



下载APP



28 | 部门分类：如何表示设备类型与设备驱动？

2021-07-12 LMOS

《操作系统实战45讲》

课程介绍 >

**讲述：陈晨**

时长 13:18 大小 12.19M



你好，我是 LMOS。

小到公司，大到国家，都有各种下属部门，比如我们国家现在有教育部、科学技术部、外交部，财政部等，这些部门各自负责完成不同的职能工作，如教育部负责教育事业和语言文字工作，科学技术部负责推动解决经济社会发展的重大科技问题。

既然大道相通，那我们的 Cosmos 中是否也是类似这样的结构呢？

答案是肯定的，在前面的课中，我们搞定了内存管理和进程管理，它们是内核不可分割的，但是计算机中还有各种类型的设备需要管理。



我们的 Cosmos 也会“成立各类部门”，用于管理众多设备，一个部门负责一类设备。具体要怎么管理设备呢？你不妨带着这个问题，正式开始今天的学习！

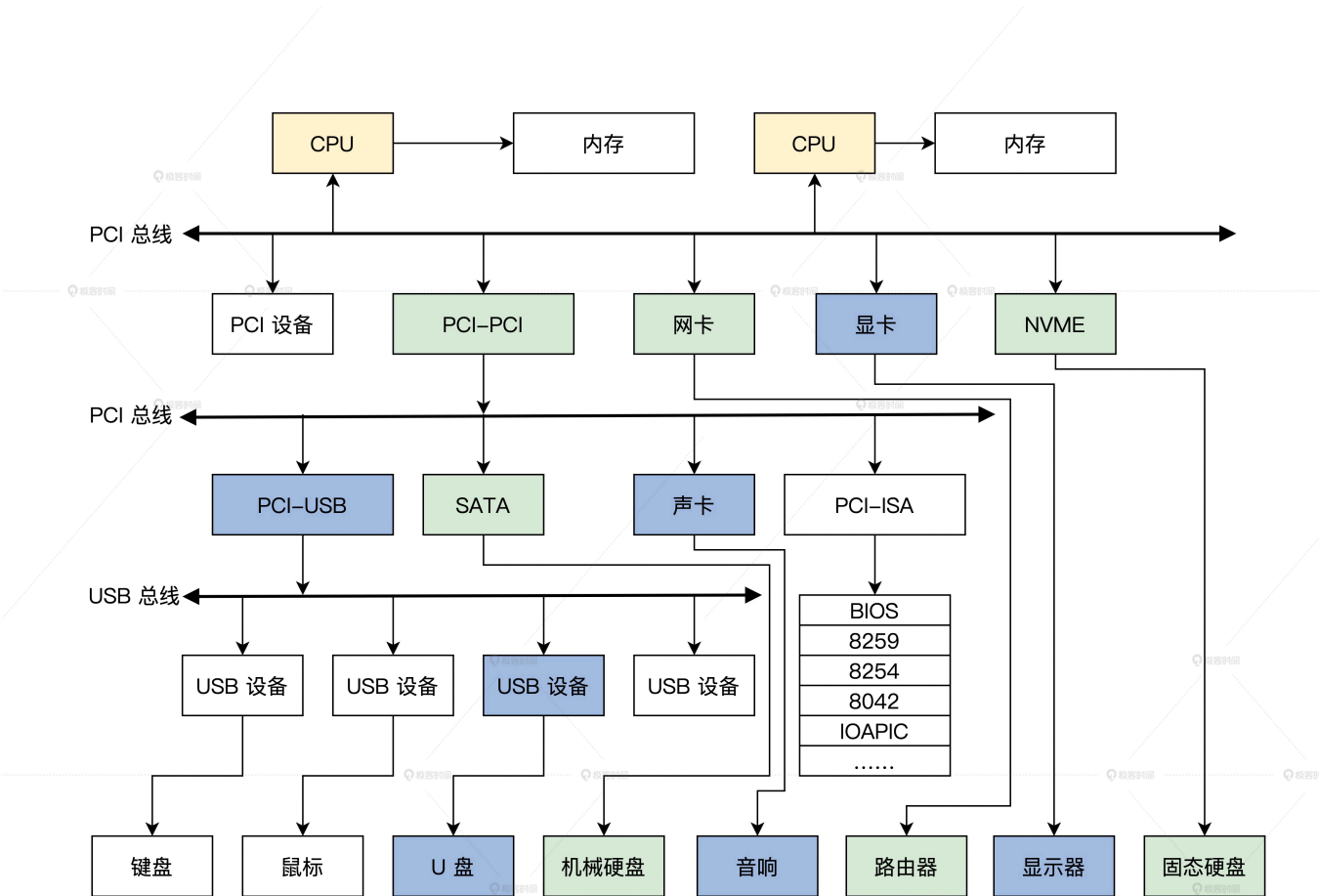
这节课的代码，你可以从[这里](#)下载。

计算机的结构

不知道你是否和我一样，经常把计算机的机箱打开，看看 CPU，看看内存条，看看显卡，看看主板上的各种芯片。

其实，这些芯片并非独立存在，而是以总线为基础连接在一起的，各自完成自己的工作，又能互相打配合，共同实现用户要求的功能。

为了帮你理清它们的连接关系，我为你画了一幅图，如下所示。



计算机结构示意图

上图是一个典型的桌面系统，你先不用管是物理上怎么样连接的，逻辑上就是这样的。实际可能比图中有更多或者更少的总线。但是总线有层级关系，各种设备通过总线相连。这里我们只需要记住，计算机中有很多种类的设备，脑中有刚才这幅图就行了。

如何管理设备

在前面的课程中，我们实现了管理内存和进程，其实进程从正面看它是管理应用程序的，反过来看它也是管理 CPU 的，它能使 CPU 的使用率达到最高。

管理内存和管理 CPU 是操作系统最核心的部分，但是这还不够，因为计算机不止有 CPU，还有各种设备。

如果把计算机内部所有的设备和数据都描述成资源，操作系统内核无疑是这些资源的管理者。既然设备也是一种资源，如何高效管理它们，以便提供给应用进程使用和操作，就是操作系统内核的重要任务。

分权而治

一个国家之所以有那么多部门，就是要把管理工作分开，专权专职专责，对于操作系统也是一样。

现代计算机早已不限于只处理计算任务，它还可以呈现图像、音频，和远程计算机通信，储存大量数据，以及和用户交互。所以，计算机内部需要处理图像、音频、网络、储存、交互的设备。这从上面的图中也可以看得出来。

操作系统内核要控制这些设备，就要包含每个设备的控制代码。如果操作系统内核被设计为通用可移植的内核，那是相当可怕的。试想一下，这个世界上有如此多的设备，操作系统内核代码得多庞大，越庞大就越危险，因为其中一行代码有问题，整个操作系统就崩溃了。

可是仅仅只有这些问题吗？当然不是，我们还要考虑到后面这几点。

1. 操作系统内核开发人员，不可能罗列世界上所有的设备，并为其写一套控制代码。

2. 为了商业目的，有很多设备厂商并不愿意公开设备的编程细节。就算内核开发人员想为其写控制代码，实际也不可行。

3. 如果设备更新换代，就要重写设备的控制代码，然后重新编译操作系统内核，这样的话操作很麻烦，操作系统内核开发人员和用户都可能受不了。

以上三点，足以证明这种方案根本不可取。

既然操作系统内核无法包含所有的设备控制代码，那就索性不包含，或者只包含最基本、最通用的设备控制代码。这样操作系统内核就可以非常通用，非常精巧。

但是要控制设备就必须要有设备的相关控制代码才行，所以我们要把设备控制代码独立出来，与操作系统内核分开、独立开发，设备控制代码可由设备厂商人员开发。

每个设备对应一个设备控制代码模块，操作系统内核要控制哪个设备，就加载相应的设备代码模块，以后不使用这个设备了，就可以删除对应的设备控制代码模块。

这种方式，给操作系统内核带来了**巨大的灵活性**。设备厂商在发布新设备时，只要随之发布一个与此相关的设备控制代码模块就行了。

设备分类

要想管理设备，先要对其分门别类，在开始分类之前，你不妨先思考一个问题：操作系统内核所感知的设备，一定要与物理设备——对应吗？

举个例子，储存设备，其实不管它是机械硬盘，还是 TF 卡，或者是一个设备控制代码模块，它向操作系统内核表明它是储存设备，但它完全有可能分配一块内存空间来储存数据，不必访问真正的储存设备。**所以，操作系统内核所感知的设备，并不需要和物理设备对应，这取决于设备控制代码自身的行为。**

操作系统内核所定义的设备，可称为内核设备或者逻辑设备，其实这只是对物理计算平台中几种类型设备的一种抽象。下面，我们在 `cosmos/include/knlinc/krldevice_t.h` 文件中对设备进行分类定义，代码如下。

```

1  #define NOT_DEVICE 0           //不表示任何设备
2  #define BRIDGE_DEVICE 4       //总线桥接器设备
3  #define CPUCORE_DEVICE 5      //CPU设备，CPU也是设备
4  #define RAMCONTER_DEVICE 6    //内存控制器设备
5  #define RAM_DEVICE 7         //内存设备
6  #define USBHOSTCONTER_DEVICE 8 //USB主控制设备
7  #define INTUPTCONTER_DEVICE 9 //中断控制器设备
8  #define DMA_DEVICE 10        //DMA设备
9  #define CLOCKPOWER_DEVICE 11  //时钟电源设备
10 #define LCDCONTER_DEVICE 12   //LCD控制器设备
11 #define NANDFLASH_DEVICE 13   //nandflash设备
12 #define CAMERA_DEVICE 14      //摄像头设备
13 #define UART_DEVICE 15        //串口设备
14 #define TIMER_DEVICE 16       //定时器设备
15 #define USB_DEVICE 17         //USB设备
16 #define WATCHDOG_DEVICE 18   //看门狗设备
17 #define RTC_DEVICE 22         //实时时钟设备
18 #define SD_DEVICE 25          //SD卡设备
19 #define AUDIO_DEVICE 26       //音频设备
20 #define TOUCH_DEVICE 27       //触控设备
21 #define NETWORK_DEVICE 28     //网络设备
22 #define VIR_DEVICE 29         //虚拟设备
23 #define FILESYS_DEVICE 30     //文件系统设备
24 #define SYSTICK_DEVICE 31     //系统TICK设备
25 #define UNKNOWN_DEVICE 32    //未知设备，也是设备
26 #define HD_DEVICE 33         //硬盘设备

```

上面定义的这些类型的设备，都是 Cosmos 内核抽象出来的逻辑设备，例如 NETWORK_DEVICE 网络设备，不管它是有线网卡还是无线网卡，或者是设备控制代码虚拟出来的虚拟网卡。Cosmos 内核都将认为它是一个网络设备，这就是设备的抽象，这样有利于我们灵活、简便管理设备。

设备驱动

刚才我们解决了设备分类，下面我来研究如何实现分权而治，就是把操作每个设备的相关代码独立出来，这种方式在业界有一个更专业的名字——**设备驱动程序**。同时在下面的内容中，我们将不区分设备驱动程序和驱动程序。

这种“分权而治”的方式，给操作系统内核带了灵活性、可扩展性.....可是也带来了新的问题，有哪些问题呢？

首先是操作系统内核如何表示多个设备与驱动的存在？然后，还有如何组织多个设备和多个驱动程序的问题，最后我们还得考虑应该让驱动程序提供一什么支持。下面我们分别


解决这些问题。

设备

你能说说一个设备包含哪些信息吗？无非是设备类型，设备名称，设备状态，设备 id，设备的驱动程序等。

我们把这些信息归纳成一个数据结构，在操作系统内核建立这个数据结构的实例变量，这个设备数据结构的实例变量，一旦建立，就表示操作系统内核中存在一个逻辑设备了。

我们接下来就一起整理一下设备的信息，然后把它们变成一个数据结构，代码如下。

 复制代码

```
1 typedef struct s_DEVID
2 {
3     uint_t    dev_mtype; //设备类型号
4     uint_t    dev_stype; //设备子类型号
5     uint_t    dev_nr;    //设备序号
6 }devid_t;
7 typedef struct s_DEVICE
8 {
9     list_h_t    dev_list; //设备链表
10    list_h_t    dev_indrvlst; //设备在驱动程序数据结构中对应的挂载链表
11    list_h_t    dev_intbllst; //设备在设备表数据结构中对应的挂载链表
12    spinlock_t  dev_lock; //设备自旋锁
13    uint_t      dev_count; //设备计数
14    sem_t       dev_sem; //设备信号量
15    uint_t      dev_stus; //设备状态
16    uint_t      dev_flg; //设备标志
17    devid_t     dev_id; //设备ID
18    uint_t      dev_intlnr; //设备中断服务例程的个数
19    list_h_t    dev_intserlst; //设备中断服务例程的链表
20    list_h_t    dev_rqlist; //对设备的请求服务链表
21    uint_t      dev_rqlnr; //对设备的请求服务个数
22    sem_t       dev_waitints; //用于等待设备的信号量
23    struct s_DRIVER* dev_drv; //设备对应的驱动程序数据结构的指针
24    void* dev_attrb; //设备属性指针
25    void* dev_privdata; //设备私有数据指针
26    void* dev_userdata; //将来扩展所用
27    void* dev_extdata; //将来扩展所用
28    char_t* dev_name; //设备名
29 }device_t;
```

设备的信息比较多，大多是用于组织设备的。这里的**设备 ID 结构十分重要**，它表示设备的类型、设备号，子设备号是为了解决多个相同设备的，还有一个指向设备驱动程序的指针，这是用于访问设备时调用设备驱动程序的，只要有人建立了一个设备结构的实例变量，内核就能感知到一个设备存在了。


至于是谁建立了设备结构的实例变量，这个问题我们接着探索。

驱动

操作系统内核和应用程序都不会主动建立设备，那么谁来建立设备呢？当然是控制设备的代码，也就是我们常说的**驱动程序**。

那么驱动程序如何表示呢，换句话说，操作系统内核是如何感知到一个驱动程序的存在呢？

根据前面的经验，我们还是要定义一个数据结构来表示一个驱动程序，数据结构中应该包含驱动程序名，驱动程序 ID，驱动程序所管理的设备，最重要的是**完成功能设备相关功能的函数**，下面我们来定义它，代码如下。

 复制代码

```
1  typedef struct s_DRIVER
2  {
3      spinlock_t drv_lock; //保护驱动程序数据结构的自旋锁
4      list_h_t drv_list; //挂载驱动程序数据结构的链表
5      uint_t drv_stuts; //驱动程序的相关状态
6      uint_t drv_flg; //驱动程序的相关标志
7      uint_t drv_id; //驱动程序ID
8      uint_t drv_count; //驱动程序的计数器
9      sem_t drv_sem; //驱动程序的信号量
10     void* drv_safedsc; //驱动程序的安全体
11     void* drv_attrb; //LMOSEM内核要求的驱动程序属性体
12     void* drv_privdata; //驱动程序私有数据的指针
13     drivcallfun_t drv_dipfun[IOIF_CODE_MAX]; //驱动程序功能派发函数指针数组
14     list_h_t drv_alldevlist; //挂载驱动程序所管理的所有设备的链表
15     drventyexit_t drv_entry; //驱动程序的入口函数指针
16     drventyexit_t drv_exit; //驱动程序的退出函数指针
17     void* drv_userdata; //用于将来扩展
18     void* drv_extdata; //用于将来扩展
19     char_t* drv_name; //驱动程序的名字
20 }driver_t;
```


上述代码，你应该很容易看懂。Cosmos 内核每加载一个驱动程序模块，就会自动分配一个驱动程序数据结构并且将其实例化。

而 Cosmos 内核在首次启动驱动程序时，就会调用这个驱动程序的入口点函数，在这个函数中驱动程序会分配一个设备数据结构，并用相关的信息将其实例化，比如填写正确的设备类型、设备 ID 号、设备名称等。


Cosmos 内核负责建立驱动数据结构，而驱动程序又建立了设备数据结构，这一来二去，就形成了一个驱动程序与 Cosmos 内核“握手”的动作。

设备驱动的组织

有了设备、驱动，我们下面探索一下怎么合理的组织好它们。

组织它们要解决的问题，就是在哪里安放驱动。然后我们还要想好怎么找到它们，下面我们用一个叫做**设备表**的数据结构，来组织这些驱动程序数据结构和设备数据结构。


这个结构我已经帮你定义好了，如下所示。

 复制代码

```
1 #define DEVICE_MAX 34
2 typedef struct s_DEVTLIST
3 {
4     uint_t dtl_type; //设备类型
5     uint_t dtl_nr; //设备计数
6     list_h_t dtl_list; //挂载设备device_t结构的链表
7 }devtlist_t;
8 typedef struct s_DEVTABLE
9 {
10     list_h_t devt_list; //设备表自身的链表
11     spinlock_t devt_lock; //设备表自旋锁
12     list_h_t devt_devlist; //全局设备链表
13     list_h_t devt_drvlist; //全局驱动程序链表，驱动程序不需要分类，一个链表就行
14     uint_t devt_devnr; //全局设备计数
15     uint_t devt_drvnr; //全局驱动程序计数
16     devtlist_t devt_devcls[DEVICE_MAX]; //分类存放设备数据结构的devtlist_t结构数组
17 }devtable_t;
```

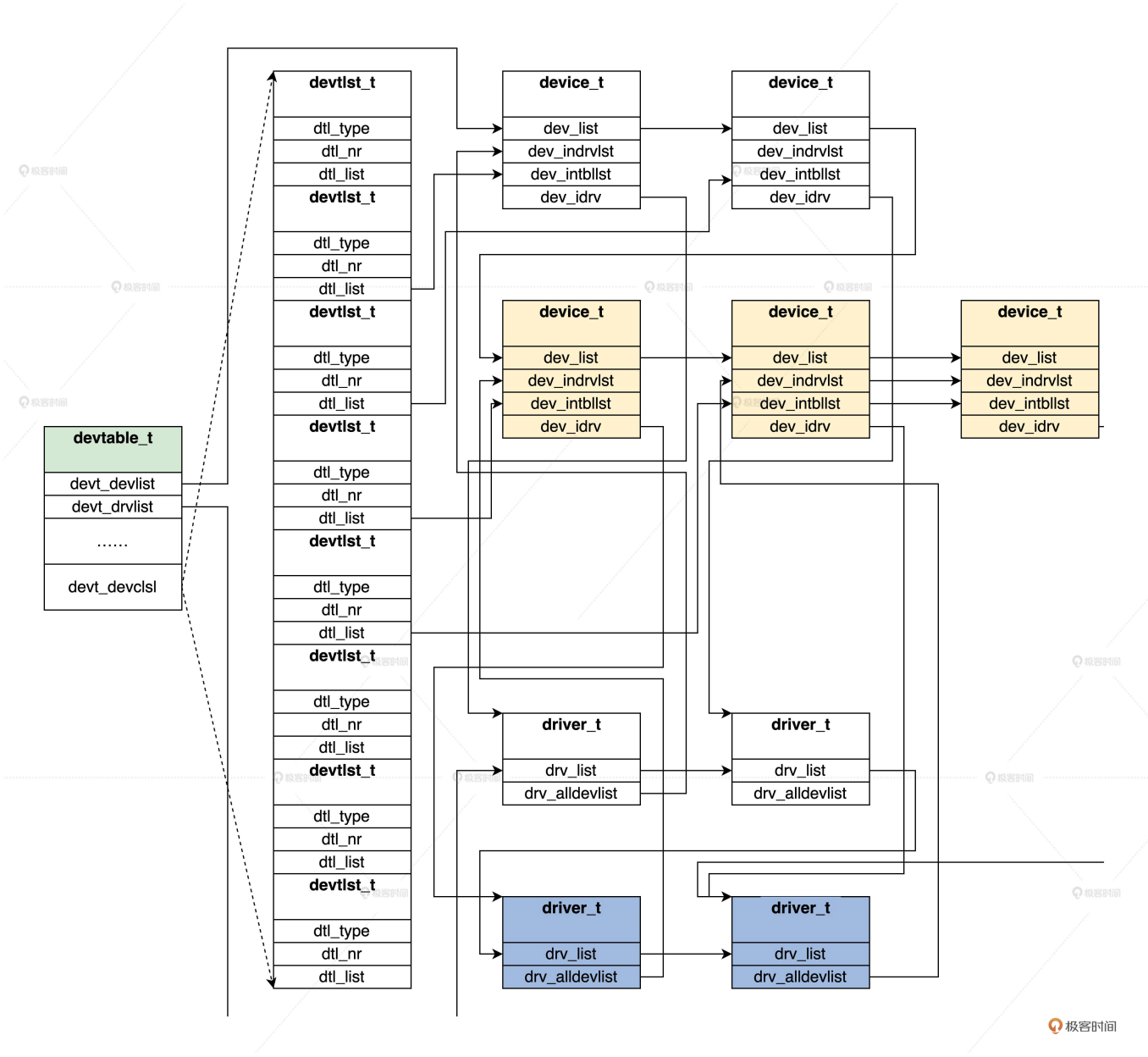
在这段代码的 devtable_t 结构中，devtlist_t 是每个设备类型一个，表示一类设备，但每一类可能有多个设备，所以在 devtlist_t 结构中，有一个设备计数和设备链表。而你可能想

到 Cosmos 中肯定要定义一个 `devtable_t` 结构的全局变量，代码如下。

 复制代码

```
1 //在 cosmos/kernel/krlglobal.c文件中
2 KRL_DEFGLOB_VARIABLE(devtable_t,osdevtable);
3 //在 cosmos/kernel/krldevice.c文件中
4 void devtlst_t_init(devtlst_t *initp, uint_t dtype)
5 {
6     initp->dtl_type = dtype;//设置设备类型    initp->dtl_nr = 0;
7     list_init(&initp->dtl_list);
8     return;
9 }
10 void devtable_t_init(devtable_t *initp)
11 {
12     list_init(&initp->devt_list);
13     krlspinlock_init(&initp->devt_lock);
14     list_init(&initp->devt_devlist);
15     list_init(&initp->devt_drvlist);
16     initp->devt_devnr = 0;
17     initp->devt_drvnr = 0;
18     for (uint_t t = 0; t < DEVICE_MAX; t++)
19     { //初始化设备链表
20         devtlst_t_init(&initp->devt_devclsl[t], t);
21     }
22     return;
23 }
24 void init_krldevice()
25 {
26     devtable_t_init(&osdevtable);//初始化系统全局设备表
27     return;
28 }
29 //在 cosmos/kernel/krlinit.c文件中
30 void init_krl()
31 {
32     init_krlmm();
33     init_krldevice();
34     //记住一定要在初始化调度器之前，初始化设备表
35     init_krlsched();
36     init_krlcpuidle();
37     return;
38 }
```

上面的设备表的初始化代码已经写好了，如果你大脑中没有设备驱动组织图，可能脑子里还是有点乱，所以我来帮你画一幅图，如下所示。



设备表结构示意图

上图看似复杂，实则简单，我帮你理一下重点：首先 devtable_t 结构中找到所有的设备和驱动，然后从设备能找到对应的驱动，从驱动也能找到其管理的所有设备，最后就能实现一个驱动管理多个设备。

驱动程序功能

我们还有一个问题需要解决，那就是驱动程序，究竟要为操作系统内核提供哪些最基本的功能支持？

我们已经知道了，写驱动程序就是为了操控相应的设备，所以这得看大多数设备能完成什么功能了。现代计算机的设备无非就是可以输入数据、处理数据、输出数据，然后完成一些特殊的功能。

当然，现代计算机的设备很多，能耗是个严重的问题，所以操作系统内核应该能控制设备能耗。下面我来帮你归纳一下用来驱动程序的几种主要函数，如下。

[复制代码](#)

```
1 //驱动程序入口和退出函数
2 drvstus_t device_entry(driver_t* drvp,uint_t val,void* p);
3 drvstus_t device_exit(driver_t* drvp,uint_t val,void* p);
4 //设备中断处理函数
5 drvstus_t device_handle(uint_t ift_nr,void* devp,void* sframe);
6 //打开、关闭设备函数
7 drvstus_t device_open(device_t* devp,void* iopack);
8 drvstus_t device_close(device_t* devp,void* iopack);
9 //读、写设备数据函数
10 drvstus_t device_read(device_t* devp,void* iopack);
11 drvstus_t device_write(device_t* devp,void* iopack);
12 //调整读写设备数据位置函数
13 drvstus_t device_lseek(device_t* devp,void* iopack);
14 //控制设备函数
15 drvstus_t device_ioctl(device_t* devp,void* iopack);
16 //开启、停止设备函数
17 drvstus_t device_dev_start(device_t* devp,void* iopack);
18 drvstus_t device_dev_stop(device_t* devp,void* iopack);
19 //设置设备电源函数
20 drvstus_t device_set_powerstus(device_t* devp,void* iopack);
21 //枚举设备函数
22 drvstus_t device_enum_dev(device_t* devp,void* iopack);
23 //刷新设备缓存函数
24 drvstus_t device_flush(device_t* devp,void* iopack);
25 //设备关机函数
26 drvstus_t device_shutdown(device_t* devp,void* iopack);
```

如上所述，我们可以把每一个操作定义成一个函数，让驱动程序实现这些函数。函数名你可以随便写，但是函数的形式却不能改变，这是操作系统内核与驱动程序沟通的桥梁。当然有很多设备本身并不支持这么多操作，例如时钟设备，驱动程序就不必实现相应的操作。

那么这些函数如何和操作系统内核关联起来呢？还记得 `driver_t` 结构中那个函数指针数组吗，如下所示。

[复制代码](#)

```
1 #define IOIF_CODE_OPEN 0 //对应于open操作
2 #define IOIF_CODE_CLOSE 1 //对应于close操作
3 #define IOIF_CODE_READ 2 //对应于read操作
```

```

4 #define IOIF_CODE_WRITE 3 //对应于write操作
5 #define IOIF_CODE_LSEEK 4 //对应于lseek操作
6 #define IOIF_CODE_IOCTL 5 //对应于ioctl操作
7 #define IOIF_CODE_DEV_START 6 //对应于start操作
8 #define IOIF_CODE_DEV_STOP 7 //对应于stop操作
9 #define IOIF_CODE_SET_POWERSTUS 8 //对应于powerstus操作
10 #define IOIF_CODE_ENUM_DEV 9 //对应于enum操作
11 #define IOIF_CODE_FLUSH 10 //对应于flush操作
12 #define IOIF_CODE_SHUTDOWN 11 //对应于shutdown操作
13 #define IOIF_CODE_MAX 12 //最大功能码
14 //驱动程序分派函数指针类型
15 typedef drvstus_t (*drivcallfun_t)(device_t*,void*);
16 //驱动程序入口、退出函数指针类型
17 typedef drvstus_t (*drventyexit_t)(struct s_DRIVER*,uint_t,void*);
18 typedef struct s_DRIVER
19 {
20     //.....
21     drivcallfun_t drv_dipfun[IOIF_CODE_MAX]; //驱动程序分派函数指针数组。
22     list_h_t drv_alldevlist; //驱动所管理的所有设备。
23     drventyexit_t drv_entry;
24     drventyexit_t drv_exit;
25     //.....
26 }driver_t;

```

看到这里，你是不是明白了？driver_t 结构中的 drv_dipfun 函数指针数组，正是存放上述那 12 个驱动程序函数的指针。这样操作系统内核就能通过 driver_t 结构，调用到对应的驱动程序函数操作对应的设备了。

重点回顾

现在，我们搞明白了一个典型计算机的结构，里面有很多设备，需要操作系统合理地管理，而操作系统通过加载驱动程序来管理和使用设备，并为此提供了一系列的机制，这也是我们这节课的重点。

1. 计算机结构，我们通过了解一个典型的计算机系统结构，明白了设备的多样性。然后我们对设备做了抽象分类，采用分权而治的方式，让操作系统通过驱动程序来管理设备，同时又能保证操作系统和驱动程序分离，达到操作系统和设备解耦的目的。

2. 把设备和设备驱动的信息归纳整理，抽象两个对应的数据结构，这两个数据结构在内存中的实例变量就代码一个设备和对应的驱动。接着我们通过设备表结构组织了驱动和设备的数据结构。

3. 驱动程序最主要的工作是要操控设备，但这些个操作设备的动作是操作系统调用的，所以对驱动定义了必须要支持的 12 种标准方法，并对应到函数，这些函数的地址保存在驱动程序的数据结构中。

你可能在想，我们驱动程序是怎么加载的，设备又是怎么建立的呢？这是正是我们后面课程要解决的。不过你可以先开动脑筋，思考一下，提出你自己的见解，考虑一下这个问题的解决方案。

思考题

请你写出一个用来访问设备的接口函数，或者想一下访问一个设备需要什么参数。

欢迎你在留言区跟我交流互动，积极输出有助于更高效地理解这节课的内容。也欢迎你把这节课分享给同事、朋友。

好，我是 LMOS。我们下节课见！

分享给需要的人，Ta 订阅后你可得 **20 元现金奖励**

 赞 2  提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 27 | 瞧一瞧Linux：Linux如何实现进程与进程调度？

下一篇 29 | 部门建立：如何在内核中注册设备？

更多学习推荐

极客时间 | 训练营

大厂面试必考 100 题

2021 最新版 | 算法篇

限量免费领取  仅限前 99 名

精选留言 (4)

[写留言](#)**neohope** 置顶

2021-07-19

一、数据结构

有一个全局devtable_t结构变量osdevtable，用于管理全部驱动程序及设备，其中包括：

A、全局驱动程序链表，保存全部驱动【driver_t结构】

B、全局设备链表，包括各种设备类型的链表【devtlist_t结构】，每个devtlist_t中包括了某一类型的全部设备链表【device_t结构】...

展开 

作者回复: 总结的好

**青玉白露**

2021-07-13

访问一个设备的接口函数大致如下：

```
drvstus_t device_getdata(device_t* devp,void* iopack);
```

其中，device* 指向设备本身的结构体，相当于给这个函数传入了设备的属性值；

而void* iopack是一个无属性的内存块，具体需要传入什么参数，根据访问该设备将要实现的功能而定。...

作者回复: 好的



 2



照葫芦画瓢：

```
//读设备数据函数
```

```
drvstus_t device_read(device_t* devp,void* iopack);
```

作者回复: 可以 可以



👍 1



叮~ 先打个卡 上一节进程调度讲的太精彩了, 相信这一篇也是宝藏文章

作者回复: 哈哈 我努力让大家满意

