠标驱动程序所绑定的接口）位于如下目录

sys/ evices/pc1000000：0009.0/ub2/2-1/2-1：1.0（第二行）

我们将描述内核如何分类USB设备，以帮助理解上面这些长长的设备路径名的含义。

第一个USB设备是一个根集线器（ root hub）.这是一个USB控制器，通常包含在一个PCI设备中。之所以这样命名该控制器，是因为它控制着连接到其上的整个USB总线该控制器是连接PCI总线和USB总线的桥，也是该总线上的第一个USB设备。

所有的根集线器都由USB核心分配了一个独特的编号，在我们的例子中，根集线器名为

usb2，因为它是注册到USB核心的第二个根集线器，单个系统中可以包含的根集线器

的编号在任何时候都是没有限制的

USB总线上的每个设备都以根集线器的编号作为其名字中的第一个号码。该号码随后是一个横杠字符和设备所插入的端口号。因为我们例子中的设备插入到第一个端口，1被添加到了名字中。因此，主USB鼠标设备的设备名是2-1.因为该USB设备包含一个接口、导致了树中的另一个设备被添加到 sysfs 路径中，USB接口的命名方案是设备名直到该接口为止：在我们的例子中，是2-1后面加一个冒号和USB配置的编号，然后是一个句点和接口的编号。因此对于本例而言，设备名是2-1：1.0，因为它是第一个配置，具有接口编号零。

24. 概言之， USB sysfs设备命名方案为

根集线器-集线器端口号：配置。接口

随着设备更深地进入USB树，和越来越多的USB集线器的使用，集线器的端口号被添加到跟随着链中前一个集线器端口号的字符串中。对于一个两层的树，其设备名类似于:

根集线器-集线器端口号-集线器端口号：配置。接口

从前面的USB设备和接口的目录列表可以看到，所有的USB特定信息都可以从 sysfs直接获得（例如， id Vendor、 id Product和 bMaxPower信息）.这些文件中的一个，即bConfigurationvalue，可以被写入以改变当前使用的活动USB配置。当内核不能够确定选择哪一个配置以恰当地操作设备时，这对于具有多个配置的设备很有用。许多USB调制解调器需要向该文件中写入适当的配置值，以便把恰当的USB驱动程序绑定到该设备。

sysfs并没有展示USB设备所有的不同部分，它只限于接口级别。设备可能包含的任何可选配置都没有显示，还有和接口相关联的端点的细节。这个信息可以从asb文件系统找到，该文件系统被挂装到系统的/ proc/bus/usb/目录。/roc/bus/usb/devices文件确实显示了和 sysfs所展示的所有信息相同的信息，还有系统中存在的所有USB设备的可选配置和端点信息。sb还允许用户空间的程序直接访问USB设备，这使得许多内核动程序可以迁移到用户空间，从而更加容易维护和调试USB扫描仪是一个很好的例子，它不再存在于内核中，因为它的功能现在包含在了用户空间的SANE库程序中。

25. 接下来我们来看一下下一部分的内容，USB urb。

我们在USB urb中需要关注需要字段，我们一个一个的来进行讲解：

26. ~30 代码讲解见ppt

31.

接下来我们来介绍一些urb逻辑，包括创建、初始化和销毁，其中初始化对于不同的端点是不同的情况。

在介绍了关于初始化和结束后是提交urb、结束urb、和取消urb

32.

struct urb结构体不能在驱动程序中或者另一个结构体中静态地创建，因为这样会破坏USB核心对urb所使用的引用计数机制。它必须使用asb\_ alloc urb函数来创建。该函数原型如下

struct urb \*usb alloc urb（int iso packets， int mem flags）：

第一个参数， iso\_packets，是该urb应该包含的等时数据包的数量。如果不打算创建等时urb，该值应该设置为0.第二个参数，mem\_f1ags，和传递给用于从内核分配内存的 kmalloc函数的标志有相同的类型。如果该函数成功地为urb分配了足够的内存空间，指向该urb的指针将被返回给调用函数。如果返回值为NULL，说明USB核心内发生了错误，驱动程序需要进行适当的清理。

当一个urb被创建之后，在它可以被USB核心使用之前必须被正确地初始化。我们之后会介绍如何初始化不同类型的urb。驱动程序必须调用 usb\_free\_urb函数来告诉USB核心驱动程序已经使用完urb，该函数只有一个参数void usb free urb（struct urb \*urb）：这个参数是指向所需释放的 struct urb的指针。在该函数被调用之后，urb结构体就消失了，驱动程序不能再访问它.

接下来我们需要注意的是初始化这一步是对于不同的端点是不同的。

我们复习一下端点分类：分成四类，控制、中断、批量和等时。

33.

其实他们之间十分接近，因此我们首先学习其中一个的初始化函数，后续仅需以此为基础讲述变动即可。

我们以中断的函数为例子，这里是中断端点的初始化函数，我们可以看到有很多参数，我们一个一个来看一下：

该函数包含很多的参数

介绍完中断的初始化函数后我们来看一下批量urb的初始化函数，名称做了更改bulk 批量，然后与上面的函数相比，仅仅是删除了int interval参数，这个参数是时间间隔参数，因为批量urb没有时间间隔值所以是没有这个参数的。

然后控制urb的初始化方法与批量urb的差距是多了一个参数（需要注意的是与中断urb相比仍缺少时间间隔参数）。

多了的参数是unsigned char \*setup\_packet，它指向即将被发送到端点的设置数据包的数据。

34.

而对于等时urb，和其他的初始化是不同的。

对于等时传输，urb里是可以指定多次传输的，所以必须一个一个的对变长数组iso\_frame\_esc内容进行初始化。

这里是一个手工初始化的例子。

35. 介绍完创建、初始化和销毁后，我们现在来看一下提交urb。

一且urb被USB驱动程序正确地创建和初始化之后，就可以提交到USB核心以发送到USB设备了。这是通过调用 usb\_submit\_urb函数来完成的。

int usb\_submit\_urb(struct urb \*urb, int mem\_flags);

一旦成功提交，在接收函数被调用之前不能访问该urb结构体中的任何字段。

三个需要注意的值：

36.

结束urb

如果调用 usb\_submit\_urb成功: 把对urb的控制转交给USB核心，且该函数返回0；

否则，返回负的错误号。

如果函数调用成功，当urb结束的时候urb的结束处理例程（由结束  
函数指针指定）正好被调用一次。

当该函数被调用时，USB核心结束了对URB的处理，  
此刻对它的控制被返回给设备驱动程序。

37.

只有三种结束ub和调用结束函数的情形：

·urb被成功地发送到了设备，设备返回了正确的确认。对于 OUT urb而言就是数据  
被成功地发送，对于 INT urb而言就是所请求的数据被成功地接收到。如果确实这  
样、urb中的 status变量被设置为

·发送数据到设备或者从设备接收数据时发生了某种错误。错误情况由urb结构体中  
的 status变量的错误值来指示

·urb从USB核心中被“解开链接”，当驱动程序通过 usb un1nk\_Hrb或 usb kill urh  
调用告诉USB核心取消一个已提交的mb时，或者当设备从系统中被移除而一个  
urb已经提交给它时，会发生这种情况

38.

应该调用 usb\_kill\_urb或 usb\_unlink\_urb函数来终止一个已经被提交到USB核心的urb  
int usb\_kill\_urb(struct urb \*urb);  
int usb\_unlink\_urb(struct urb \*urb);

注：这两个函数的urb参数是指向即将被取消的urb的指针.

一般性的调用情形：当设备从系统中被断开时，在断开回调函数中调用该函数。

39. 然后是本节的最后一个内容，注册USB驱动程序

40. 所有USB驱动程序都必须创建的主要结构体是 struct usb\_ driver.该结构体必须由USB驱动程序来填写，包括许多回调函数和变量，它们向USB核心代码描述了USB驱动程序。

接下来我们需要了解一下该结构体中需要关注的字段。

41.

struct module \*owner

指向该驱动程序的模块所有者的指针。USB核心使用它来正确地对该USB驱动程序进行引用计数，使它不会在不合适的时刻被卸载掉。该变量应该被设置为THIS\_MODULE宏。

const char \*name

指向驱动程序名字的指针。在内核的所有USB驱动程序中它必须是唯一的，通常被设置为和驱动程序模块名相同的名字。如果该驱动程序运行在内核中，可以在sysfs的/sys/ bus/usb/ drivers/下面找到它。

onst struct usb-device id \*id table

指向 struct usb device-id表的指针，该表包含了一列该驱动程序可以支持的所有不同类型的USB设备。如果没有设置该变量，USB驱动程序中的探测回调函数不会被调用。如果想要驱动程序对于系统中的每一个USB设备都被调用，创建一个只设置 driver-info字段的条目：

static struct usb\_ device\_ id usb\_ids [ ]={

{ .driver\_info = 42 },

{ }

};

int（\*probe）（struct usb\_interface \*intf，const struct usb\_device\_id \*id）

指向USB驱动程序中的探测函数的指针。当USB核心认为它有一个 structusb interface可以由该驱动程序处理时，它将调用该函数（在“探测和断开的细节”一节中描述）.USB核心用来作判断的指向 struct usb device id的指针也被传递给该函数。如果USB驱动程序确认传递给它的 struct usb interface，它应该恰当地初始化设备然后返回0.如果驱动程序不确认该设备，或者发生了错误它应该返回一个负的错误值。

43.

以 struct usb\_driver指针为参数的 usb\_register\_driver函数调用把 struct

usb\_driver注册到USB核心。

传统上是在USB驱动程序的模块初始化代码中完成该工作的,如PPT中的这些代码。

44.

当USB驱动程序将要被卸载时，需要把struct usb\_driver从内核中注销。通过调用usb\_deregister\_driver来完成该工作。当该调用发生时，当前绑定到该驱动程序上的任何USB接口都被断开，断开函数将被调用。

45. 好的，本节课的内容就到这里。