Linux 那些事儿之我是 U 盘

摘要

2005 年 6 月,复旦大学微电子系本科毕业答辩上,老师问我:请你用一句话介绍一下 usb 技术.我回了一句:老师,你有病吧,要能用一句话介绍我还费这么大劲写这么长的文章干嘛?

关键词：Linux, Kernel, 2.6, bus, usb, device driver, mass storage, scsi, urb, bulk, control, host, pipe, command, 林志玲

目录

引子 2
小城故事 3
MAKEFILE 不是 MAKE LOVE 4
变态的模块机制 6
想到达明天现在就要启程 9
未曾开始却似结束 11
狂欢是一群人的孤单 12
总线,设备,和驱动(上) 13
总线,设备,和驱动(下) 15
我是谁的他? 16
从协议中来,到协议中去(上) 19
从协议中来,到协议中去(中) 20
从协议中来,到协议中去(下) 23
梦开始的地方 24
设备花名册 27
冰冻三尺非一日之寒 32
冬天来了,春天还会远吗?(一) 36
冬天来了,春天还会远吗?(二) 41
冬天来了,春天还会远吗?(三) 45
冬天来了,春天还会远吗(四) 47
冬天来了,春天还会远吗?(五) 52
通往春天的管道 57
传说中的 URB 63
心锁 69
第一次亲密接触(一) 71
第一次亲密接触(二) 76

第一次亲密接触(三) 80
第一次亲密接触(四) 82
将控制传输进行到底 85
横空出世的 SCSI 88
谁是最变态的结构体 92
SCSI 数据结构-像雾像雨又像风 102
彼岸花的传说(一) 108
彼岸花的传说(二) 109
彼岸花的传说(三) 114
彼岸花的传说(四) 117
彼岸花的传说(五) 123
彼岸花的传说(六) 128
彼岸花的传说(七) 131
彼岸花的传说(八) 135
彼岸花的传说(THE END) 138
SCSI 命令之我型我秀 139
迷雾重重的 BULK 传输(一) 144
迷雾重重的 BULK 传输(二) 150
迷雾重重的 BULK 传输(三) 154
迷雾重重的 BULK 传输(四) 158
迷雾重重的 BULK 传输(五) 163
迷雾重重的 BULK 传输(六) 166
跟着感觉走(一) 171
跟着感觉走(二) 173
光荣属于苹果,属于诺基亚,属于摩托罗拉,属于索尼爱立信! 179
有多少爱可以胡来?(一) 182
有多少爱可以胡来?(二) 188
当梦醒了天晴了 193
其实世上本有路,走的人多了,也便没了路 198

引子

也许是在复旦养成了昼伏夜出的坏习惯,工作之后也总是很晚也不愿意睡.来到北京之后,开始听广播听都市之声的北京不眠夜.这个节目是从 23 点直到第二天凌晨一点,我常常是听完了才会睡觉.无论是北京还是上海,对我来说,生存总是那么困难,生活的压力总是那么大,每天只有在这个节目中才能够寻找到一丝温暖.我不喜欢躺在床上听,而是喜欢一边听一边做点别的事情,于是心血来潮的决定,写点文字吧,听着电波里别人分享心情,不妨也用文字来记录自己的心情吧.

我首先想到的是写一些和 Linux 相关的文字.事实上我并不喜欢 Linux,学习 Linux 完全是一种无奈,工作中要用,迫于生计,不得不去学习,而学习 Linux 的过程中唯一让我觉得还有些乐趣的是当遇到问题的时候可以去网上问去网上查,很多人写了很多文档可以让我们这些菜鸟们参考学习,这样才让我们在工作中走了很多弯路.挺感谢那些分享自己知识的人.碰巧最近

我也看了点冬冬,并且这些冬冬在网上的资料也比较少,所以我想我不妨也把自己那一夜的收获写出来,或许以后也能给别人提供一些帮助,想想也是,整个 Linux 社区不正是这样吗,像陈奕迅唱的那样,"把一个人的温暖转移到另一个人的胸膛".

我要写的是 Linux 设备驱动程序相关的,主要分析的是 Linux 中与 U 盘相关的那部分代码.过去也没有看过,但是今年 4 月底的某一天,一个偶然的原因,我一时冲动就看了一遍.我们几个同学在人大附近打麻将,打到夜深了,因为我们几人人住的位置都离得挺远的,各自回去都得打车,于是决定不如去权金城开个房间,晚上就睡那得了.在权金城洗浴中心,和几个同学洗浴过后,有人去按摩了,而我和另一个人则留在了房间里,无聊中,那位哥们见我带了电脑,说他有部 A 片,很不错,不是很大,所以他存在 U 盘里的,他还挺逗的说这是 2008 年北京奥运会指定 A 片,问我有没有兴趣,这还用问,当然有兴趣了,于是立马打开电脑,插入 u 盘,然后不一会我就傻了,因为我的电脑根本就不能识别 U 盘,首先我的电脑比较旧,装的是双系统,一个是 Win 98,这个没办法,没有 U 盘驱动,另一个是 Linux,2.6 的内核,按理应该是支持 U 盘的,问题是实际情况却是我没有看到 U 盘,/dev/目录下面根本没有这么一个盘符,于是我没办法了,一脸沮丧,而同学在旁边自然表示出了对 Linux 很鄙视的神情.

过了一会,他去看电视了,正好有英超,我却没有心情看电视,想想就觉得奇怪,怎么会不能使用 U 盘呢,这不可能啊,一定是我自己对 Linux 下面的一些冬冬没有弄清楚,于是我决定好好看看问题到底出在哪,记得当时看了一下/var/log/messages 这个日志文件里边好像记录了一些信息,感觉像是一些错误信息,但是看不明白它到底在说什么.同学开始劝我,算了算了,改天再看吧,这话我可不愿意听,不是说 Linux 内核源代码是公开的吗,大不了看看源代码,搞清楚工作原理了还怕问题不能解决?无非就是一些 C 代码而已,好歹哥们也是认真学过谭浩强大哥那本 C 程序设计的.而且当初那本书课后习题老师基本上都让我们做了,虽说是参考了那本习题解答的书,可就算写代码不行,读代码还是没问题吧,语法什么的基本上还是很清楚的,什么判断结构循环结构,包括 goto 语句,还是记得的.

所以我就开始看了,正所谓梦想有多远,就能走多远.以前我只是玩 CS 玩仙剑的时候能够整晚整晚不睡,但那个晚上,为了告诉我同学,Linux 下也能看 A 片,Linux 下遇到问题更适合自己解决,我愣是从一点看到快天亮,终于把 drivers/usb/storage/目录下面一万余行的代码给看了一遍.当然没有看得太仔细,但是很显然把整个原理搞清楚了,问题也很快得以解决.

所以此刻,我整理了一下思路,决定把那晚看的冬冬用文字记录下来.也算为了纪念那个不寻常的夜晚吧.不过我估计这个篇幅不会短,因为光那一万余行的代码贴出来就得占许许多多页了,所以这件事情也许会占用我不少时间,然而,还好,每晚有北京不眠夜的陪伴,而且,也许当我把心思投入到写这个故事的时候,能够把那些压力那些烦恼那种孤独那种郁闷以及那种对生活的绝望给暂时忘记些许.

小城故事

这个故事中使用的是 2.6.10 的内核代码.Linux 内核代码目录中, 所有去设备驱动程序有关的代码都在 drivers/目录下面,在这个目录中我们用 ls 命令可以看到很多子目录.

```
localhost:/usr/src/linux-2.6.10/drivers # ls
Kconfig  atm      cdrom  eisa  ide      macintosh  message  net      parpo
rt  s390  tc       w1
Makefile  base     char   fc4   ieee1394  mca        misc     nubus    pci
sbus  telephony  zorro
acorn    block    cpufreq  firmware  input  md         mmc      oprofile  pcmci
a  scsi  usb
acpi     bluetooth  dio    i2c       isdn   media      mtd      parisc    pnp      se
rial  video
```

其中 usb 目录包含了所有 usb 设备的驱动,而 usb 目录下面又有它自己的子目录,进去看一下,

```
localhost:/usr/src/linux-2.6.10/drivers # cd usb/
locahost:/usr/src/linux-2.6.10/drivers/usb # ls
Kconfig  Makefile  README  atm  class  core  gadget  host  image  input  media
  misc  net  serial  storage  usb-skeleton.c
```

注意到每一个目录下面都有一个 Kconfig 文件和一个 Makefile,这很重要.稍后会有介绍.

而我们的故事其实是围绕着 drivers/usb/storage 这个目录来展开的.实际上这里边的代码清清楚楚地展示了我们日常频繁接触的 U 盘是如何工作的,是如何被驱动起来的.但是这个目录里边的冬冬并不是生活在世外桃源,他们总是和外面的世界有着千丝万缕的瓜葛.可以继续进来看一下,

```
localhost:/usr/src/linux-2.6.10/drivers/usb # cd storage/
localhost:/usr/src/linux-2.6.10/drivers/usb/storage # ls
Kconfig  debug.c  freecom.c  isd200.c  protocol.c  sddr09.c  shuttle_usbat
.c  unusual_devs.h
Makefile  debug.h  freecom.h  isd200.h  protocol.h  sddr09.h  shuttle_usba
t.h  usb.c
datafab.c  dpcm.c  initializers.c  jumpshot.c  scsiglue.c  sddr55.c  transport.c
  usb.h
datafab.h  dpcm.h  initializers.h  jumpshot.h  scsiglue.h  sddr55.h  transport.h
```

咋一看,着实吓了一跳,用`wc -l *`这个命令统计一下,12076 行,晕死...

但是,也许,生活中总是充满了跌宕起伏.

认真看了一下 Makefile 和 Kconfig 之后,心情明显好了许多.

Makefile 不是 Make Love

出来混,迟早要还的.

从前在复旦,混了四年,没有学到任何东西,每天就是逃课,上网,玩游戏,睡觉.毕业的时候,身边的人读研的读研,出国的出国,找工作的吧,去麦肯锡的去麦肯锡,去 IBM 的去 IBM.而自己却一无所长,没有任何技能,直到这时候才发现那四年欠了很多债,早知今日,何必当初.幸运的是,我还有一

张复旦的文凭,依靠着这张文凭,混进了 Intel.然而,工作以后,更是发现当初在校期间没有好好读书其实真是在欠债,当初没学,工作以后还是要学,的确是迟早要还的,逃是逃不掉的.

毕业的时候,人家跟我说 Makefile 我完全不知,但是一说 Make Love 我就来劲了.现在想来依然觉得丢人.

基本上,Linux 内核中每一个目录下边都有一个 Makefile,Makefile 和 Kconfig 就像一个城市的地图,地图带领我们去认识一个城市,而 Makefile 和 Kconfig 则可以让我们了解这个目录下面的结构.drivers/usb/storage/目录下边的 Makefile 内容如下:

```
#
# Makefile for the USB Mass Storage device drivers.
#
# 15 Aug 2000, Christoph Hellwig <hch@infradead.org>
# Rewritten to use lists instead of if-statements.
#

EXTRA_CFLAGS    := -Idrivers/scsi

obj-$(CONFIG_USB_STORAGE)       += usb-storage.o

usb-storage-obj-$(CONFIG_USB_STORAGE_DEBUG)     += debug.o
usb-storage-obj-$(CONFIG_USB_STORAGE_HP8200e)   += shuttle_usbat.o
usb-storage-obj-$(CONFIG_USB_STORAGE_SDDR09)    += sddr09.o
usb-storage-obj-$(CONFIG_USB_STORAGE_SDDR55)    += sddr55.o
usb-storage-obj-$(CONFIG_USB_STORAGE_FREECOM)   += freecom.o
usb-storage-obj-$(CONFIG_USB_STORAGE_DPCM)      += dpcm.o
usb-storage-obj-$(CONFIG_USB_STORAGE_ISD200)    += isd200.o
usb-storage-obj-$(CONFIG_USB_STORAGE_DATAFAB)   += datafab.o
usb-storage-obj-$(CONFIG_USB_STORAGE_JUMPSHOT)  += jumpshot.o

usb-storage-objs :=     scsiglue.o protocol.o transport.o usb.o \
                        initializers.o $(usb-storage-obj-y)
```

关于 Kconfig 文件,在故事的最后会介绍,此刻暂且不表,Kconfig 文件比较长,就不贴出来了.但是通过看 Kconfig 文件,我们可以知道,除了 CONFIG_USB_STORAGE 这个编译选项是我们真正需要的以外,别的选项我们都可以不予理睬.比如,关于 CONFIG_USB_STORAGE_DATAFAB,Kconfig 文件中有这么一段,

```
config USB_STORAGE_DATAFAB
        bool "Datafab Compact Flash Reader support (EXPERIMENTAL)"
        depends on USB_STORAGE && EXPERIMENTAL
        help
          Support for certain Datafab CompactFlash readers.
          Datafab has a web page at <http://www.datafabusa.com/>.
```

显然,这个选项和我们没有关系,首先这是专门针对 Datafab 公司的产品的,其次 CompactFlash reader 是一种 flash 设备,但这显然不是 U 盘,因为 drivers/usb/storage 这个目录里边的代码是针对一类设备的,不是某一种特定的设备,这一类设备就是 usb mass storage 设备,关于这类设备,有专门的文档进行介绍,有相应的 spec,描述这类设备的通信或者物理上电特性上等方面的

规范,U 盘只是其中的一种,这种设备使用的通信协议被称为 Bulk-Only Transport 协议.再比如,关于 CONFIG_USB_STORAGE_SDDR55 这个选项,Kconfig 文件中也有对应的一段,

```
config USB_STORAGE_SDDR55
        bool "SanDisk SDDR-55 SmartMedia support (EXPERIMENTAL)"
        depends on USB_STORAGE && EXPERIMENTAL
        help
          Say Y here to include additional code to support the Sandisk SDDR-55
          SmartMedia reader in the USB Mass Storage driver.
```

很显然这是 SanDisk 的产品,并且是针对 SM 卡的,这也不是 U 盘,所以我们也都不去理睬了.事实上,很容易确定,只有 CONFIG_USB_STORAGE 这个选项是我们真正关心的,而它所对应的模块叫 usb-storage,Makefile 中最后一行也说了,

```
usb-storage-objs :=     scsiglue.o protocol.o transport.o usb.o \
                        initializers.o $(usb-storage-obj-y)
```

这就意味着我们只需要关注的文件就是 scsiglue.c,protocol.c,transport.c,usb.c,initializers.c 以及它们同名的.h 头文件.再次使用 wc -l 命令统计一下这几个文件,发现总长度只有 3701 行,比最初看到的 12000 多行少了许多,当时信心就倍增.

不过需要特别注意的是,CONFIG_USB_STORAGE_DEBUG 这个编译选项,它不是我们必须的,但是如果真的要自己修改或者调试 usb-storage 的代码,那么打开这个选项是很有必要的,因为它会负责打印一些调试信息,以后在源代码中我们会看到它的作用.

变态的模块机制

有一种感动,叫泪流满面,有一种机制,叫模块机制,十月革命一声炮响,给 Linux 送来了模块机制.显然,这种模块机制给那些 Linux 的发烧友们带来了方便,因为模块机制意味着人们可以把庞大的 Linux 内核划分为许许多多个小的模块,对于编写设备驱动程序的那帮家伙来说,从此以后他们可以编写设备驱动程序却不需要把她编译进内核,不用 reboot 机器,她只是一个模块,当你需要她的时候,你可以把她抱入怀中(insmod),当你不再需要她的时候,你可以把她一脚踢开,甚至,你可以对她咆哮:"滚吧,贱人!"(rmmod).她不能成为你的手足,只能算你的衣服.

也许在现实世界里不会这样,但是在 Linux 的虚拟世界里,确实可以是如此,time and time again,我问自己,模块是否就像现实生活中的妓女一样呢?Linux 内核是嫖客,当他需要这个模块的时候,他就把人家揽入怀中,当他不需要人家的时候,就把别人踢开,而且,模块总是能够逆来顺受,尽管 Linux 内核会一次次抛弃她,但是每当 Linux 内核再次需要她的时候,当内核再次执行 insmod 的时候,模块依然会尽自己的能力去取悦内核,这是否太可悲了些!记得孔子曾经说过,读懂Linux内核代码不难,难得是读懂Linux内核代码背后的哲学!难道这就是传说中的藏在Linux代码背后的哲学!天哪!

抛开这见鬼的哲学吧.让我们从一个伟大的例子去认识模块.这就是传说中的"Hello World!",这个梦幻般的名字我们看过无数次了,每一次她出现在眼前,就意味着我们开始接触一种新的计算机语言了,或者,如此刻,开始描述一个新的故事.

请看下面这段代码,她就是 Linux 下的一个最简单的模块.当你安装这个模块的时候,她会用她特有的语言向你表白,"Hello,world!",千真万确,她没有说"Honey,I love you!",虽然,她可以这

么说,如果你要求她这么说.而后来你卸载了这个模块,你无情抛弃了她,她很伤心,她很绝望,但她没有抱怨,她只是淡淡地说,"Goodbye,cruel world!"(再见,残酷的世界!)

++++++++++++++++++hello.c++++++++++++++++++++

```
 1 #include <linux/init.h>   /* Needed for the macros */
 2 #include <linux/module.h> /* Needed for all modules */
 3 MODULE_LICENSE("Dual BSD/GPL");
 4 MODULE_AUTHOR("fudan_abc");
 5
 6 static int __init hello_init(void)
 7 {
 8         printk(KERN_ALERT "Hello, world!\n");
 9         return 0;
10 }
11
12 static void __exit hello_exit(void)
13 {
14         printk(KERN_ALERT "Goodbye, cruel world\n");
15 }
16
17 module_init(hello_init);
18 module_exit(hello_exit);
```

++

你需要使用 module_init()和 module_exit(),你可以称她们为函数,不过实际上她们是一些宏(macro),现在你可以不用去知道她们背后的故事,只需要知道,在 Linux Kernel 2.6 的世界里,你写的任何一个模块都需要使用她们来初始化或退出,或者说注册以及后来的注销.当你用 module_init()为一个模块注册了之后,在你使用 insmod 这个命令去安装的时候,module_init()注册的函数将会被执行,而当你用 rmmod 这个命令去卸载一个模块的时候,module_exit()注册的函数将会被执行.module_init()被称为驱动程序的初始化入口(driver initialization entry point).

怎么样演示以上代码的运行呢?没错,你需要一个 Makefile.

++++++++++++++++++++Makefile++++++++++++++++++++++++++

```
 1 # To build modules outside of the kernel tree, we run "make"
 2 # in the kernel source tree; the Makefile these then includes this
 3 # Makefile once again.
 4 # This conditional selects whether we are being included from the
 5 # kernel Makefile or not.
 6 ifeq ($(KERNELRELEASE),)
 7
 8     # Assume the source tree is where the running kernel was built
 9     # You should set KERNELDIR in the environment if it's elsewhere
10     KERNELDIR ?= /lib/modules/$(shell uname -r)/build
11     # The current directory is passed to sub-makes as argument
12     PWD := $(shell pwd)
```

```
13
14 modules:
15        $(MAKE) -C $(KERNELDIR) M=$(PWD) modules
16
17 modules_install:
18        $(MAKE) -C $(KERNELDIR) M=$(PWD) modules_install
19
20 clean:
21        rm -rf *.o *~ core .depend .*.cmd *.ko *.mod.c .tmp_versions
22
23 .PHONY: modules modules_install clean
24
25 else
26     # called from kernel build system: just declare what our modules are
27     obj-m := hello.o
28 endif
```

++

在 lwn 上可以找到这个例子,你可以把以上两个文件放在你的某个目录下,然后执行 make,也许你不一定能成功,因为 LK 2.6 要求你编译模块之前,必须先在内核源代码目录下执行 make,换言之,你必须先配置过内核,执行过 make,然后才能 make 你自己的模块.原因我就不细说了,你按着她要求的这么去做就行了.在内核顶层目录 make 过之后,你就可以在你当前放置 Makefile 的目录下执行 make 了.Ok,make 之后你就应该看到一个叫做 hello.ko 的文件生成了,恭喜你,这就是你将要测试的模块.

执行命令,

#insmod hello.ko

同时在另一个窗口,用命令 tail -f /var/log/messages 察看日志文件,你会看到

Hello world 被打印了出来.

再执行命令,

#rmmod hello.ko

此时,在另一窗口你会看到 Goodbye,cruel world!被打印了出来.

到这里,我该恭喜你,因为你已经能够编写 Linux 内核模块了.这种感觉很美妙,不是吗?你可以嘲笑秦皇汉武略输文采唐宗宋祖稍逊风骚,还可以嘲笑一代天骄成吉思汗只识弯弓射大雕了. 是的,Twins 姐姐(s)告诉我们,只要我喜欢,还有什么不可以.

日后我们会看到,2.6 内核中,每个模块都是以 module_init 开始,以 module_exit 结束.大多数来说没有必要知道这是为什么,记住就可以了,相信每一个对 Linux 有一点常识的人都会知道这一点的,对大多数人来说,这就像是 1+1 为什么等于 2 一样,就像是两点之间最短的是直线,不需要证明,如果一定要证明两点之间直线最短,可以扔一块骨头在 B 点,让一条狗从 A 点出发,你会发现狗走的是直线,是的,狗都知道,你还能不知道吗?

想到达明天现在就要启程

既然知道了怎么编写一个模块,那么编写设备驱动程序自然也就不难了.我相信,每一个会写模块的人都不会觉得写设备驱动有困难.对自己行不行不确定的话,可以去问一下葛优,他准说:"(神州行),我看行."

真的,我没说假话.写驱动不是什么难事,你完全可以很自信的说,你已经可以写 Device Driver 了.对,没错,飘柔,就这么自信.

前面说了每一个模块都是以 module_init 开始,以 module_exit 结束,那么我们就来看一下 U 盘的驱动的这个模块.在茫茫人海中,我们很容易找到这个文件:drivers/usb/storage/usb.c,在这个文件中又不难发现下面这段:

```
/***********************************************************************
 * Initialization and registration
 ***********************************************************************/

static int __init usb_stor_init(void)
{
	int retval;
	printk(KERN_INFO "Initializing USB Mass Storage driver...\n");

	/* register the driver, return usb_register return code if error */
	retval = usb_register(&usb_storage_driver);
	if (retval == 0)
		printk(KERN_INFO "USB Mass Storage support registered.\n");

	return retval;
}

static void __exit usb_stor_exit(void)
{
	US_DEBUGP("usb_stor_exit() called\n");

	/* Deregister the driver
	 * This will cause disconnect() to be called for each
	 * attached unit
	 */
	US_DEBUGP("-- calling usb_deregister()\n");
	usb_deregister(&usb_storage_driver) ;
}

```

```
1084 module_init(usb_stor_init);
1085 module_exit(usb_stor_exit);
```

其实,module_init/module_exit 只是一个宏,通常写模块的人为了彰显自己的个性,会给自己的初始化函数和注销函数另外起个名字,比如这里 module_init(usb_stor_init)以及 module_exit(usb_stor_exit)实际上就是告诉这个世界,真正的函数是 usb_stor_init 和 usb_stor_exit.这种伎俩在 Linux 内核代码中屡见不鲜.见多了也就不必大惊小怪了,天要下雨娘要嫁人,随她去吧.我们下面当然就从 usb_stor_init 正式开始我们的探索之旅.

外面的世界很精彩

看代码之前,我曾经认真的思考过这么一个问题,我需要关注的仅仅是 drivers/usb/storage/目录下面那相关的 3000 多行代码吗?就是这样几个文件就能让一个个不同的 U 盘在 Linux 下面工作起来吗? 像一开始那样把这个目录比作一个小城的话,也许,城里的月光很漂亮,她能够把人的梦照亮,能够温暖人的心房.但我们真的就能厮守在这个城里,一生一世吗?

很不幸,问题远不是这样简单.外面的世界很精彩,作为 U 盘,她需要与 usb core 打交道,需要与 scsi core 打交道,需要与内存管理单元打交道,还有内核中许许多多其它模块打交道.外面的世界很大,远比我们想象的大.

什么是 usb core?她负责实现一些核心的功能,为别的设备驱动程序提供服务,比如申请内存,比如实现一些所有的设备都会需要的公共的函数,事实上,在 usb 的世界里,一个普通的设备要正常的工作,除了要有设备本身以外,还需要有一个叫做控制器的冬冬,老外把它叫做 host controller,和这个控制器相连接在一起的有另一个咚咚,她叫 root hub,hub 我们应该不会陌生,在大学里,有的宿舍里网口有限,但是我们这一代人上大学基本上是每人一台电脑,所以网口不够,于是有人会使用 hub,让多个人共用一个网口,这是以太网上的 hub,而 usb 的世界里同样有 hub,其实原理是一样的,任何支持 usb 的电脑不会说只允许你只能一个时刻使用一个 usb 设备,比如你插入了 u 盘,你同样还可以插入 usb 键盘,还可以再插一个 usb 鼠标,因为你会发现你的电脑里并不只是一个 usb 接口.这些口实际上就是所谓的 hub 口.而现实中经常是让一个 usb 控制器和一个 hub 绑定在一起,专业一点说叫集成,而这个 hub 也被称作 root hub,换言之,和 usb 控制器绑定在一起的 hub 就是系统中最根本的 hub,其它的 hub 可以连接到她这里,然后可以延伸出去,外接别的设备,当然也可以不用别的 hub,让 usb 设备直接接到 root hub 上.hub 干嘛用的我们知道了,那么 usb host controller 本身是干什么用的呢?controller,控制器,顾名思义,用于控制,控制什么,控制所有的 usb 设备的通信.通常计算机的 cpu 并不是直接和 usb 设备打交道,而是和控制器打交道,他要对设备做什么,他会告诉控制器,而不是直接把指令发给设备,然后控制器再去负责处理这件事情,他会去指挥设备执行命令,而 cpu 就不用管剩下的事情,他还是该干嘛干嘛去,控制器替他去完成剩下的事情,事情办完了再通知 cpu.否则让 cpu 去盯着每一个设备做每一件事情,那是不现实的,那就好比让一个学院的院长去盯着我们每一个本科生上课,去管理我们的出勤,只能说,不现实.所以我们就被分成了几个系,通常院长有什么指示直接跟各系领导说就可以了,如果他要和三个系主任说事情,他即使不把三个人都召集起来开个会,也可以给三个人各打一个电话,打完电话他就忙他自己的事情去了,比如去和他带的女硕士风花雪月.而三个系主任就会去安排下面的人去执行具体的任务,完了之后他们就会像院长汇报.

所以,Linux 内核开发者们,专门写了一些代码,并美其名曰 usb core.时代总在发展,当年胖杨贵妃照样迷死唐明皇,而如今人们欣赏的则是林志玲这样的魔鬼身材.同样,早期的 Linux 内核,其结构并不是如今天这般有层次感,远不像今天这般错落有致,那时候 drivers/usb/这个目录下边放了很多很多文件,usb core 与其他各种设备的驱动程序的代码都堆砌在这里,后来,怎奈世间万千的变幻,总爱把有情的人分两端.于是在 drivers/usb/目录下面出来了一个 core 目录,就专门放

一些核心的代码,比如初始化整个 usb 系统,初始化 root hub,初始化 host controller 的代码,再后来甚至把 host controller 相关的代码也单独建了一个目录,叫 host 目录,这是因为 usb host controller 随着时代的发展,也开始有了好几种,不再像刚开始那样只有一种,所以呢,设计者们把一些 host controller 公共的代码仍然留在 core 目录下,而一些各 host controller 单独的代码则移到 host 目录下面让负责各种 host controller 的人去维护,常见的 host controller 有三种,分别叫做 EHCI,UHCI,OHCI,所以这样,出来了三个概念,usb core,usb host,usb device,即原本是一家人,却被活生生的分成了两岸三地...的确,现实总是很无奈,然而,心若知道灵犀的方向,哪怕不能够朝夕相伴?没错,usb 通信的灵魂就是 usb 协议. usb 协议将是所有 usb 设备和 usb 主机所必须遵循的游戏规则.这种规则也很自然的体现在了代码中.于是,我们需要了解的不仅仅是 drivers/usb/storage/目录下面的冬冬,还得去了解那外面的世界,虽然,只需要了解一点点.

未曾开始却似结束

还是回到那个初始化函数吧,usb_stor_init,看了它的代码每一个人的心中都有一种莫名的兴奋,因为它太短了,就那么几行,除了两个 printk 语句以外,就是一个函数的调用,usb_register.

printk 不用我说,每一个有志青年都该知道,就算没见过 printk 也该见过 printf 吧,否则的话,你扪心自问,你对得起谭浩强大哥吗?在谭浩强大哥的带领下我们学会了用 #include<stdio.h>->main()->printf()来打印 hello,world!从而向全世界展示了我们懂 C 语言.而 stdio.h 就是一个 C 库,printf 是一个函数,来自函数库,可是内核中没有标准 C 库,所以开发者们自己准备了一些函数,专门用于内核代码中,所以就出来了一个 printk,printk 的"k"就是 kernel,内核.所以我们只要把它当作 printf 的兄弟即可,如果感兴趣,可以去研究一下 printk 的特点,她和 printf 多少有些不同,但基本思想是一样的.所以我们就不多讲了,当然驱动程序中所有的 printk 语句对 U 盘的工作都没有什么用,她无非是打出来给我们看的,或者说打印给用户看,或者呢,打印给开发者看,特别是开发者要调试程序的时候,就会很有用.

于是我们更开心了,不用看 printk 的话,那就只有一个函数调用了,usb_register.这个函数是干嘛的?首先这个函数正是来自 usb core.凡是 usb 设备驱动,都要调用这个函数来向 usb core 注册,从而让 usb core 知道有这么一个设备.这就像政府规定,一对夫妻结婚要到相关部门那里去登记是一样的,我们无需知道政府是如何管理的,只需要知道去政府那里登记即可.

这样,insmod 的时候,usb_stor_init 这个函数会被调用,初始化就算完成了.于是设备就开始工作了...而当我们 rmmod 的时候,usb_stor_exit 这个函数会被调用,我们发现,这个函数也很短,我们能看出来,US_DEBUG 也就是打印一些咚咚,因此,这里实际上也就是调用了一个函数 usb_deregister(),她和 usb_register()是一对,完成了注销的工作,从此设备就从 usb core 中消失了.于是我们惊人的发现,编写设备驱动竟是如此的简单,驱动程序真的就这么结束了?...

这一切,不禁让人产生了一种幻觉,让人分不清故事从哪里开始,又从哪里结束,一切都太短暂了.仿佛开始在结束的时候开始,而结束却在开始的时候就早已结束.

真的吗?

答案是否定的.孔子已经教育过我们,不光要看懂代码,更要理解代码背后的哲学.

所以我们在继续之前,先来看看这里到底有什么哲学.而这,就是伟大的 Linux Kernel 2.6 中的统一的设备模型.

我们并无意去详细介绍 2.6 中的设备模型,但是不懂设备模型又怎能说自己懂设备驱动呢?读代码的人,写代码的人,都要知道,什么是设备驱动?什么又是设备?设备和驱动之间究竟是什么关系?设备如何与计算机主机联系起来?我相信在中关村买盗版光盘的哥们儿也能回答这个问题.计算机世界里,设备有很多种类,比如 PCI 设备,比如 ISA 设备,再比如 SCSI 设备,再比如我们这里的 USB 设备.为设备联姻的是总线,是他把设备连入了计算机主机.但是与其说设备是嫁给了计算机主机,倒不如说设备是嫁给了设备驱动程序.很显然,在计算机世界里,无论风里雨里,陪伴着设备的正是驱动程序.

唯一的遗憾是,计算机中的设备和驱动程序的关系却并非如可乐和拉环的关系那样,一对一.然而世上又有多少事情总能如人愿呢.

狂欢是一群人的孤单

Linux 设备模型中三个很重要的概念就是总线,设备,驱动.即 bus,device,driver,而实际上内核中也定义了这么一些数据结构,他们是 struct bus_type,struct device,struct device_driver,这三个重要的数据结构都来自一个地方,include/linux/device.h.我们知道总线有很多种,pci 总线,scsi 总线,usb 总线,所以我们会看到 Linux 内核代码中出现 pci_bus_type,scsi_bus_type,usb_bus_type,他们都是 struct bus_type 类型的变量.而 struct bus_type 结构中两个非常重要的成员就是 struct kset drivers 和 struct kset devices.kset 和另一个叫做 kobject 正是 Linux Kernel 2.6 中设备模型的基本元素,但此处我们却不愿多讲,因为暂时不用去认识他们.我们的生命中会遇见许许多多的人和事,但更多的人和事与我们只是擦肩而过,只是我们生命中的过客而已.在我们人生的电影中,他们也许只有一个镜头,甚至那一个镜头后来也被剪辑掉了.这里我们只需要知道,drivers 和 devices 的存在,让 struct bus_type 与两个链表联系了起来,一个是 devices 的链表,一个是 drivers 的链表,也就是说,知道一条总线所对应的数据结构,就可以找到这条总线所关联的设备有哪些,又有哪些支持这类设备的驱动程序.

而要实现这些,就要求每次出现一个设备就要向总线汇报,或者说注册,每次出现一个驱动,也要向总线汇报,或者说注册.比如系统初始化的时候,会扫描连接了哪些设备,并为每一个设备建立起一个 struct device 的变量,每一次有一个驱动程序,就要准备一个 struct device_driver 结构的变量.把这些变量统统加入相应的链表,device 插入 devices 链表,driver 插入 drivers 链表. 这样通过总线就能找到每一个设备,每一个驱动.

然而,假如计算机里只有设备却没有对应的驱动,那么设备无法工作.反过来,倘若只有驱动却没有设备,驱动也起不了任何作用.在他们遇见彼此之前,双方都如同路埂的野草,一个飘啊飘,一个摇啊摇,谁也不知道未来在哪里,只能在生命的风里飘摇.于是总线上的两张表里就慢慢的就挂上了那许多孤单的灵魂.devices 开始多了,drivers 开始多了,他们像是两个来自世界,devices 们彼此取暖,drivers 们一起狂欢,但他们有一点是相同的,都只是在等待属于自己的那个另一半.

看代码的我,一直好奇的想知道,他们是否和我们现实中一样,有些人注定是等别人,而有些人是注定被人等的.

总线,设备,和驱动(上)

struct bus_type 中为 devices 和 drivers 准备了两个链表,而代表 device 的结构体 struct device 中又有两个成员,struct bus_type *bus 和 struct device_driver *driver,同样,代表 driver 的结构体 struct device_driver 同样有两个成员,struct bus_type *bus 和 struct list_head devices,struct device 和 struct device_driver 的定义和 struct bus_type 一样,在 include/linux/device.h 中.凭一种男人的直觉,可以知晓,struct device 中的 bus 记录的是这个设备连在哪条总线上,driver 记录的是这个设备用的是哪个驱动,反过来,struct device_driver 中的 bus 代表的也是这个驱动属于哪条总线,devices 记录的是这个驱动支持的那些设备,没错,是 devices(复数),而不是 device(单数),因为一个驱动程序可以支持一个或多个设备,反过来一个设备则只会绑定给一个驱动程序.

于是我们想知道,关于 bus,关于 device,关于 driver,他们是如何建立联系的呢?换言之,这三个数据结构中的指针是如何被赋值的?绝对不可能发生的事情是,一旦为一条总线申请了一个 struct bus_type 的数据结构之后,它就知道它的 devices 链表和 drivers 链表会包含哪些东西,这些咚咚一定不会是先天就有的,只能是后天填进来的.而具体到 usb 系统,完成这个工作的就是 usb core.usb core 的代码会进行整个 usb 系统的初始化,比如申请 struct bus_type usb_bus_type,然后会扫描 usb 总线,看线上连接了哪些 usb 设备,或者说 root hub 上连了哪些 usb 设备,比如说连了一个 usb 键盘,那么就为它准备一个 struct device,根据它的实际情况,为这个 struct device 赋值,并插入 devices 链表中来.又比如 root hub 上连了一个普通的 hub,那么除了要为这个 hub 本身准备一个 struct device 以外,还得继续扫描看这个 hub 上是否又连了别的设备,有的话继续重复之前的事情,这样一直进行下去,直到完成整个扫描,最终就把 usb_bus_type 中的 devices 链表给建立了起来.

那么 drivers 链表呢?这个就不用 bus 方面主动了,而该由每一个 driver 本身去 bus 上面登记,或者说挂牌.具体到 usb 系统,每一个 usb 设备的驱动程序都会有一个 struct usb_driver 结构体,其代码如下,来自 include/linux/usb.h

```
485 /* ------------------------------------------------------------------------ */
486
487 /**
488  * struct usb_driver - identifies USB driver to usbcore
489  * @owner: Pointer to the module owner of this driver; initialize
490  *      it using THIS_MODULE.
491  * @name: The driver name should be unique among USB drivers,
492  *      and should normally be the same as the module name.
493  * @probe: Called to see if the driver is willing to manage a particular
494  *      interface on a device.  If it is, probe returns zero and uses
495  *      dev_set_drvdata() to associate driver-specific data with the
496  *      interface.  It may also use usb_set_interface() to specify the
497  *      appropriate altsetting.  If unwilling to manage the interface,
498  *      return a negative errno value.
499  * @disconnect: Called when the interface is no longer accessible, usually
500  *      because its device has been (or is being) disconnected or the
501  *      driver module is being unloaded.
```

```
502  * @ioctl: Used for drivers that want to talk to userspace through
503  *      the "usbfs" filesystem.  This lets devices provide ways to
504  *      expose information to user space regardless of where they
505  *      do (or don't) show up otherwise in the filesystem.
506  * @suspend: Called when the device is going to be suspended by the
system.
507  * @resume: Called when the device is being resumed by the system.
508  * @id_table: USB drivers use ID table to support hotplugging.
509  *      Export this with MODULE_DEVICE_TABLE(usb,...).  This must be set
510  *      or your driver's probe function will never get called.
511  * @driver: the driver model core driver structure.
512  *
513  * USB drivers must provide a name, probe() and disconnect() methods,
514  * and an id_table.  Other driver fields are optional.
515  *
516  * The id_table is used in hotplugging.  It holds a set of descriptors,
517  * and specialized data may be associated with each entry.  That table
518  * is used by both user and kernel mode hotplugging support.
519  *
520  * The probe() and disconnect() methods are called in a context where
521  * they can sleep, but they should avoid abusing the privilege.  Most
522  * work to connect to a device should be done when the device is opened,
523  * and undone at the last close.  The disconnect code needs to address
524  * concurrency issues with respect to open() and close() methods, as
525  * well as forcing all pending I/O requests to complete (by unlinking
526  * them as necessary, and blocking until the unlinks complete).
527  */
528 struct usb_driver {
529       struct module *owner;
530
531       const char *name;
532
533       int (*probe) (struct usb_interface *intf,
534                     const struct usb_device_id *id);
535
536       void (*disconnect) (struct usb_interface *intf);
537
538       int (*ioctl) (struct usb_interface *intf, unsigned int code, void *buf);
539
540       int (*suspend) (struct usb_interface *intf, u32 state);
541       int (*resume) (struct usb_interface *intf);
542
543       const struct usb_device_id *id_table;
544
```

```
545         struct device_driver driver;
546 };
547 #define to_usb_driver(d) container_of(d, struct usb_driver, driver)
```

看似很长一段,实际上也就是注释为主.而此刻我们只需注意到其中的 struct device_driver driver 这个成员,usb core 为每一个设备驱动准备了一个函数,让它把自己的这个 struct device_driver driver 插入到 usb_bus_type 中的 drivers 链表中去.而这个函数正是我们此前看到的 usb_register.而与之对应的 usb_deregister 所从事的正是与之相反的工作,把这个结构体从 drivers 链表中删除.可以说,usb core 的确是用心良苦,为每一个 usb 设备驱动做足了功课,正因为如此,作为一个实际的 usb 设备驱动,它在初始化阶段所要做的事情就很少,很简单了,直接调用 usb_register 即可.事实上,没有人是理所当然应该为你做什么的,但 usb core 这么做了.所以每一个写 usb 设备驱动的人应该铭记,usb device driver 绝不是一个人在工作,在他身后,是 usb core 所提供的默默无闻又不可或缺的支持.

总线,设备,和驱动(下)

bus 上的两张链表记录了每一个 device 和 driver,那么 device 和 driver 这两者之间又是如何联系起来的呢?此刻,必须抛出这样一个问题,先有 device 还是 driver?

很久很久以前,在那激情燃烧的岁月里,先有的是 device,每一个要用的 device 在计算机启动之前就已经插好了,插放在它应该在的位置上,然后计算机启动,然后操作系统开始初始化,总线开始扫描设备,每找到一个设备,就为其申请一个 struct device 结构,并且挂入总线中的 devices 链表中来,然后每一个驱动程序开始初始化,开始注册其 struct device_driver 结构,然后它去总线的 devices 链表中去寻找(遍历),去寻找每一个还没有绑定 driver 的设备,即 struct device 中的 struct device_driver 指针仍为空的设备,然后它会去观察这种设备的特征,看是否是他所支持的设备,如果是,那么调用一个叫做 device_bind_driver 的函数,然后他们就结为了秦晋之好.换句话说,把 struct device 中的 struct device_driver driver 指向这个 driver,而 struct device_driver driver 把 struct device 加入他的那张 struct list_head devices 链表中来.就这样,bus,device,和 driver,这三者之间或者说他们中的两两之间,就给联系上了.知道其中之一,就能找到另外两个.一荣俱荣,一损俱损.

但现在情况变了,在这红莲绽放的日子里,在这樱花伤逝的日子里,出现了一种新的名词,叫热插拔.device 可以在计算机启动以后在插入或者拔出计算机了.因此,很难再说是先有 device 还是先有 driver 了.因为都有可能.device 可以在任何时刻出现,而 driver 也可以在任何时刻被加载,所以,出现的情况就是,每当一个 struct device 诞生,它就会去 bus 的 drivers 链表中寻找自己的另一半,反之,每当一个一个 struct device_driver 诞生,它就去 bus 的 devices 链表中寻找它的那些设备.如果找到了合适的,那么 ok,和之前那种情况一下,调用 device_bind_driver 绑定好.如果找不到,没有关系,等待吧,等到昙花再开,等到风景看透,心中相信,这世界上总有一个人是你所等的,只是还没有遇到而已.

好,继续,事实上,完善这个三角关系,正是每一个设备驱动初始化阶段所完成的重要使命之一.让我们还是回到代码中来,usb_register 这个函数调用是调用了,但是传递给他的参数是什么呢?

我们注意到,那句调用是这样子的,

```
1064         /* register the driver, return usb_register return code if error */
1065         retval = usb_register(&usb_storage_driver);
```

是的,传递了一个叫做 usb_storage_driver 的家伙,这是什么?同一文件中,drivers/usb/storage/usb.c:

```
232 struct usb_driver usb_storage_driver = {
233         .owner =        THIS_MODULE,
234         .name =         "usb-storage",
235         .probe =        storage_probe,
236         .disconnect =   storage_disconnect,
237         .id_table =     storage_usb_ids,
238 };
```

可以看到这里定义了一个 struct usb_driver 的结构体变量,usb_storage_driver,关于 usb_driver 我们上节已经说过了,当时主要说的是其中的成员 driver,而眼下要讲的则是另外几个成员.首先,.owner 和.name 这两个没啥好多说的,owner 这玩艺是用来给模块计数的,每个模块都这么用,赋值总是 THIS_MODULE,而 name 就是这个模块的名字,usb core 会处理它,所以如果这个模块正常被加载了的话,使用 lsmod 命令能看到一个叫做 usb-storage 的模块名.重点要讲一讲,.probe 和.disconnect 以及这个 id_table.

我是谁的他?

probe,disconnect,id_table,这三个咚咚中首先要登场亮相的是 id_table,它是干嘛用的呢?

我们说过,一个 device 只能绑定一个 driver,但 driver 并非只能支持一种设备,道理很简单,比如我有两块 U 盘,那么我可以一起都插入,但是我只需要加载一个模块,usb-storage,没听说过插入两块 U 盘就得加载两次驱动程序的,除非这两块 U 盘本身就得使用不同的驱动程序.也正是因为一个模块可以被多个设备共用,才会有模块计数这么一个说法.

ok,既然一个 driver 可以支持多个 device,那么当发现一个 device 的时候,如何知道哪个 driver 才是她的 Mr.Right 呢?这就是 id_table 的用处,让每一个 struct usb_driver 准备一张表,里边注明该 driver 支持哪些设备,这总可以了吧.如果你这个设备属于这张表里的,那么 ok,绑定吧,如果不属于这张表里的,那么不好意思,您请便.哪凉快上哪去.

来自 struct usb_driver 中的 id_table,

const struct usb_device_id *id_table;

实际上是一个指针,一个 struct usb_device_id 结构体的指针,当然赋了值以后就是代表一个数组名了,正如我们在定义 struct usb_driver usb_storage_driver 中所赋的值那样,.id_table=storage_usb_ids,那好,我们来看一下 usb_device_id 这究竟是怎样一个结构体.

struct usb_device_id 来自 include/linux/mod_devicetable.h,

```
40 /*
41  * Device table entry for "new style" table-driven USB drivers.
42  * User mode code can read these tables to choose which modules to load.
43  * Declare the table as a MODULE_DEVICE_TABLE.
44  *
45  * A probe() parameter will point to a matching entry from this table.
46  * Use the driver_info field for each match to hold information tied
```

```
 47  * to that match:  device quirks, etc.
 48  *
 49  * Terminate the driver's table with an all-zeroes entry.
 50  * Use the flag values to control which fields are compared.
 51  */
 52
 53 /**
 54  * struct usb_device_id - identifies USB devices for probing and hotplugging
 55  * @match_flags: Bit mask controlling of the other fields are used to match
 56  *      against new devices.  Any field except for driver_info may be used,
 57  *      although some only make sense in conjunction with other fields.
 58  *      This is usually set by a USB_DEVICE_*() macro, which sets all
 59  *      other fields in this structure except for driver_info.
 60  * @idVendor: USB vendor ID for a device; numbers are assigned
 61  *      by the USB forum to its members.
 62  * @idProduct: Vendor-assigned product ID.
 63  * @bcdDevice_lo: Low end of range of vendor-assigned product version
numbers.
 64  *      This is also used to identify individual product versions, for
 65  *      a range consisting of a single device.
 66  * @bcdDevice_hi: High end of version number range.  The range of product
 67  *      versions is inclusive.
 68  * @bDeviceClass: Class of device; numbers are assigned
 69  *      by the USB forum.  Products may choose to implement classes,
 70  *      or be vendor-specific.  Device classes specify behavior of all
 71  *      the interfaces on a devices.
 72  * @bDeviceSubClass: Subclass of device; associated with bDeviceClass.
 73  * @bDeviceProtocol: Protocol of device; associated with bDeviceClass.
 74  * @bInterfaceClass: Class of interface; numbers are assigned
 75  *       by the USB forum.  Products may choose to implement classes,
 76  *       or be vendor-specific.  Interface classes specify behavior only
 77  *       of a given interface; other interfaces may support other classes.
 78  * @bInterfaceSubClass: Subclass of interface; associated with bInterfaceClass.
 79  * @bInterfaceProtocol: Protocol of interface; associated with bInterfaceClass.
 80  * @driver_info: Holds information used by the driver.  Usually it holds
 81  *       a pointer to a descriptor understood by the driver, or perhaps
 82  *       device flags.
 83  *
 84  * In most cases, drivers will create a table of device IDs by using
 85  * USB_DEVICE(), or similar macros designed for that purpose.
 86  * They will then export it to userspace using MODULE_DEVICE_TABLE(),
 87  * and provide it to the USB core through their usb_driver structure.
 88  *
 89  * See the usb_match_id() function for information about how matches are
```

```
 90  * performed.  Briefly, you will normally use one of several macros to help
 91  * construct these entries.  Each entry you provide will either identify
 92  * one or more specific products, or will identify a class of products
 93  * which have agreed to behave the same.  You should put the more specific
 94  * matches towards the beginning of your table, so that driver_info can
 95  * record quirks of specific products.
 96  */
 97 struct usb_device_id {
 98         /* which fields to match against? */
 99         __u16           match_flags;
100
101         /* Used for product specific matches; range is inclusive */
102         __u16           idVendor;
103         __u16           idProduct;
104         __u16           bcdDevice_lo;
105         __u16           bcdDevice_hi;
106
107         /* Used for device class matches */
108         __u8            bDeviceClass;
109         __u8            bDeviceSubClass;
110         __u8            bDeviceProtocol;
111
112         /* Used for interface class matches */
113         __u8            bInterfaceClass;
114         __u8            bInterfaceSubClass;
115         __u8            bInterfaceProtocol;
116
117         /* not matched against */
118         kernel_ulong_t  driver_info;
119 };
```

实际上这个结构体对每一个 usb 设备来说,就相当于是她的身份证,记录了她的一些基本信息,通常我们的身份证上会记录我们的姓名,性别,出生年月,户口地址等等,而 usb 设备她也有她需要记录的信息,以区分她和别的 usb 设备,比如 Vendor-厂家,Product-产品,以及其他一些比如产品编号,产品的类别,遵循的协议,这些都会在 usb 的规范里边找到对应的冬冬.暂且先不细说.

于是我们知道,一个 usb_driver 会把它的这张 id 表去和每一个 usb 设备的实际情况进行比较,如果该设备的实际情况和这张表里的某一个 id 相同,准确地说,只有这许多特征都吻合,才能够把一个 usb device 和这个 usb driver 进行绑定,这些特征哪怕差一点也不行.就像我们每个人都是一道弧,都在不停寻找能让彼此嵌成完整的圆的另一道弧,事实却是,每个人对ㄖ(PI)的理解不尽相同,而圆心能否重合,或许只有痛过才知道.差之毫厘,失之交臂.

那么 usb 设备的实际情况是什么时候建立起来的?嗯,在介绍.probe 指针之前有必要先谈一谈另一个数据结构了,她就是 struct usb_device.

从协议中来,到协议中去(上)

在 struct usb_driver 中,.probe 和.disconnect 的原型如下:

int (*probe) (struct usb_interface *intf,const struct usb_device_id *id);

void (*disconnect) (struct usb_interface *intf);

我们来看其中的参数,struct usb_device_id 这个不用说了,刚才已经介绍过,那么 struct usb_interface 从何而来?还是让我们先从 struct usb_device 说起.

我们知道每一个 device 对应一个 struct device 结构体变量,但是 device 不可能是万能的,生命是多样性的,就像我们可以用"人"来统称全人类,但是分的细一点,又有男人和女人的区别,那么 device 也一样,由于有各种各样的设备,于是又出来了更多的词汇(数据结构),比如针对 usb 设备,开发者们设计了一个叫做 struct usb_device 的结构体.她定义于 include/linux/usb.h,

```
294 /*
295  * struct usb_device - kernel's representation of a USB device
296  *
297  * FIXME: Write the kerneldoc!
298  *
299  * Usbcore drivers should not set usbdev->state directly.  Instead use
300  * usb_set_device_state().
301  */
302 struct usb_device {
303         int             devnum;         /* Address on USB bus */
304         char            devpath [16];   /* Use in messages: /port/port/... */
305         enum usb_device_state   state;  /* configured, not attached, etc */
306         enum usb_device_speed   speed;  /* high/full/low (or error) */
307
308         struct usb_tt   *tt;            /* low/full speed dev, highspeed hub */
309         int             ttport;         /* device port on that tt hub */
310
311         struct semaphore serialize;
312
313         unsigned int toggle[2];         /* one bit for each endpoint ([0] = IN, [1] =
OUT) */
314         int epmaxpacketin[16];          /* INput endpoint specific maximums */
315         int epmaxpacketout[16];         /* OUTput endpoint specific maximums
*/
316
317         struct usb_device *parent;      /* our hub, unless we're the root */
318         struct usb_bus *bus;            /* Bus we're part of */
319
320         struct device dev;              /* Generic device interface */
321
```

```
322         struct usb_device_descriptor descriptor;/* Descriptor */
323         struct usb_host_config *config; /* All of the configs */
324         struct usb_host_config *actconfig;/* the active configuration */
325
326         char **rawdescriptors;          /* Raw descriptors for each config */
327
328         int have_langid;                /* whether string_langid is valid yet */
329         int string_langid;              /* language ID for strings */
330
331         void *hcpriv;                   /* Host Controller private data */
332
333         struct list_head filelist;
334         struct dentry *usbfs_dentry;    /* usbfs dentry entry for the device */
335
336         /*
337          * Child devices - these can be either new devices
338          * (if this is a hub device), or different instances
339          * of this same device.
340          *
341          * Each instance needs its own set of data structures.
342          */
343
344         int maxchild;                   /* Number of ports if hub */
345         struct usb_device *children[USB_MAXCHILDREN];
346 };
347 #define to_usb_device(d) container_of(d, struct usb_device, dev)
```

看起来很复杂的一个数据结构,不过我们目前不需要去理解她的每一个成员,不过我们可以看到,其中有一个成员 struct device dev,没错,这就是前面说的那个属于每个设备的 struct device 结构体变量.

实际上,U 盘驱动里边并不会直接去处理这个结构体,因为对于一个 U 盘来说,她就是对应这么一个 struct usb_device 的变量,这个变量由 usb core 负责申请和赋值.但是我们需要记住这个结构体变量,因为日后我们调用 usb core 提供的函数的时候,会把这个变量作为参数传递上去,因为很简单,要和 usb core 交流,总得让人家知道我们是谁吧,比如后来要调用的一个函数,usb_buffer_alloc,它就需要这个参数.

而对 U 盘设备驱动来说,比这个 struct usb_device 更重要的数据结构是,struct usb_interface.走到这一步,我们不得不去了解一点 usb 设备的规范了,或者专业一点说,usb 协议.因为我们至少要知道什么是 usb interface.

从协议中来,到协议中去(中)

星爷说,人有人他妈,妖有妖他妈.说的就是任何事物都有其要遵守的规矩.usb 设备要遵循的就是 usb 协议. 不管是软件还是硬件,在设计的伊始,总是要参考 usb 协议.怎么设计硬件,如何编写软件,不看 usb 协议,谁也不可能凭空想象出来.毕竟不是写小说,有几人能像海岩那样,光凭想象就能写出便衣警察,永不瞑目,玉观音这些经典的爱情加案情的作品来呢.

usb 协议规定了,每个 usb 设备都得有些基本的元素,称为描述符,有四类描述符是任何一种 usb 设备都得有的.他们是,device descriptor,configuration descriptor,interface descriptor,endpoint descriptor.描述符里的冬冬是一个设备出厂的时候就被厂家给固化在设备里了.这种东西不管怎样也改变不了,比如我有个 Intel 的 U 盘,那里边的固有的信息肯定是在 Intel 出厂的时候就被烙在里边了,厂家早已把它的一切,烙上 Intel 印.所以当我插入 U 盘,用 cat /proc/scsi/scsi 命令看一下的话,"Vendor"那一项显示的肯定是 Intel. 关于这几种描述符,usb core 在总线扫描那会就会去读取,会去获得里边的信息,其中,device 描述符描述的是整个 device,注意了,这个 device 和咱们一直讲的 device 和 driver 那里的 device 是不一样的.因为一个 usb device 实际上指的是一种宏观上的概念,它可以是一种多功能的设备,改革开放之后,多功能的东西越来越多了,比如外企常见的多功能一体机,就是集打印机,复印机,扫描仪,传真机于一体的设备, 当然,这不属于 usb 设备,但是 usb 设备当然也有这种情况,比如电台 DJ 可能会用到的,一个键盘,上边带一个扬声器,它们用两个 usb 接口接到 usb hub 上去,而 device 描述符描述的就是这整个设备的特点.那么 configuration 描述符呢,老实说,对我们了解 U 盘驱动真是没有什么意义,但是作为一个有责任心的男人,此刻,我必须为它多说几句,虽然只是很简单的说几句.一个设备可以有一种或者几种配置,这能理解吧?没见过具体的 usb 设备?那么好,手机见过吧,每个手机都会有多种配置,或者说"设定",比如,我的这款,Nokia6300,手机语言,可以设定为 English,繁体中文,简体中文,一旦选择了其中一种,那么手机里边所显示的所有的信息都是该种语言/字体.还有最简单的例子,操作模式也有好几种,标准,无声,会议,etc. 基本上如果我设为"会议",那么就是只振动不发声,要是设为无声,那么就啥动静也不会有,只能凭感觉了,以前去公司面试的话通常就是设为无声,因为觉得振动也不好,让人家面试官听到了还是不合适.那么 usb 设备的配置也是如此,不同的 usb 设备当然有不同的配置了,或者说需要配置哪些东西也会不一样.好了,关于配置,就说这么多,更多的我们暂时也不需要了解了.

对于 usb 设备驱动程序编写者来说,更为关键的是下面两个,interface 和 endpoint.先说,interface.第一句,一个 interface 对应一个 usb 设备驱动程序.没错,还说前边那个例子,一个 usb 设备,两种功能,一个键盘,上面带一个扬声器,两个接口,那这样肯定得要两个驱动程序,一个是键盘驱动程序,一个是音频流驱动程序.道上的兄弟喜欢把这样两个整合在一起的东西叫做一个设备,那好,让他们去叫吧,我门用 interface 来区分这两者行了吧.于是有了我们前面提到的那个数据结构,struct usb_interface.它定义于 include/linux/usb.h:

```
71 /**
72  * struct usb_interface - what usb device drivers talk to
73  * @altsetting: array of interface structures, one for each alternate
74  *      setting that may be selected.  Each one includes a set of
75  *      endpoint configurations.  They will be in no particular order.
76  * @num_altsetting: number of altsettings defined.
77  * @cur_altsetting: the current altsetting.
78  * @driver: the USB driver that is bound to this interface.
79  * @minor: the minor number assigned to this interface, if this
80  *      interface is bound to a driver that uses the USB major number.
81  *      If this interface does not use the USB major, this field should
82  *      be unused.  The driver should set this value in the probe()
83  *      function of the driver, after it has been assigned a minor
84  *      number from the USB core by calling usb_register_dev().
85  * @condition: binding state of the interface: not bound, binding
86  *      (in probe()), bound to a driver, or unbinding (in disconnect())
87  * @dev: driver model's view of this device
```

```
 88  * @class_dev: driver model's class view of this device.
 89  *
 90  * USB device drivers attach to interfaces on a physical device.  Each
 91  * interface encapsulates a single high level function, such as feeding
 92  * an audio stream to a speaker or reporting a change in a volume control.
 93  * Many USB devices only have one interface.  The protocol used to talk to
 94  * an interface's endpoints can be defined in a usb "class" specification,
 95  * or by a product's vendor.  The (default) control endpoint is part of
 96  * every interface, but is never listed among the interface's descriptors.
 97  *
 98  * The driver that is bound to the interface can use standard driver model
 99  * calls such as dev_get_drvdata() on the dev member of this structure.
100  *
101  * Each interface may have alternate settings.  The initial configuration
102  * of a device sets altsetting 0, but the device driver can change
103  * that setting using usb_set_interface().  Alternate settings are often
104  * used to control the the use of periodic endpoints, such as by having
105  * different endpoints use different amounts of reserved USB bandwidth.
106  * All standards-conformant USB devices that use isochronous endpoints
107  * will use them in non-default settings.
108  *
109  * The USB specification says that alternate setting numbers must run from
110  * 0 to one less than the total number of alternate settings.  But some
111  * devices manage to mess this up, and the structures aren't necessarily
112  * stored in numerical order anyhow.  Use usb_altnum_to_altsetting() to
113  * look up an alternate setting in the altsetting array based on its number.
114  */
115 struct usb_interface {
116         /* array of alternate settings for this interface,
117          * stored in no particular order */
118         struct usb_host_interface *altsetting;
119
120         struct usb_host_interface *cur_altsetting;      /* the currently
121                                          * active alternate setting */
122         unsigned num_altsetting;        /* number of alternate settings */
123
124         int minor;                      /* minor number this interface is bound to */
125         enum usb_interface_condition condition;         /* state of binding */
126         struct device dev;              /* interface specific device info */
127         struct class_device *class_dev;
128 };
129 #define to_usb_interface(d) container_of(d, struct usb_interface, dev)
130 #define interface_to_usbdev(intf) \
131         container_of(intf->dev.parent, struct usb_device, dev)
```

嗬,贴这么长一段,怎么又是注释为主啊?知足吧,Linux 代码中注释实在是太少了,等你真的需要认真看某一个模块的时候你就会嫌注释少了.这个结构体是一个贯穿整个 U 盘驱动程序的,所以虽然我们不用去深入了解,但是需要记住,整个 U 盘驱动程序在后面任何一处提到的 struct usb_interface 都是同一个变量,这个变量是在 usb core 总线扫描的时候就申请好了的.我们只需贯彻鲁迅先生的拿来主义即可,直接用就是了.比如前面说过的 storage_probe(struct usb_interface *intf,const struct usb_device_id *id),storage_disconnect(struct usb_interface *intf)这两个函数中的那个参数 intf.

而这里 130 行这个宏-interface_to_usbdev,也会用得着的,顾名思义,就是从一个 struct usb_interface 转换成一个 struct usb_device,我们说过了,有些函数需要的参数就是 struct usb_device,而不是 usb_interface,所以这种转换是经常会用到的,而这个宏,usb core 的设计者们也为我们准备好了,除了感激,我们还能说什么呢?

从协议中来,到协议中去(下)

如果你是急性子,那这时候你一定很想开始看 storage_probe 函数了,因为整个 U 盘的工作就是从这里开始的.不过,莎士比亚说过,磨刀不误砍柴功.不妨继续花点时间,至少把四大关键词中最后一个给弄明白了,

前面我们已经了解了 device,configuration,interface,还剩最后一个 endpoint.USB 通信的最基本的形式就是通过 endpoint,道上的兄弟管这个叫做端点,一个接口有一个或多个端点,而作为像 U 盘这样的存储设备吧,它至少有一个控制端点,两个 bulk 端点.这些端点都是干嘛的?说来话长,真是一言难尽哪.

usb 协议里规定了,usb 设备有四种通信方式,分别是控制传输,中断传输,批量传输,等时传输.其中,等时传输显然是用于音频和视频一类的设备,这类设备期望能够有个比较稳定的数据流,比如你在网上 QQ 视频聊天,肯定希望每分钟传输的图像/声音速率是比较稳定的,不能说这一分钟对方看到你在向她向你深情表白,可是下一分钟却看见画面停滞在那里,只能看到你那傻样一动不动,你说这不浪费感情嘛.所以,每一个有良知的男人都应该知道,usb-storage 里边肯定不会用到等时传输.因为我们只管 copy 一个文件,管它第一秒和第二秒的传输有什么区别,只要几十秒内传完了就 ok.

相比之下,等时传输是四种传输中最麻烦的,所以,U 盘里边用不着,那您就偷着乐去吧.不过我要说,中断传输也用不着,对于 U 盘来说,的确用不着,虽然说 usb mass storage 的协议里边有一个叫做 CBI 的传输协议,CBI 就是 Control/Bulk/Interrupt,即控制/批量/中断,这三种传输都会用到,但这种传输协议并不适用于 U 盘,U 盘使用的是一个叫做 Bulk-Only 的传输协议.使用这种协议的设备只有两种传输方式,一种是批量传输,另一种是控制传输,控制传输是任何一种 usb 设备都必须支持的,它专门用于传输一些控制信息.比如我想查询一下关于这个 interface 的一些信息,那么就用控制传输,而 bulk 传输,它就是 U 盘的主要工作了,读写数据,这种情况就得用 bulk 传输.具体的传输我们后面再讲.

好了,知道了传输方式,就可以来认识 endpoint 了.和 endpoint 齐名的有一个叫做管道,或者有文化的人管这个叫 pipe.endpoint 就是通信的发送或者接收点,你要发送数据,那你只要把数据发送到正确的端点那里就可以了.之所以 U 盘有两个 bulk 端点,是因为端点也是有方向的,一个叫做 Bulk in,一个叫做 Bulk out,从 usb 主机到设备称为 out,从设备到主机称为 in.而管道,实际上只是为了让我们能够找到端点,就相当于我们日常说的邮编地址,比如一个国家,为了通信,我们必须给各个地方取名,完了给各条大大小小的路取名,比如你要揭发你们那里的官员腐败,你要去

上访,你从某偏僻的小县城出发,要到北京来上访,那你的这个端点(endpoint)就是北京,而你得知道你来北京的路线,那这个从你们县城到北京的路线就算一条管道.有人好奇的问了,管道应该有两端吧,一个端点是北京,那另一个端点呢?答案是,这条管道有些特殊,就比如上访,我们只需要知道一端是北京,而另一端是哪里无所谓,因为不管你在哪里你都得到北京来上访.没听说过你在山西你可以上访,你要在宁夏还不能上访了,没这事对吧.

严格来说,管道的另一端应该是 usb 主机,即前面说的那个 host,usb 协议里边也是这么说的,协议里边说 pipes 代表着一种能力,怎样一种能力呢,在主机和设备上的端点之间移动数据,听上去挺玄的.因为事实上,usb 里边所有的数据传输都是有主机发起的.一切都是以主机为核心,各种设备紧紧围绕在主机周围.所以呢,usb core 里边很多函数就是,为了让 usb host 能够正确的完成数据传输或者说传输调度,就得告诉 host 这么一个 pipe,换言之,它得知道自己这次调度的数据是传送给谁或者从谁那里传输过来.不过有人又要问了,比如说我要从 U 盘里读一个文件,那我告诉 usb host 某个端点能有用吗?那个文件又不是存放在一个端点里边,它不该是存放在 U 盘里边吗?这个就只能在后面的代码里去知道了.实际上端点这东西是一个很虚的东西,它的身上充分体现了我国军事思想中的声东击西的想法,即数据本身并不是在端点里,但是看上去却觉得仿佛在端点里.这一切的谜团,让我们在 storage_probe()函数里去慢慢解开吧.

梦开始的地方

对于整个 usb-storage 模块,usb_stor_init()是它的开始,然而,对于 U 盘驱动程序来说,它真正驱使 U 盘工作却是始于 storage_probe().

两条平行线只要相交,就注定开始纠缠一生,不管中间是否短暂分离. usbcore 为设备找到了适合她的驱动程序,或者为驱动程序找到了他所支持的设备,但这只是表明双方的第一印象还可以,但彼此是否真的适合对方还需要进一步的了解.毋庸置疑,了解对方的第一步是,知道她有哪些爱好,她的生日,她的星座,喜欢吃什么,而 U 盘 driver 则会调用函数 storage_probe()去认识对方,她是个什么样的设备,她的生活习惯是?她的联系方式是?这里调用了四个函数 get_device_info,get_protocol,get_transport,get_pipes. 当然还有一些别人并不了解的冬冬你也会知道,比如她的三围,这里对应的就是 usb_stor_Bulk_man_lun().

整个 U 盘驱动这部大戏,由 storage_probe 开始,由 storage_disconnect 结束.其中,storage_probe 这个函数占了相当大的篇幅.我们一段一段来看.这两个函数都来自 drivers/usb/storage/usb.c 中:

```
926 /* Probe to see if we can drive a newly-connected USB device */
927 static int storage_probe(struct usb_interface *intf,
928                          const struct usb_device_id *id)
929 {
930         struct us_data *us;
931         const int id_index = id - storage_usb_ids;
932         int result;
933
934         US_DEBUGP("USB Mass Storage device detected\n");
935
936         /* Allocate the us_data structure and initialize the mutexes */
937         us = (struct us_data *) kmalloc(sizeof(*us), GFP_KERNEL);
```

```
938         if (!us) {
939                 printk(KERN_WARNING USB_STORAGE "Out of memory\n");
940                 return -ENOMEM;
941         }
942         memset(us, 0, sizeof(struct us_data));
```

首先先贴出这么几行,两个参数不用多说了,struct usb_interface 和 struct usb_device_id 的这两个指针都是前面介绍过的,来自 usb core 那一层,我们整个故事里用到的就是这么一个,不是说一会指向你,一会指向他,这两个指针的指向是定下来的.

930 行,最重要的一个数据结构终于在这种神不知鬼不觉的地方惊艳亮相了.整个 usb-storage 模块里边自己定义的数据结构不多,但是 us_data 算一个.这个数据结构是跟随我们的代码一直走下去的,如影随形,几乎处处都可以看见她的身影.先把它的代码贴出来,来自 drivers/usb/storage/usb.h:

```
105 /* we allocate one of these for every device that we remember */
106 struct us_data {
107         /* The device we're working with
108          * It's important to note:
109          *    (o) you must hold dev_semaphore to change pusb_dev
110          */
111         struct semaphore        dev_semaphore;    /* protect pusb_dev */
112         struct usb_device       *pusb_dev;        /* this usb_device */
113         struct usb_interface    *pusb_intf;       /* this interface */
114         struct us_unusual_dev   *unusual_dev;     /* device-filter entry     */
115         unsigned long           flags;            /* from filter initially */
116         unsigned int            send_bulk_pipe;   /* cached pipe values */
117         unsigned int            recv_bulk_pipe;
118         unsigned int            send_ctrl_pipe;
119         unsigned int            recv_ctrl_pipe;
120         unsigned int            recv_intr_pipe;
121
122         /* information about the device */
123         char                    vendor[USB_STOR_STRING_LEN];
124         char                    product[USB_STOR_STRING_LEN];
125         char                    serial[USB_STOR_STRING_LEN];
126         char                    *transport_name;
127         char                    *protocol_name;
128         u8                      subclass;
129         u8                      protocol;
130         u8                      max_lun;
131
132         u8                      ifnum;            /* interface number    */
133         u8                      ep_bInterval;     /* interrupt interval */
134
135         /* function pointers for this device */
```

```
136         trans_cmnd              transport;       /* transport function     */
137         trans_reset             transport_reset; /* transport device reset */
138         proto_cmnd              proto_handler;   /* protocol handler       */
139
140         /* SCSI interfaces */
141         struct Scsi_Host        *host;           /* our dummy host data */
142         struct scsi_cmnd        *srb;            /* current srb         */
143
144         /* thread information */
145         int                     pid;             /* control thread      */
146
147         /* control and bulk communications data */
148         struct urb              *current_urb;    /* USB requests        */
149         struct usb_ctrlrequest  *cr;             /* control requests    */
150         struct usb_sg_request   current_sg;      /* scatter-gather req. */
151         unsigned char           *iobuf;          /* I/O buffer          */
152         dma_addr_t              cr_dma;          /* buffer DMA addresses */
153         dma_addr_t              iobuf_dma;
154
155         /* mutual exclusion and synchronization structures */
156         struct semaphore        sema;            /* to sleep thread on  */
157         struct completion       notify;          /* thread begin/end    */
158         wait_queue_head_t       dev_reset_wait;  /* wait during reset   */
159         wait_queue_head_t       scsi_scan_wait;  /* wait before scanning */
160         struct completion       scsi_scan_done;  /* scan thread end     */
161
162         /* subdriver information */
163         void                    *extra;          /* Any extra data      */
164         extra_data_destructor   extra_destructor;/* extra data
destructor   */
165 };
```

不难发现,Linux 内核中每一个重要的数据结构都很复杂,这体现了内核代码编写者们的一种清高,仿佛不用点复杂的数据结构不足以体现他们是个腕儿.这可就苦了我们这些读代码的了,尤其是中国的学生,毕竟谭浩强的书里边翻多少遍也翻不出这么一变态的数据结构吧.所以,此刻,每一个有志青年都应该倍感责任重大,只有我们国家强大了,我们才能把谭浩强的书籍向全世界推广,从而告诉那些写内核代码的家伙,不要写那么复杂的冬冬,要按照谭浩强的书里的规矩来设计数据结构,来编写代码.这才是造福全人类的做法.不是吗?

先不说这些了,总之,这个令人头疼的数据结构是每一个 device 都有的,换句话说,我们会为每一个 device 申请一个 us_data,因为这个结构里边的冬冬我们之后一直会用得着的.至于怎么用,每个成员什么意思,以后用上了再细说.930 行,struct us_data *us,于是,日后我们会非常频繁的看到 us 的.另,us 什么意思?尼采说:us 者,usb storage 是也.

937 行,就是为 us 申请内存空间,而 938 行就是判断内存空间是否申请成功,成功的话 us 就不会为 0,或者说为 NULL,如果为 NULL 那么就是失败了,那么别再浪费表情了,整部戏就这么提前夭

折了.在这里需要强调的是,整个内核代码中,像这种判断内存申请是否成功的语句是无处不在,每次有内存申请的语句,其后边一定会跟随一句判断申请成功与否的语句.写过代码的人都该知道,这样做是很有必要的,因为你没有申请到内存,那么继续下去就是没有意义的,除了可能让人见识计算机是如何崩溃之外,没有别的好处.而内存申请不管申请了多大,都有可能失败,写代码的人这样做无非是想告诫我们,我们的计算机并不总像人民币那般坚挺,她很脆弱.当你真正用计算机写代码的时候你就会发现计算机多么的脆弱和无力。

942 行,给 us 初始化为全 0.

934 行这个 US_DEBUGP,是一个宏,来自 drivers/usb/storage/debug.h,接下来很多代码中我们也会看到这个宏,她无非就是打印一些调试信息.debug.h 中有这么一段,

```
54 #ifdef CONFIG_USB_STORAGE_DEBUG
55 void usb_stor_show_command(struct scsi_cmnd *srb);
56 void usb_stor_show_sense( unsigned char key,
57               unsigned char asc, unsigned char ascq );
58 #define US_DEBUGP(x...) printk( KERN_DEBUG USB_STORAGE x )
59 #define US_DEBUGPX(x...) printk( x )
60 #define US_DEBUG(x) x
61 #else
62 #define US_DEBUGP(x...)
63 #define US_DEBUGPX(x...)
64 #define US_DEBUG(x)
65 #endif
66
67 #endif
```

这里一共定义了几个宏,US_DEBUGP,US_DEBUGPX,US_DEBUG,差别不大,只是形式上略有不同罢了.

需要注意的是,这些调试信息得是我们打开了编译选项 CONFIG_USB_STORAGE_DEBUG 才有意义的,这里也看出来了,如果这个选项为 0,那么这几个宏就什么也不干,因为它们被赋为空了.关于 US_DEBUG 系列的这几个宏,就讲这么多,之后再碰上,将直接过滤掉,不予多说.

关于 prink 和 kmalloc,这两个函数也没有太多需要说的,对大多数人来讲,就把 printk 当成 printf,把 kmalloc 当成 malloc 即可,只不过是这两个函数是专门用于内核代码中的.一个是打印一些东西,一个是申请内存空间.

931 行呢?id_index 干嘛用的?让我们在下节不见不散吧.Be there or be square!-孙楠如是说.

设备花名册

storage_probe 这个函数挺有意思的,总长度不足 100 行,但是干了许多事情,这就像足球场上的后腰,比如切尔西的马克莱莱,在场上并不起眼,但是却为整个团队做出了卓越的贡献.也有很多评论家说银河战舰皇家马德里这几年的衰落正是从赶走这个不起眼的马克莱莱开始的.

在讲 id_index 之前,我们继续贴 storage_probe 的代码:

```
943         init_MUTEX(&(us->dev_semaphore));
944         init_MUTEX_LOCKED(&(us->sema));
945         init_completion(&(us->notify));
946         init_waitqueue_head(&us->dev_reset_wait);
947         init_waitqueue_head(&us->scsi_scan_wait);
948         init_completion(&us->scsi_scan_done);
949
950         /* Associate the us_data structure with the USB device */
951         result = associate_dev(us, intf);
952         if (result)
953                 goto BadDevice;
954
955         /*
956          * Get the unusual_devs entries and the descriptors
957          *
958          * id_index is calculated in the declaration to be the index number
959          * of the match from the usb_device_id table, so we can find the
960          * corresponding entry in the private table.
961          */
962         get_device_info(us, id_index);
```

storage_probe 这个函数之所以短小,是因为它调用了大量的函数.所以,看起来短短一段代码,实际上却要花费我们读代码的人好几个小时,想想真是觉得不划算,难怪朱自清先生看到这个 storage_probe 函数的时候,不禁感慨,燕子去了,有再来的时候,杨柳枯了,有再青的时候,桃花谢了,有再开的时候,但是,聪明的你告诉我,我们读这段不足 100 行的函数花掉的时间为何一去不复返呢?

其实我们不知道 id_index 也不影响对后面问题的理解,甚至某种意义上来说,花太多笔墨去讲这个 id_index 有一点喧宾夺主的感觉,但是,有时候,有些事情,你明知它是无奈的事,无奈的心情,却还要无奈的面对,无奈的去选择,有时想无奈的逃避都不可能,因为我们都被无奈禁锢了.比如这里,注意到 962 行出现了一个 get_device_info 的函数,它的一个参数就是 id_index,所以,我们别无选择,只能看看这个 id_index 究竟是干嘛的.

上节我们注意到 id_index=id-storage_usb_ids,id 我们知道,storage_probe 函数的两个形参之一,而 storage_usb_ids,不是别人,正是我们曾经赋给 usb_storage_driver 的成员 id_table 的值.忘记了 id_table 的可以回去看.它实际上就是一张表格,告诉全世界我这个 driver 支持怎样的一些设备.storage_usb_ids 同样来自 drivers/usb/storage/usb.c 中,

```
111 /* The entries in this table, except for final ones here
112  * (USB_MASS_STORAGE_CLASS and the empty entry), correspond,
113  * line for line with the entries of us_unsuaul_dev_list[].
114  */
115
116 #define UNUSUAL_DEV(id_vendor, id_product, bcdDeviceMin, bcdDeviceMax, \
117                 vendorName, productName,useProtocol, useTransport, \
118                 initFunction, flags) \
```

```
119 { USB_DEVICE_VER(id_vendor, id_product, bcdDeviceMin,bcdDeviceMax) }
120
121 static struct usb_device_id storage_usb_ids [] = {
122
123 #       include "unusual_devs.h"
124 #undef UNUSUAL_DEV
125         /* Control/Bulk transport for all SubClass values */
126         { USB_INTERFACE_INFO(USB_CLASS_MASS_STORAGE, US_SC_RBC,
US_PR_CB) },
127         { USB_INTERFACE_INFO(USB_CLASS_MASS_STORAGE, US_SC_8020,
US_PR_CB) },
128         { USB_INTERFACE_INFO(USB_CLASS_MASS_STORAGE, US_SC_QIC,
US_PR_CB) },
129         { USB_INTERFACE_INFO(USB_CLASS_MASS_STORAGE, US_SC_UFI,
US_PR_CB) },
130         { USB_INTERFACE_INFO(USB_CLASS_MASS_STORAGE, US_SC_8070,
US_PR_CB) },
131         { USB_INTERFACE_INFO(USB_CLASS_MASS_STORAGE, US_SC_SCSI,
US_PR_CB) },
132
133         /* Control/Bulk/Interrupt transport for all SubClass values */
134         { USB_INTERFACE_INFO(USB_CLASS_MASS_STORAGE, US_SC_RBC,
US_PR_CBI) },
135         { USB_INTERFACE_INFO(USB_CLASS_MASS_STORAGE, US_SC_8020,
US_PR_CBI) },
136         { USB_INTERFACE_INFO(USB_CLASS_MASS_STORAGE, US_SC_QIC,
US_PR_CBI) },
137         { USB_INTERFACE_INFO(USB_CLASS_MASS_STORAGE, US_SC_UFI,
US_PR_CBI) },
138         { USB_INTERFACE_INFO(USB_CLASS_MASS_STORAGE, US_SC_8070,
US_PR_CBI) },
139         { USB_INTERFACE_INFO(USB_CLASS_MASS_STORAGE, US_SC_SCSI,
US_PR_CBI) },
140
141         /* Bulk-only transport for all SubClass values */
142         { USB_INTERFACE_INFO(USB_CLASS_MASS_STORAGE, US_SC_RBC,
US_PR_BULK) },
143         { USB_INTERFACE_INFO(USB_CLASS_MASS_STORAGE, US_SC_8020,
US_PR_BULK) },
144         { USB_INTERFACE_INFO(USB_CLASS_MASS_STORAGE, US_SC_QIC,
US_PR_BULK) },
145         { USB_INTERFACE_INFO(USB_CLASS_MASS_STORAGE, US_SC_UFI,
US_PR_BULK) },
146         { USB_INTERFACE_INFO(USB_CLASS_MASS_STORAGE, US_SC_8070,
```

```
US_PR_BULK) },
   147 #if !defined(CONFIG_BLK_DEV_UB) && !defined(CONFIG_BLK_DEV_UB_MODULE)
   148         { USB_INTERFACE_INFO(USB_CLASS_MASS_STORAGE, US_SC_SCSI,
US_PR_BULK) },
   149 #endif
   150
   151         /* Terminating entry */
   152         { }
   153 };
```

注意到这是一个 struct usb_device_id 结构体的数组,所以即使我们用下半身思考也能知道,其中每一项必然是一个 struct usb_device_id 的结构体变量.我们先来看 USB_INTERFACE_INFO 这个咚咚,很显然这是一个宏,来自 include/linux/usb.h,

```
   473 /**
   474  * USB_INTERFACE_INFO - macro used to describe a class of usb interfaces
   475  * @cl: bInterfaceClass value
   476  * @sc: bInterfaceSubClass value
   477  * @pr: bInterfaceProtocol value
   478  *
   479  * This macro is used to create a struct usb_device_id that matches a
   480  * specific class of interfaces.
   481  */
   482 #define USB_INTERFACE_INFO(cl,sc,pr) \
   483         .match_flags = USB_DEVICE_ID_MATCH_INT_INFO, .bInterfaceClass =
(cl), .bInterfaceSubClass = (sc), .bInterfaceProtocol = (pr)
```

每一个 USB_INTERFACE_INFO 就是构造一个 struct usb_device_id 的结构体变量,回顾一下我们之前给出的 struct usb_device_id 的定义,这里实际上就是为其中的四个元素赋了值,它们是 match_flags,bInterfaceClass,bInterfaceSubClass,bInterfaceProtocol.这里不得不说的是,这个世界上有许许多多的 usb 设备,它们各有各的特点,为了区分它们,usb 规范,或者说 usb 协议,把 usb 设备分成了很多类,然而每个类又分成子类,这很好理解,我们一个大学也是如此,先是分成很多个学院,比如我们复旦大学,信息学院,经济学院,管理学院,外文学院,等等.然后每个学院又被分为很多个系,比如信息学院,下面分了电子工程系,微电子系,计算机系,通信工程系,然后可能每个系下边又分了各个专业,usb 协议也是这样干的,首先每个 Interface 属于一个 Class,(为什么是把 Interface 分类,而不把 Device 分类?前面讲过了,在 usb 设备驱动中,不用再提 Device,因为每个设备驱动对应的是一种 Interface,而不是一种 Device),然后 Class 下面又分了 SubClass,完了 SubClass 下面又按各种设备所遵循的不同的通信协议继续细分.usb 协议里边为每一种 Class,每一种 SubClass,每一种 Protocol 定义一个数值,比如 mass storage 的 Class 就是 0x08,而这里 USB_CLASS_MASS_STORAGE 这个宏在 include/linux/usb_ch9.h 中定义,其值正是 8.

我们拿第 126 行来举例,

```
  { USB_INTERFACE_INFO(USB_CLASS_MASS_STORAGE, US_SC_RBC, US_PR_CB) },
```

把这个宏展开,就是说定义了这么一个 usb_device_id 结构体变量,其 match_flags=USB_DEVICE_ID_MATCH_INT_INFO,而

bInterfaceClass=USB_CLASS_MASS_STORAGE,bInterfaceSubClass=US_SC_RBC,而 bInterfaceProtocol=US_PR_CB.

USB_CLASS_MASS_STORAGE 就不用再说了,咱们这个驱动程序所支持的每一种设备都是属于这个类,或者说这个 Class.但是这个 Class 里边包含不同的 SubClass,比如 subclass 02 为 CD-ROM 设备,04 为软盘驱动器,06 为通用 SCSI 类设备.而通信协议则主要有 CBI 协议和 Bulk-Only 协议.

像 US_SC_RBC 这些关于 sub class 的宏的定义是在文件 drivers/usb/storage/protocol.h 中:

```
47 /* Sub Classes */
48
49 #define US_SC_RBC       0x01          /* Typically, flash devices */
50 #define US_SC_8020      0x02          /* CD-ROM */
51 #define US_SC_QIC       0x03          /* QIC-157 Tapes */
52 #define US_SC_UFI       0x04          /* Floppy */
53 #define US_SC_8070      0x05          /* Removable media */
54 #define US_SC_SCSI      0x06          /* Transparent */
55 #define US_SC_ISD200    0x07          /* ISD200 ATA */
```

而像 US_PR_CB 这些关于传输协议的宏则在另一个文件中,drivers/usb/storage/transport.h

```
/* Protocols */

#define US_PR_CBI       0x00          /* Control/Bulk/Interrupt */
#define US_PR_CB        0x01          /* Control/Bulk w/o interrupt */
#define US_PR_BULK      0x50          /* bulk only */
```

这个文件中还定义了更多的协议,不过我们暂时只需要知道这三种,因为其她协议都是专门针对一些特殊设备的,在 storage_usb_ids 数组中使用宏 USB_INTERFACE_INFO 定义的 usb_device_id 都只是用的这三种协议.(US_PR_CBI 和 US_PR_CB 这两种协议在 usb 协议中都唤作 CBI,不过之间有点差别.)而对于一些特殊的设备,则还在 unusual_devs.h 文件中有专门的一些变量定义,我们暂且不去关注它们.

说了这许多,U 盘属于其中的哪一种呢?usb 协议中规定,U 盘的 Subclass 是属于 US_SC_SCSI 的.而其通信协议使用的是 Bulk-Only 的.显然这些东西我们后来都会用得上.

那么这里还有一个 match_flag,它又是表示什么意思?USB_INTERFACE_INFO 这个宏貌似把所有的设备的 match_flag 都给设成了 USB_DEVICE_ID_MATCH_INT_INFO,这是为啥?这个宏来自 include/linux/usb.h,

```
435 #define USB_DEVICE_ID_MATCH_INT_INFO \
436         (USB_DEVICE_ID_MATCH_INT_CLASS |
USB_DEVICE_ID_MATCH_INT_SUBCLASS | USB_DEVICE_ID_MATCH_INT_PROTOCOL)
```

match_flag 这个咚咚是给 usb core 去用的,usb core 负责给设备寻找适合她的 driver,负责给 driver 寻找适合他的 device,它所比较的就是 struct usb_device_id 的变量,而 struct usb_device_id 结构体中有许多成员,那么是不是一定要把每一个成员都给比较一下呢,其实没有必要那么细,差不多就行了,比如咱们这里,就是告诉 usb core,你只要比较 bInterfaceClass,bInterfaceSubClass,bInterfaceProtocol 即可.include/linux/mod_devicetable.h 中针对 struct usb_device_id 中的每一个要比较的项定义了一个宏:

```
121 /* Some useful macros to use to create struct usb_device_id */
122 #define USB_DEVICE_ID_MATCH_VENDOR              0x0001
123 #define USB_DEVICE_ID_MATCH_PRODUCT             0x0002
124 #define USB_DEVICE_ID_MATCH_DEV_LO              0x0004
125 #define USB_DEVICE_ID_MATCH_DEV_HI              0x0008
126 #define USB_DEVICE_ID_MATCH_DEV_CLASS           0x0010
127 #define USB_DEVICE_ID_MATCH_DEV_SUBCLASS        0x0020
128 #define USB_DEVICE_ID_MATCH_DEV_PROTOCOL        0x0040
129 #define USB_DEVICE_ID_MATCH_INT_CLASS           0x0080
130 #define USB_DEVICE_ID_MATCH_INT_SUBCLASS        0x0100
131 #define USB_DEVICE_ID_MATCH_INT_PROTOCOL        0x0200
```

回去对比一下 struct usb_device_id 就知道这些宏是什么意思了.

然后我们再看 storage_usb_ids 中那个#include "unusual_devs.h",实际上这个文件也是在我们这个 drivers/usb/storage/目录下面,它里边定义了一些特殊的设备,也是以 struct usb_device_id 结构体变量的形式,这些设备或者是有一些别的设备不具备的特性,或者是他们遵循的通信协议有些与众不同,比如,它既不是 Bulk-Only 也不是 CBI,像这些不按常理出牌的设备,写代码的同志们把他们单独给列了出来.整理在这个文件中.当然,从大的类来分,它们依然是属于 usb mass storage 这个类别的,否则也没必要放在这个目录下面了.

至此,我们可以知道 storage_usb_ids 这个表是怎么回事了.usb core 为每个设备在这张表里查找,如果找到了某一行和这个设备相匹配,那么该行就是我们前面提到的那个 storage_probe()的参数 id.所以 id_index=id-storage_usb_ids 就是如其字面意义那样,在表中的编号.至于这个编号有什么用,那我们骑驴看唱本.总之,费这么大劲干了这么一件事,总是有它的意义的.

最后,总结陈词,这个所谓的花名册,就好比我们大学生申请国外学校,每个人都会事先搜集一大批学校名单,然后结合各方面,比如师资力量,是否牛的导师够多,比如经济实力,是否能给够多的奖学金,比如教授本人资历,是否获得过重大奖项,取得过何种成绩,比如学校声望, 是否是名校,综合这些来看最终确定一份名单,就是自己真正心仪的学校花名册.那么那个 match_flags 是什么意思呢,而有的同学嫌这样太麻烦,又或者他们可能只是需要一个能发学历的,像杨澜老公吴征所获得博士学位的那个巴林顿大学那样卖假文凭的学校也许就能满足他们了,那么他们就不必去评估那么多项,反正他们心中的 match_flags 就是能够和吴征成为校友就可以了,管它是否因为该学校被取缔以后他们也会像吴征那样痛失母校呢.

冰冻三尺非一日之寒

罗马不是一天建成的.在让 U 盘工作之前,其实我们的驱动作了很多准备工作.

我们继续跟着感觉走,storage_probe(),943 行至 948 行,一系列的以 init_*命名的函数在此刻被调用,这里涉及了一些锁机制,等待机制,不过只是初始化,暂且不理睬,到后面用到的时候再细说,不过请记住,这几行每一行都是有用的.后面自然会用得着.

此时,我们先往下走,951 行 associate_dev()和 962 行 get_device_info(),这两个函数是我们目前需要看的,一个一个来.

先看 associate_dev(),定义于 drivers/usb/storage/usb.c,

```
429 /* Associate our private data with the USB device */
430 static int associate_dev(struct us_data *us, struct usb_interface *intf)
431 {
432         US_DEBUGP("-- %s\n", __FUNCTION__);
433
434         /* Fill in the device-related fields */
435         us->pusb_dev = interface_to_usbdev(intf);
436         us->pusb_intf = intf;
437         us->ifnum = intf->cur_altsetting->desc.bInterfaceNumber;
438         US_DEBUGP("Vendor: 0x%04x, Product: 0x%04x, Revision: 0x%04x\n",
439                         us->pusb_dev->descriptor.idVendor,
440                         us->pusb_dev->descriptor.idProduct,
441                         us->pusb_dev->descriptor.bcdDevice);
442         US_DEBUGP("Interface Subclass: 0x%02x, Protocol: 0x%02x\n",
443                         intf->cur_altsetting->desc.bInterfaceSubClass,
444                         intf->cur_altsetting->desc.bInterfaceProtocol);
445
446         /* Store our private data in the interface */
447         usb_set_intfdata(intf, us);
448
449         /* Allocate the device-related DMA-mapped buffers */
450         us->cr = usb_buffer_alloc(us->pusb_dev, sizeof(*us->cr),
451                         GFP_KERNEL, &us->cr_dma);
452         if (!us->cr) {
453                 US_DEBUGP("usb_ctrlrequest allocation failed\n");
454                 return -ENOMEM;
455         }
456
457         us->iobuf = usb_buffer_alloc(us->pusb_dev, US_IOBUF_SIZE,
458                         GFP_KERNEL, &us->iobuf_dma);
459         if (!us->iobuf) {
460                 US_DEBUGP("I/O buffer allocation failed\n");
461                 return -ENOMEM;
462         }
463         return 0;
464 }
```

我们首先来关注函数 associate_dev 的参数, struct us_data *us,传递给它的是 us,这个不用多说了吧,此前刚刚为它申请了内存,并且初始化各成员为 0. 这个 us 将一直陪伴我们走下去,直到我们的故事结束.所以其重要性不言而喻. struct usb_interface *intf,这个也不用说,storage_probe()函数传进来的两个参数之一.总之,此处郑重申明一次,struct us_data 的结构体指针 us,struct usb_interface 结构体的指针 intf,以及 struct usb_device 结构体和 struct usb_device_id 结构体在整个 U 盘驱动的故事中是唯一的,每次提到都是那个. 而以后我们会遇上的几个重要的数据结构,struct urb urb,struct scsi_cmnd

srb 这也非常重要,但是它们并不唯一,也许每次遇上都不一样,就像演戏一样.前边这几个数据结构的变量就像那些主角,而之后遇见的 urb 啊,srb 啊,虽然频繁露面,但是只是群众演员,只不过这次是路人甲,下次是路人乙. 所以,以后我们将只说 us,不再说 struct us_data *us,struct usb_interface * intf 也将只用 intf 来代替.

us 之所以重要,是因为接下来很多函数都要用到它以及它的各个成员.实际上目前这个函数,associate_dev 所做的事情就是为 us 的各成员赋值,毕竟此刻 us 和我们之前提到的那些 struct usb_device 啊,struct usb_interface 啊,还没有一点关系.因而,这个函数,以及这之后的好几个函数都是为了给 us 的各成员赋上适当的值,之所以如此兴师动众去为它赋值,主要就是因为后面要利用它.所谓天下没有免费的午餐.

432 行,本来无须多讲,因为只是一个 debug 语句,不过提一下__FUNCTION__这个"宏",gcc 2.95 以后的版本支持这么一个冬冬,这个"宏"在编译的时候会被转换为函数名(字符串),这里自然就是"associate_dev"这么一个字符串,于是函数执行到这里就会打印一句话告诉世人我们执行到这个函数来了,这种做法显然会有利于咱们调试程序.不过这个冬冬实际上不是宏,因为预处理器对她一无所知.她的心只有编译器才懂.

435 行,pusb_dev,就是 point of usb device 的意思.struct us_data 中的一个成员,按照我们刚才约定的规矩,此刻我将说 us 的一个成员,us->pusb_dev= interface_to_usbdev(intf), interface_to_usbdev 我们前面已经讲过,其含义正如字面表示的那样,把一个 struct interface 结构体的指针转换成一个 struct usb_device 的结构体指针.前面我们说过,struct usb_device 对我们没有什么用,但是 usb core 层的一些函数要求使用这个参数,所以我们不得已而为止,正所谓人在江湖身不由己.

436 行,把 intf 赋给 us 的 pusb_intf.

437 行,us 的 ifnum, 先看 intf 的 cur_altsetting,这个容易令外行混淆.usb 设备有一个 configuration 的概念,这个我们前面讲协议的时候说了,而这里又有一个 setting,咋一看有些奇怪,这两个词不是一回事吗.这时候,就体现出外语水平了,上过新东方没上过新东方,背没背过俞敏洪的 GRE 红宝书,在这时候就体现出差距了.还是拿我们最熟悉的手机来打比方,configuration 不说了,setting,一个手机可能各种配置都确定了,是振动还是铃声已经确定了,各种功能都确定了,但是声音的大小还可以变吧,通常手机的音量是一格一格的变动,大概也就 5,6 格,那么这个可以算一个 setting 吧.这里 cur_altsetting 就是表示的当前的这个 setting,或者说设置.cur_altsetting 是一个 struct usb_host_interface 的指针,这个结构体定义于 include/linux/usb.h:

```
51 /* host-side wrapper for one interface setting's parsed descriptors */
52 struct usb_host_interface {
53         struct usb_interface_descriptor desc;
54
55         /* array of desc.bNumEndpoint endpoints associated with this
56          * interface setting.  these will be in no particular order.
57          */
58         struct usb_host_endpoint *endpoint;
59
60         unsigned char *extra;   /* Extra descriptors */
61         int extralen;
```

```
62 };
```
它的成员 desc 是一个 struct usb_interface_descriptor 结构体变量,这个结构体的定义是和 usb 协议直接对应的,定义于 include/linux/usb_ch9.h.(这里取名为"ch9"是因为这个文件很多东西对应于 usb spec 2.0 中的第九章,chapter 9.):

```
242 /* USB_DT_INTERFACE: Interface descriptor */
243 struct usb_interface_descriptor {
244       __u8  bLength;
245       __u8  bDescriptorType;
246
247       __u8  bInterfaceNumber;
248       __u8  bAlternateSetting;
249       __u8  bNumEndpoints;
250       __u8  bInterfaceClass;
251       __u8  bInterfaceSubClass;
252       __u8  bInterfaceProtocol;
253       __u8  iInterface;
254 } __attribute__ ((packed));
```
而其中我们这里提到的是 bInterfaceNumber,一个设备可以有多个 Interface,于是每一个 Interface 当然就得用个编号了,要不然咋区分啊?所有这些描述符里的冬冬都是出厂的时候就固化在设备里边的,而我们这里之所以可以用 bInterfaceNumber 来赋值,是因为 usbcore 在为设备初始化的时候就已经做足了这些功课.否则的话,我们真是寸步难行.

总之,us->ifnum 就是这样,最终就是等于咱们眼下这个 interface 的编号.

438 到 444 行就是两句调试语句,打印更多一些描述符信息,包括 device 描述符和 interface 描述符.

447 行, usb_set_intfdata(),这其实是一个内联函数,她就一行代码,也是定义于 include/linux/usb.h 中:

```
138 static inline void usb_set_intfdata (struct usb_interface *intf, void *data)
139 {
140       dev_set_drvdata(&intf->dev, data);
141 }
```
有趣的是,dev_set_drvdata 这个函数也是内联函数,也只有一行代码,她定义于 include/linux/device.h 中:

```
302 static inline void
303 dev_set_drvdata (struct device *dev, void *data)
304 {
305       dev->driver_data = data;
306 }
```
所以,结合来看,最终做的事情就是让&intf->dev->driver_data=data,即 &intf->dev->driver_data=us.

再往下走,就是申请内存了,us->cr 和 us->iobuf 都是指针,这里让它们指向两段内存空间,下面会用得到.需要注意的是 usb_buffer_alloc(),这个函数是 usbcore 提供的,我们只管调用即可.从名字上就能知道它是用来申请内存的,第一个参数就是 struct usb_device 结构体的指针,所以我们要传递一个 pusb_dev,第三个参数,GFP_KERNEL,是一个内存申请的 flag,通常内存申请都用这个 flag,除非是中断上下文,不能睡眠,那就得用 GPF_ATOMIC,这里没那么多要求.第二个参数申请的 buffer 的大小,对于 cr,传递的是 sizeof(*us->cr),而对于 iobuf,传递的是 US_IOBUF_SIZE,这是一个宏,大小是 64,是我们自己定义的,来自 drivers/usb/storage/usb.h:

```
91 /*
92  * We provide a DMA-mapped I/O buffer for use with small USB transfers.
93  * It turns out that CB[I] needs a 12-byte buffer and Bulk-only needs a
94  * 31-byte buffer.  But Freecom needs a 64-byte buffer, so that's the
95  * size we'll allocate.
96  */
97
98 #define US_IOBUF_SIZE           64      /* Size of the DMA-mapped I/O buffer
*/
```

而 usb_buffer_alloc()的第四个参数有些意思了,第一次我们传递的是&us->cr_dma,第二次传递的是&us->iobuf_dma,这涉及到 dma 传输.这两个咚咚此前我们都没有赋过值,相反它们是在这个函数调用之后被赋上值的.cr_dma 和 iobuf_dma 都是 dma_addr_t 类型的变量,这个数据类型是 Linux 内核中专门为 dma 传输而准备的.为了支持 dma 传输,usb_buffer_alloc 不仅仅是申请了地址,并且建立了 dma 映射,cr_dma 和 iobuf_dma 就是记录着 cr 和 iobuf 的 dma 地址.关于什么是 cr,关于这些 dma 地址究竟有什么用,我们稍候就会遇到,那时候再讲也不迟.现在需要知道的就是 usb_buffer_alloc 申请的空间分别返回给了 cr 和 iobuf.顺便提一下,用 usb_buffer_alloc 申请的内存空间需要用它的搭档 usb_buffer_free()来释放.

452 行和 459 行,每一次申请完内存就要检查成功与否,这是惯例.驱动程序能否驱动设备,关键就是看能否申请到内存空间,任何一处内存空间申请失败,整个驱动程序就没法正常工作.这就像如今找对象,谈婚姻,总是要看有没有房子.没有房子的话,那么基本上爱情也就没戏.然而现实中要拥有房子比计算机里分配内存却要难上许多,许多.可怜的我们这一代人,当我们不能挣钱的时候,房子是分配的,当我们能挣钱的时候,却发现房子已经买不起了.哭...

冬天来了,春天还会远吗?(一)

整个 usb-storage 模块的代码中,其最灵魂的部分在一个叫做 usb_stor_control_thread()的函数中,而那也自然是我们整个故事的高潮.这个函数的调用有些特殊,我们是从 usb_stor_acquire_resources()函数进入的,而后者我们即将遇到,它在整部戏中只出现过一次,即 storage_probe 中,行号为 998 的地方.然而在此之前,有四个函数挡在我们面前,它们就是

get_device_info,get_transport,get_protocol,get_pipes.如我前面所说,两个人要走到一起,首先要了解彼此,这四个函数就是让 driver 去认识 device 的.这一点我们从名字上也能看出来.driver 需要知道 device 姓甚名谁,所以有了 get_device_info,driver 需要知道 device 喜欢用什么方式沟通,是用 QQ 还是用 msn 还是只用手机短信,如果是某一种,那么账号是多少,或者手机号是多少,写代码的人把这些工作分配给了 get_transport,get_protocol,get_pipes.

实际上,这四个函数,加上之前刚说过的那个 associate_dev(),是整个故事中最平淡最枯燥的部分,第一次读这部分代码总让人困惑,怎么没看见一点 usb 数据通信啊?完全没有看到 usb host 和 usb device 是如何在交流的,这是 usb 吗?这一刻,这颗浮躁的心,在这个城市,迷失了.但是,我们知道,爱情,并非都与风花雪月有关,友情,并非总与疯斗打闹有关.这几个函数应该说是给后面做铺垫,红花总要有绿叶配,没有这段代码的铺垫,到了后面 usb 设备恐怕也无法正常工作吧.不过,一个利好消息是,这几个函数我们只会遇见这一次,它们在整个故事中就这么一次露脸的机会,像我们每个人的青春,只有一次,无法回头.和我们每个人的青春一样,都是绝版的.所以,让我们享受这段平淡无奇的代码吧.

get_device_info, 这个函数定义于 drivers/usb/storage/usb.c 中:

```
466 /* Get the unusual_devs entries and the string descriptors */
467 static void get_device_info(struct us_data *us, int id_index)
468 {
469         struct usb_device *dev = us->pusb_dev;
470         struct usb_interface_descriptor *idesc =
471                 &us->pusb_intf->cur_altsetting->desc;
472         struct us_unusual_dev *unusual_dev = &us_unusual_dev_list[id_index];
473         struct usb_device_id *id = &storage_usb_ids[id_index];
474
475         /* Store the entries */
476         us->unusual_dev = unusual_dev;
477         us->subclass = (unusual_dev->useProtocol == US_SC_DEVICE) ?
478                         idesc->bInterfaceSubClass :
479                         unusual_dev->useProtocol;
480         us->protocol = (unusual_dev->useTransport == US_PR_DEVICE) ?
481                         idesc->bInterfaceProtocol :
482                         unusual_dev->useTransport;
483         us->flags = unusual_dev->flags;
484
485         /* Log a message if a non-generic unusual_dev entry contains an
486          * unnecessary subclass or protocol override.  This may stimulate
487          * reports from users that will help us remove unneeded entries
488          * from the unusual_devs.h table.
489          */
490         if (id->idVendor || id->idProduct) {
491                 static char *msgs[3] = {
492                         "an unneeded SubClass entry",
493                         "an unneeded Protocol entry",
494                         "unneeded SubClass and Protocol entries"};
```

```
495             struct usb_device_descriptor *ddesc = &dev->descriptor;
496             int msg = -1;
497
498             if (unusual_dev->useProtocol != US_SC_DEVICE &&
499                     us->subclass == idesc->bInterfaceSubClass)
500                     msg += 1;
501             if (unusual_dev->useTransport != US_PR_DEVICE &&
502                     us->protocol == idesc->bInterfaceProtocol)
503                     msg += 2;
504             if (msg >= 0 && !(unusual_dev->flags & US_FL_NEED_OVERRIDE))
505                     printk(KERN_NOTICE USB_STORAGE "This device "
506                             "(%04x,%04x,%04x S %02x P %02x)"
507                             " has %s in unusual_devs.h\n"
508                             "   Please send a copy of this message to "
509                             "<linux-usb-devel@lists.sourceforge.net>\n",
510                             ddesc->idVendor, ddesc->idProduct,
511                             ddesc->bcdDevice,
512                             idesc->bInterfaceSubClass,
513                             idesc->bInterfaceProtocol,
514                             msgs[msg]);
515     }
516
517     /* Read the device's string descriptors */
518     if (dev->descriptor.iManufacturer)
519             usb_string(dev, dev->descriptor.iManufacturer,
520                     us->vendor, sizeof(us->vendor));
521     if (dev->descriptor.iProduct)
522             usb_string(dev, dev->descriptor.iProduct,
523                     us->product, sizeof(us->product));
524     if (dev->descriptor.iSerialNumber)
525             usb_string(dev, dev->descriptor.iSerialNumber,
526                     us->serial, sizeof(us->serial));
527
528     /* Use the unusual_dev strings if the device didn't provide them */
529     if (strlen(us->vendor) == 0) {
530             if (unusual_dev->vendorName)
531                     strlcpy(us->vendor, unusual_dev->vendorName,
532                             sizeof(us->vendor));
533             else
534                     strcpy(us->vendor, "Unknown");
535     }
536     if (strlen(us->product) == 0) {
537             if (unusual_dev->productName)
538                     strlcpy(us->product, unusual_dev->productName,
```

```
539                         sizeof(us->product));
540             else
541                     strcpy(us->product, "Unknown");
542         }
543         if (strlen(us->serial) == 0)
544             strcpy(us->serial, "None");
545
546         US_DEBUGP("Vendor: %s,  Product: %s\n", us->vendor, us->product);
547 }
```

469 行,不多说,dev 还是那个 dev.

470 行,struct usb_interface_descriptor *idesc,这个也无须再说,之前那个 associate_dev 函数里已经介绍过这个结构体,而且整个故事就是针对一个 interface 的,一个 interface 就对应一个 interface 描述符.所以不管在哪里看到,她总还是她.

472 行, struct us_unusual_dev,这个结构体是第一次出现,她定义于 drivers/usb/storage/usb.h 中,

```
55 /*
56  * Unusual device list definitions
57  */
58
59 struct us_unusual_dev {
60         const char* vendorName;
61         const char* productName;
62         __u8  useProtocol;
63         __u8  useTransport;
64         int (*initFunction)(struct us_data *);
65         unsigned int flags;
66 };
```

而等号右边的 us_unusal_dev_list 是一个数组,定义于 drivers/usb/storage/usb.c:

```
157 /* This is the list of devices we recognize, along with their flag data */
158
159 /* The vendor name should be kept at eight characters or less, and
160  * the product name should be kept at 16 characters or less. If a device
161  * has the US_FL_FIX_INQUIRY flag, then the vendor and product names
162  * normally generated by a device thorugh the INQUIRY response will be
163  * taken from this list, and this is the reason for the above size
164  * restriction. However, if the flag is not present, then you
165  * are free to use as many characters as you like.
166  */
167
168 #undef UNUSUAL_DEV
169 #define UNUSUAL_DEV(idVendor, idProduct, bcdDeviceMin, bcdDeviceMax, \
```

```
170                     vendor_name, product_name, use_protocol, use_transport,
\
171                     init_function, Flags) \
172 { \
173         .vendorName = vendor_name,      \
174         .productName = product_name,    \
175         .useProtocol = use_protocol,    \
176         .useTransport = use_transport,  \
177         .initFunction = init_function,  \
178         .flags = Flags, \
179 }
180
181 static struct us_unusual_dev us_unusual_dev_list[] = {
182 #       include "unusual_devs.h"
183 #       undef UNUSUAL_DEV
184         /* Control/Bulk transport for all SubClass values */
185         { .useProtocol = US_SC_RBC,
186           .useTransport = US_PR_CB},
187         { .useProtocol = US_SC_8020,
188           .useTransport = US_PR_CB},
189         { .useProtocol = US_SC_QIC,
190           .useTransport = US_PR_CB},
191         { .useProtocol = US_SC_UFI,
192           .useTransport = US_PR_CB},
193         { .useProtocol = US_SC_8070,
194           .useTransport = US_PR_CB},
195         { .useProtocol = US_SC_SCSI,
196           .useTransport = US_PR_CB},
197
198         /* Control/Bulk/Interrupt transport for all SubClass values */
199         { .useProtocol = US_SC_RBC,
200           .useTransport = US_PR_CBI},
201         { .useProtocol = US_SC_8020,
202           .useTransport = US_PR_CBI},
203         { .useProtocol = US_SC_QIC,
204           .useTransport = US_PR_CBI},
205         { .useProtocol = US_SC_UFI,
206           .useTransport = US_PR_CBI},
207         { .useProtocol = US_SC_8070,
208           .useTransport = US_PR_CBI},
209         { .useProtocol = US_SC_SCSI,
210           .useTransport = US_PR_CBI},
211
212         /* Bulk-only transport for all SubClass values */
```

```
213         { .useProtocol = US_SC_RBC,
214           .useTransport = US_PR_BULK},
215         { .useProtocol = US_SC_8020,
216           .useTransport = US_PR_BULK},
217         { .useProtocol = US_SC_QIC,
218           .useTransport = US_PR_BULK},
219         { .useProtocol = US_SC_UFI,
220           .useTransport = US_PR_BULK},
221         { .useProtocol = US_SC_8070,
222           .useTransport = US_PR_BULK},
223 #if !defined(CONFIG_BLK_DEV_UB)
&& !defined(CONFIG_BLK_DEV_UB_MODULE)
224         { .useProtocol = US_SC_SCSI,
225           .useTransport = US_PR_BULK},
226 #endif
227
228         /* Terminating entry */
229         { NULL }
230 };
```

无可奈何花落去,似曾相识燕归来.这个数组看上去特别面熟对不对?可曾记得当初我们见过的那个 storage_usb_ids,仔细对比一下会发现,这两个数组的元素个数完全一样,只不过,一个由是 struct usb_device_id 结构体构成的,另一个则是 struct us_unusual_dev 结构体构成的,其实这两个表格是一一对应的,它们也都定义于同一个文件中. 细心一点会注意到,UNUSUAL_DEV 这个宏在这个文件里被定义了两次,这是干嘛用的?听仔细了,我们曾经提过 unusual_devs.h 这个文件,这个文件的作用是什么?假设没有这个文件,那么 storage_usb_ids 这张表就是用 USB_INTERFACE_INFO 这个宏定义了几种常规的 usb mass storage 的设备,比如,

```
{ USB_INTERFACE_INFO(USB_CLASS_MASS_STORAGE, US_SC_RBC, US_PR_CB) },
```

这一行就是定义一个设备的 bInterfaceClass,bInterfaceSubClass,bInterfaceProtocol,也就是说一个 usb interface 只要其这三个项特点与这张表里定义的任何一行匹配,那么就可以把这个设备和这个驱动绑定,但是这三个条件毕竟只是规定了一些主流的或者说常规的设备的特点,这个世界上 usb mass storage 设备有许多种,林子大了,什么鸟都有.如果你的设备不符合这三个条件但是你仍然要用这个驱动怎么办?或者说你的设备不一般,在正式工作之前需要进行一些初始化的操作,你想自己提供一个初始化函数,那怎么办?伟大的 Linux 内核开发者们为你准备了一个文件,它就是让诸多 usb mass storage 设备厂家欢欣鼓舞的 unusual_devs.h,有了它,厂家们不用再为自己的设备不被 Linux 内核支持而烦恼了.虽然我们的 U 盘基本上不属于这些另类设备,但作为一个有责任心的社会主义新青年,我们有必要为这个文件说两句.

冬天来了,春天还会远吗?(二)

我们打开 unusual_devs.h 吧,随便看一下,发现里边就是很多一个个 UNUSUAL_DEV 宏,每一行就是这么一个宏,毫无疑问它对应一种设备,我们从其中挑一个来看,比如挑一个三星的吧,过去在 Intel 的时候,前辈们会说,若不是当初我们对自己太自信了,这个世界上又怎么有三星的生存空间.说的是上世纪末,Intel 决定提高 flash 产品的价格,而 Nokia 这个大客户不干了,它想找别人,一个叫三星的小公司鬼魅般的出现了,没

有人相信这样一个小公司能够满足 Nokia,可是,韩国人,韩国人的韧劲不仅仅是体现在足球场上.于是,世界上有了三星,Nokia 养活了三星,而 Intel,在 flash 这一领域失去了一个重要的客户,副 CEO 也为此引咎辞职了.而下面这个设备,正是来自三星的一个 flash 产品.

```
711 /* Submitted by Hartmut Wahl <hwahl@hwahl.de>*/
712 UNUSUAL_DEV( 0x0839, 0x000a, 0x0001, 0x0001,
713                 "Samsung",
714                 "Digimax 410",
715                 US_SC_DEVICE, US_PR_DEVICE, NULL,
716                 US_FL_FIX_INQUIRY),
```

Digimax 410,熟悉数码相机的哥们儿大概对三星的这款 410 万像素 3 倍光学变焦的产品不会陌生,不过数码相机更新得很快,这款 2002 年推出的相机如今当然也算是很一般了,市场上卖的话也就 1500 以下,不过当时刚推出的时候也是 3000 到 4000 元的.ok,我们来看这一行是什么意思.

UNUSUAL_DEV 这个宏被定义过两次,当然,#define 了之后再下一次#define 之前有一个#undef,否则就重复定义了.在 storage_usb_ids 之前,它的定义是

```
116 #define UNUSUAL_DEV(id_vendor, id_product, bcdDeviceMin, bcdDeviceMax, \
117                     vendorName, productName,useProtocol, useTransport, \
118                     initFunction, flags) \
119 { USB_DEVICE_VER(id_vendor, id_product, bcdDeviceMin,bcdDeviceMax) }
```

USB_DEVICE_VER 的定义在 include/linux/usb.h 中,

```
448 /**
449  * USB_DEVICE_VER - macro used to describe a specific usb device with a version range
450  * @vend: the 16 bit USB Vendor ID
451  * @prod: the 16 bit USB Product ID
452  * @lo: the bcdDevice_lo value
453  * @hi: the bcdDevice_hi value
454  *
455  * This macro is used to create a struct usb_device_id that matches a
456  * specific device, with a version range.
457  */
458 #define USB_DEVICE_VER(vend,prod,lo,hi) \
459         .match_flags = USB_DEVICE_ID_MATCH_DEVICE_AND_VERSION, .idVendor = (vend), .idProduct = (prod), .bcdDevice_lo          = (lo), .bcdDevice_hi = (hi)
```

所以这行最终出现在 storage_usb_ids 中的意思就是令 match_flags 为 USB_DEVICE_ID_MATCH_DEVICE_AND_VERSION,idVendor 为 0x0839,idProduct 为 0x000a,bcdDevice_lo 为 0x0001,bcdDevice_hi 为 0x0001.

而在 us_unusal_dev_list 这张表之前,UNUSUAL_DEV 又被定义为:

```
168 #undef UNUSUAL_DEV
169 #define UNUSUAL_DEV(idVendor, idProduct, bcdDeviceMin, bcdDeviceMax, \
170                     vendor_name, product_name, use_protocol, use_transport,
\
171                     init_function, Flags) \
172 { \
173         .vendorName = vendor_name,      \
174         .productName = product_name,    \
175         .useProtocol = use_protocol,    \
176         .useTransport = use_transport,  \
177         .initFunction = init_function,  \
178         .flags = Flags, \
179 }
```

Ok.这样这个宏的意思又是令 vendorName 为"Samsung",令 productName 为"Digimax 410", 而 useProtocol 为 US_SC_DEVICE, useTransport 为 US_PR_DEVICE,initFunction 为 NULL,flag 为 US_FL_FIX_INQUIRY.

看明白了吗?首先不去管各项的具体含义,至少我们看出来,针对同一个文件,我们使用两次定义 UNUSUAL_DEV 这个宏的方法,两次利用了它的不同元素,换言之,UNUSUAL_DEV 这个宏本来可以设定 10 个参数,而 storage_usb_ids 中需要使用其中的前 4 个参数,同时 us_unusual_dev_list 中需要使用其中的后 6 个参数,所以在 unusual_devs.h 中定义的一行起了两个作用.我们注意到不管是 storage_usb_ids 数组还是 us_unusual_dev_list,其中都通过这么一行把 unusual_devs.h 文件给包含了进来.storage_usb_ids 中:

```
121 static struct usb_device_id storage_usb_ids [] = {
122
123 #       include "unusual_devs.h"
124 #undef UNUSUAL_DEV
```

us_unusual_dev_list 中:

```
181 static struct us_unusual_dev us_unusual_dev_list[] = {
182 #       include "unusual_devs.h"
183 #       undef UNUSUAL_DEV
```

而我们之所以使用两个数组的原因是,storage_usb_ids 是提供给 usb core 的,它需要比较 driver 和 device 从而确定设备是被这个 driver 所支持的,我们只需要比较四项就可以了,因为这四项已经足以确定一个设备了,厂商,产品,序列号.比较这些就够了,而 us_unusual_dev_list 这个数组中的元素是我们接下来的代码要用的,比如它用什么协议,它有什么初始化函数,所以我们使用了两个数组.而我们需要注意的是,这两个数组中元素的顺序是一样的,所以我们从 storage_usb_ids 中得到的 id_index 用于 us_unusual_dev_list 也是可以的,表示的还是同一个设备.而这也就是我们刚才在 get_device_info 中看到的.

```
472         struct us_unusual_dev *unusual_dev = &us_unusual_dev_list[id_index];
473         struct usb_device_id *id = &storage_usb_ids[id_index];
```

这样,unusual_dev 和 id 就各取所需了.下面我们将会用到这两个指针.暂且不表.

总结陈词,最后具体解释一下这行为三星这款数码相机写的语句,

1. 关于 match_flags,它的值是 USB_DEVICE_ID_MATCH_DEVICE_AND_VERSION,这是一个宏,它就告诉 usb core,要比较这样几个方面,idVendor,idProduct,bcdDevice_lo,bcdDevice_hi,其中 idVendor 和下面的 vendorName 是对应的,而 idProduct 和下面的 productName 是对应的,业内为每家公司编一个号,这样便于管理,比如三星的编号就是 0x0839,那么三星的产品中就会在其设备描述符中 idVendor 的烙上 0x0839.而三星自己的每种产品也会有个编号,和 Digimax 410 对应的编号就是 0x000a,而 bcdDevice_lo 和 bcdDevice_hi 共同组成一个具体设备的编号(device release number),bcd 就意味着这个编号是二进制的格式.

2. vendorName 和 productName 不用再说了, "Samsung"和"Digimax 410".

3. useProtocol 为 US_SC_DEVICE, useTransport 为 US_PR_DEVICE,这种情况就说明对于这种设备,它属于什么 subclass,它使用什么通信协议,得从设备描述符里边去读取,它都写在那里边了.一会我们会看到我们的代码中会如何判断这个的.

4. initFunction 等于 NULL,这个很有意义的,这个函数就是设备的初始化函数,一般的设备都不需要这个函数,但是有些设备它偏要标新立异,它就告诉你,要用我的设备你必须先做一些初始化,于是它提供了一个函数,initFunction 当然是一个函数指针,这里如果不为 NULL 的话,到时候就会被调用,以后我们会看到代码中对这个指针进行了判断.如果为空不理睬,否则就会执行.比如我们看下面两处,惠普的两个设备,它就提供了一个叫做 init_8200e 的初始化函数,

```
63 UNUSUAL_DEV(   0x03f0, 0x0207, 0x0001, 0x0001,
64                 "HP",
65                 "CD-Writer+ 8200e",
66                 US_SC_8070, US_PR_SCM_ATAPI, init_8200e, 0),
67
68 UNUSUAL_DEV(   0x03f0, 0x0307, 0x0001, 0x0001,
69                 "HP",
70                 "CD-Writer+ CD-4e",
71                 US_SC_8070, US_PR_SCM_ATAPI, init_8200e, 0),
```

5. flag 等于 US_FL_FIX_INQUIRY,这个 flag 可以设为很多值,这个 flag 的存在本身就表示这个设备有些与众不同,因为一般的设备是不用这个 flag 的,有这个 flag 就表明这个设备可能在某些地方需要进行特殊的处理,所以今后在代码中我们会看到突然跳出一句,判断 us->flag 等于某个咚咚不,如果等于,就执行一些代码,如果不等于,那就不做任何事情.这个 flag 的存在也使得我们可以方便处理一些设备的 bug,比如正常的设备你问它吃了吗?它就回答吃了.可是不正常的设备可能就会根本不回答,或者回答北京房价真贵!于是对于这种设备,可能我们就需要一些专门的代码来对付.具体到这个 US_FL_FIX_INQUIRY,这个 flag 这么一设置,就表明这个设备在接受到 INQUIRY 命令的时候会有一些异常的特征,所以以后我们会

在代码里看到我们是如何处理它的.设置了这个 flag 的当然不只是三星的这款相机,别的设备也有可能设置的.

6. 既然明白了 unusual_devs.h 的作用,那么很显然的一个事情,如果一个厂家推出了一个新的设备,它有一些新的特征,而目前的设备驱动不足以完全支持它,那么厂家首先需要做的事情就是在 unusual_devs.h 中添加一个 UNUSUAL_DEV 来定义自己的设备,然后再看是否需要给内核打补丁以及如何打.因此这几年 unusual_devs.h 这个文件的长度也是慢慢在增长.

冬天来了,春天还会远吗?(三)

从两张表得到了我们需要的冬冬,然后下面的代码就是围绕着这两个指针来展开了.(unusual_dev 和 id)

476 行,把 unusual_dev 给记录在 us 里边,反正 us 里边也有这么一个成员.这样记录下来日后要用起来就方便了,因为 us 是贯穿整个故事的,所以访问他的成员很方便,随时都可以,但是 us_unusual_dev_list 以及 storage_usb_ids 这两张表这次之后就不会再用了.因为我们已经得到了我们想要的,所以我们就不用再去骚扰这两个数组了.

477 至 483 行,给 us 的另外三个成员赋值,subclass,protocol,flags.比如我们的 U 盘,它属于主流设备,在 us_unusual_dev_list 列表中能找到它,其 subclass 是 US_SC_SCSI,而 protocol 是 Bulk-only,即这里用宏 US_PR_BULK 所代表的.(224 行和 225 行.)关于 US_SC_DEVICE 和 US_PR_DEVICE 我们之前讲那个三星的数码相机的时候已经看到了,它就表示 subclass 和 protocol 得从设备的描述符里边读出来.这样做看起来很滑稽,因为三星完全可以把 subclass 和 protocol 在 UNUSUAL_DEV 中写清楚,何必让我们再去读设备描述符呢.然而,我们可以想象,它这样的好处是定义一个 UNUSUAL_DEV 可以代表几种设备,即它可以让几个不同 subclass 的设备共用这么一个宏,或者几个不同 protocol 的设备共用这么一个宏.能省一点就省一点吧,这里体现了开源社区人们勤俭节约的高尚品德. 需要特别指出的是 us->flags,对于 U 盘来说,它当然没有什么 flags 需要设定,但是 unusual_devs.h 中的很多设备都设置了各种 flags,稍后在代码中我们会看到,时不时我们就得判断一下是否某个 flag 设置了,通常是如果设置了,就要多执行某段代码,以满足某种要求.

490 至 515 行,这是一段纯粹的调试代码,对我们理解 usb 没有任何意义的.这段代码检查 unusual_devs.h,看是否这个文件定义了一行没有意义的句子.什么叫没有意义?我们刚才看见了,如果这个设备设了 US_SC_DEVICE,那么其 subclass 将从描述符中读出来,如果不然,则让 subclass=unusual_dev->useProtocol,但是如果后者又真的和描述符里读出来的一样,那么这个设备就没有必要把自己的 useProtocol 定义在 unusual_devs.h 中了,因为反正也可以从描述符里读出来.还不如和大众一样设为 US_SC_DEVICE 得了.就比如我们来看下面这行代表一个 Sony 的 Memory Stick 产品的代码:

```
371 UNUSUAL_DEV(  0x054c, 0x0069, 0x0000, 0x9999,
372                 "Sony",
373                 "Memorystick MSC-U03",
374                 US_SC_UFI, US_PR_CB, NULL,
375                 US_FL_SINGLE_LUN ),
```

我们看到其 useProtocol 这一栏里写了 US_SC_UFI,这表明它自称是属于 UFI 这个 subclass 的,但是如果我们从它的描述符里边读出来也是这个,那就没有必要注明在这里了,这里直接写成 US_SC_DEVICE 好了.当然,总的来说这段代码有些傻.写代码的是希望能够更好的管理 unusual_devs.h,希望它不要不断的增加,他总希望能够从这个文件里删除一些行,并且即使不能删除一行,也希望每一行都看上去整齐一点,让这个文件看上去更加小巧玲珑,更加精致.而不是无休的增加,不息的扩充.于是我们也只能对其良苦用心多一份理解吧.

518 至 526 行,这里调用了一个来自 usb core 的函数,usb_string,这个函数的作用是获得一个设备的字符串描述符.咦?怎么跳出来一个叫做字符串描述符的冬冬?之前不是只讲了四种描述符吗?没错,设备描述符,配置描述符,接口描述符,端点描述符,这是每个设备都有的,而还有些描述符是可有可无的,字符串描述符就是这么一种情况,有的设备有,有的设备没有.又比如,hub,它就有一个 hub 描述符,这当然是一般的设备没有的.那么字符串描述符是干嘛的呢?有些东西模糊一些也未偿不是一件好事,看得太透彻了才知道很残酷.如果你一定要知道的话,举个例子,我的机器里很多 usb 设备,有一个和 lspci 类似的命令,可以查看一下,这个命令就是 lsusb.你也可以试一下,安装一个软件包 usbutils,然后就可以使用这个命令.我们看:

```
localhost:~/test # lsusb
Bus 004 Device 003: ID 0ea0:1001 Ours Technology, Inc.
Bus 004 Device 002: ID 04b4:6560 Cypress Semiconductor Corp. CY7C65640 USB-2.0
"TetraHub"
Bus 004 Device 001: ID 0000:0000
Bus 003 Device 001: ID 0000:0000
Bus 002 Device 002: ID 0624:0294 Avocent Corp.
Bus 002 Device 001: ID 0000:0000
Bus 001 Device 001: ID 0000:0000
```

看这个第二行, Cypress Semiconductor Corp., 这么一长串的东西,你说哪来的?是不是应该从设备里来?设备的那几个标准描述符,整个描述符的大小也不一定放得下这么一长串,所以,一些设备专门准备了一些字符串描述符(string descriptor).就用来记这些长串的东西,我们结合刚才的 518 行开始讲,如果设备描述符里边 iManufacturer 不为 0,那么调用 usb_string,这句话具体做了什么?就是根据 iManufactuer 的值得到公司名字,而 iManufactuer 的第一个字母 i,就表示 index,它记录的是真正的公司名字保存在哪一个字符串描述符中,因为字符串描述符可以有多个,那么必然就有个号码来区分,接下来几行,iProduct 记录了产品名在第几个字符串描述符中,iSerialNumber 记录了产品序列号在第几个字符串描述中,然后调用 usb_string 这个函数,就把真正的字符串描述符里的冬冬给记录了下来.我们看到,我们三次调用的时候分别传递了 us->vendor,us->product,us->serial.这样函数调用结束之后,这三个里面就记录了必要的信息,于是以后我们就可以用了.

得到了 us->vendor,us->product,us->serial,那么下面 528 直到 547 行就不需要多讲了,就是说如果得到的东西是空的,(得到的是空可以有两种可能,一个是设备根本就没提供这些字符串描述符,另一种情况是 usb_string 函数没能成功,但是这个函数不成功也无所谓,没影响.)那也没关系,毕竟这些信息我们可有可无,无非是打印出来给客户看看.如果 unusual_dev 里边有的话,那就拷贝过来,如果也没有,那没办法,设为 Unknown.而序列号这个就索性置为 None 好了,最后 US_DEBUGP 把这些信息给打印出来,如果你打开了 debug 开关,那么你会在日志文件里看到这么一句话,在/var/log/messages 里边.

至此,get_device_info 这个函数就结束了他的使命.在 usb storage 这部戏里,他将不再出场.但我想说,对于 usb storage 这整个模块来说,主角配角不重要,每个函数都是画布上的一抹色彩.就像我们每一个人,不也是别人人生中的配角,但总是自己人生的主角吗?

冬天来了,春天还会远吗(四)

结束了 get_device_info,我们继续沿着 storage_probe 一步一步走下去.为了保持原汁原味,我们贴代码的原则是一个函数的每一行都贴出来.get_device_info 是 962 行,我们已经贴过,所以下面从 963 行开始了.

```
963
964 #ifdef CONFIG_USB_STORAGE_SDDR09
965         if (us->protocol == US_PR_EUSB_SDDR09 ||
966                         us->protocol == US_PR_DPCM_USB) {
967                 /* set the configuration -- STALL is an acceptable response here */
968                 if (us->pusb_dev->actconfig->desc.bConfigurationValue != 1) {
969                         US_DEBUGP("active config #%d != 1 ??\n", us->pusb_dev
970                                 ->actconfig->desc.bConfigurationValue);
971                         goto BadDevice;
972                 }
973                 result = usb_reset_configuration(us->pusb_dev);
974
975                 US_DEBUGP("Result of usb_reset_configuration is %d\n", result);
976                 if (result == -EPIPE) {
977                         US_DEBUGP("-- stall on control interface\n");
978                 } else if (result != 0) {
979                         /* it's not a stall, but another error -- time to bail */
980                         US_DEBUGP("-- Unknown error.  Rejecting device\n");
981                         goto BadDevice;
982                 }
983         }
984 #endif
```

看到这段代码,我笑了.因为#ifdef CONFIG_USB_STORAGE_SDDR09 说明这段代码跟我们无关.关于这些编译选项我们前面已然提过,第六感告诉我们这个选项是针对某种特殊产品的,对于这种特殊的产品,它在某些方面有它自己的要求,所以它会有它特殊的代码.具体到这个选项,我们看一下 drivers/usb/storage/Kconfig 文件,这个文件里边介绍了该目录下每一个编译选项的作用.

```
 99 config USB_STORAGE_SDDR09
100         bool "SanDisk SDDR-09 (and other SmartMedia) support (EXPERIMENTAL)"
101         depends on USB_STORAGE && EXPERIMENTAL
102         help
103           Say Y here to include additional code to support the Sandisk SDDR-09
104           SmartMedia reader in the USB Mass Storage driver.
```

```
105             Also works for the Microtech Zio! SmartMedia reader.
```

可以看到,如果要支持 Sandisk SDDR-09 SmartMedia 的读卡器,那你就打开这个编译选项吧.Sandisk 是公司名,玩数码的人对这家公司不会陌生,中文名叫晟碟,这是一家全球最大的闪存数据存储产品供应商. 而 SmartMedia(简称 SM)卡通常用于数码相机中,也曾一度被用于 MP3 中,它也是一种 flash memory 存储卡.不过如今市面上很少有 SM 卡了,因为其兼容性不好.眼下用得比较多的应该是 CF 卡(Compact Flash)了.而读卡器就是用来把卡里边的数据读出来,使用 USB 接口,从原理上来看和 U 盘也是差不多. (顺便介绍一下,U 盘和存储卡的区别吧,我们所讲的 U 盘，就是可以直接读写的存储器,而存储卡需要外部设备才能进行访问,如手机的闪存卡,数码相机的闪存卡等就只是一张卡,电脑不能直接对其进行访问,这就需要一种叫"读卡器"的外部设备进行识别,存储卡有多种,如 XD、CF、SD、SM 等等,有些读卡器具有"多合一"的功能,可以对不同的闪存卡进行读写.而我们这段代码里以及接下来的代码中每一个条件编译开关显然对应的是一种读卡器.她们属于不同的厂商的不同的产品.)

继续,这就是我们前面提到过的三个函数.get_transport,get_protocol,get_pipes.一旦结束了这三个函数,我们就将进入本故事的高潮部分.而在这之前,我们只能一个一个来看.好在这几个函数虽然不短,但是真正有用的信息只有一点点,所以可以很快的看完.

```
985
986         /* Get the transport, protocol, and pipe settings */
987         result = get_transport(us);
988         if (result)
989                 goto BadDevice;
990         result = get_protocol(us);
991         if (result)
992                 goto BadDevice;
993         result = get_pipes(us);
994         if (result)
995                 goto BadDevice;
```

第一个,get_transport(us),

```
549 /* Get the transport settings */
550 static int get_transport(struct us_data *us)
551 {
552         switch (us->protocol) {
553         case US_PR_CB:
554                 us->transport_name = "Control/Bulk";
555                 us->transport = usb_stor_CB_transport;
556                 us->transport_reset = usb_stor_CB_reset;
557                 us->max_lun = 7;
558                 break;
559
560         case US_PR_CBI:
561                 us->transport_name = "Control/Bulk/Interrupt";
562                 us->transport = usb_stor_CBI_transport;
```

```
563                 us->transport_reset = usb_stor_CB_reset;
564                 us->max_lun = 7;
565                 break;
566
567         case US_PR_BULK:
568                 us->transport_name = "Bulk";
569                 us->transport = usb_stor_Bulk_transport;
570                 us->transport_reset = usb_stor_Bulk_reset;
571                 break;
572
573 #ifdef CONFIG_USB_STORAGE_HP8200e
574         case US_PR_SCM_ATAPI:
575                 us->transport_name = "SCM/ATAPI";
576                 us->transport = hp8200e_transport;
577                 us->transport_reset = usb_stor_CB_reset;
578                 us->max_lun = 1;
579                 break;
580 #endif
581
582 #ifdef CONFIG_USB_STORAGE_SDDR09
583         case US_PR_EUSB_SDDR09:
584                 us->transport_name = "EUSB/SDDR09";
585                 us->transport = sddr09_transport;
586                 us->transport_reset = usb_stor_CB_reset;
587                 us->max_lun = 0;
588                 break;
589 #endif
590
591 #ifdef CONFIG_USB_STORAGE_SDDR55
592         case US_PR_SDDR55:
593                 us->transport_name = "SDDR55";
594                 us->transport = sddr55_transport;
595                 us->transport_reset = sddr55_reset;
596                 us->max_lun = 0;
597                 break;
598 #endif
599
600 #ifdef CONFIG_USB_STORAGE_DPCM
601         case US_PR_DPCM_USB:
602                 us->transport_name = "Control/Bulk-EUSB/SDDR09";
603                 us->transport = dpcm_transport;
604                 us->transport_reset = usb_stor_CB_reset;
605                 us->max_lun = 1;
606                 break;
```

```
607 #endif
608
609 #ifdef CONFIG_USB_STORAGE_FREECOM
610         case US_PR_FREECOM:
611                 us->transport_name = "Freecom";
612                 us->transport = freecom_transport;
613                 us->transport_reset = usb_stor_freecom_reset;
614                 us->max_lun = 0;
615                 break;
616 #endif
617
618 #ifdef CONFIG_USB_STORAGE_DATAFAB
619         case US_PR_DATAFAB:
620                 us->transport_name  = "Datafab Bulk-Only";
621                 us->transport = datafab_transport;
622                 us->transport_reset = usb_stor_Bulk_reset;
623                 us->max_lun = 1;
624                 break;
625 #endif
626
627 #ifdef CONFIG_USB_STORAGE_JUMPSHOT
628         case US_PR_JUMPSHOT:
629                 us->transport_name  = "Lexar Jumpshot Control/Bulk";
630                 us->transport = jumpshot_transport;
631                 us->transport_reset = usb_stor_Bulk_reset;
632                 us->max_lun = 1;
633                 break;
634 #endif
635
636         default:
637                 return -EIO;
638         }
639         US_DEBUGP("Transport: %s\n", us->transport_name);
640
641         /* fix for single-lun devices */
642         if (us->flags & US_FL_SINGLE_LUN)
643                 us->max_lun = 0;
644         return 0;
645 }
```

咋一看,这么长一段,用长沙话讲,这叫非洲老头子跳高—吓(黑)老子一跳.(长沙话"黑"和"吓"一个音)不过明眼人一看,就知道了,主要就是一个switch,选择语句,语法上来说很简单,谭浩强大哥的书里边介绍的很清楚.所以我们看懂这段代码不难,只是,我想说的是,虽然这里做出一个选择不难,但是不同选择就意味着后来整个故事会有千差万别的结局,当鸟儿选择在两翼上系上黄金,就意味着它放弃展翅高飞;选择云天搏击,

就意味着放弃身外的负累.所以,此处,我们需要仔细的看清楚我们究竟选择了怎样一条路.很显然,前面我们已经说过,对于 U 盘,spec 规定了,它就属于 Bulk-only 的传输方式,即它的 us->protocol 就是 US_PR_BULK.这是我们刚刚在 get_device_info 中确定下来的.于是,在整个 switch 段落中,我们所执行的只是 US_PR_BULK 这一段,即,

us 的 transport_name 被赋值为"Bulk",transport 被赋值为 usb_stor_Bulk_transport,transport_reset 被赋值为 usb_stor_Bulk_reset.其中我们最需要记住的是,us 的成员 transport 和 transport_reset 是两个函数指针.程序员们把这个称作钩子.这两个赋值我们需要牢记,日后我们定会用到它们的,因为这正是我们真正的数据传输的时候调用的冬冬.关于 usb_stor_Bulk_*的这两个函数,咱们到时候调用了再来看.现在只需知道,日后我们一定会回过来看这个赋值的.

573 行到 634 行,不用多说了,这里就全是与各种特定产品相关的一些编译开关,它们有些自己定义一些传输函数,有些则共用那些通用的函数.

641 行,判断 us->flags,还记得我们在讲 unusual_devs.h 文件的时候说的那个 flags 吧,这里第一次用上了.有些设备设置了 US_FL_SINGLE_LUN 这么一个 flag,就表明它是只有一个 LUN 的.像这样的设备挺多的,随便从 unusual_devs.h 中抓一个出来:

```
338 UNUSUAL_DEV(  0x054c, 0x002d, 0x0100, 0x0100,
339                 "Sony",
340                 "Memorystick MSAC-US1",
341                 US_SC_DEVICE, US_PR_DEVICE, NULL,
342                 US_FL_SINGLE_LUN ),
```

比如这个 Sony 的 Memorystick.中文名叫记忆棒,大小就跟箭牌口香糖似的,也是一种存储芯片,是 Sony 公司推出的,广泛用于 Sony 的各种数码产品中.比如数码相机,数码摄影机.

有人问了,啥是 LUN 啊?logical unit number.通常在谈到 scsi 设备的时候不可避免的要说起 LUN.关于 LUN,曾几何时,一位来自 Novell(SUSE)的参与开发 Linux 内核中 usb 子系统的工程师这样对我说,一个 lun 就是一个 device 中的一个 drive.换言之,usb 中引入 lun 的目的在于,举例来说,有些读卡器可以有多个插槽,比如就是两个,其中一个支持 CF 卡,另一个支持 SD 卡,那么这种情况要区分这两个插槽里的冬冬,就得引入 lun 这么一个词.这叫逻辑单元.很显然,像 U 盘这样简单的设备其 LUN 必然是一个.有时候,人们常把 U 盘中一个分区当作一个 LUN,这样说可能对小学三年级以下的朋友是可以接受的,但是作为一个成年人,不应该这么理解.

知道了 LUN 以后,自然就可以知道 US_FL_SINGLE_LUN 是干嘛了,这个 flag 的意义很明显,直截了当的告诉你,我这个设备只有一个 LUN,它不支持多个 LUN.而 max_lun 又是什么意思?us 中的成员 max_lun 等于一个设备所支持的最大的 lun 号.即如果一个设备支持四个 LUNs,那么这四个 LUN 的编号就是 0,1,2,3,而 max_lun 就是 3.如果一个设备不用支持多个 LUN,那么它的 max_lun 就是 0.所以这里 max_lun 就是设为了 0.

另外一个需要注意的地方是,比较一下各个 case 语句,发现, US_PR_BULK 和别的 case 不一样,别的 case 下面都设置了 us->max_lun,而对应于 Bulk-Only 协议的这个 case,它没有设置 us->max_lun,这是为何?别急,后来我们会专门有一个函数去读取这个值的,之所以不设,是因为这个值由设备说了算,必须向设备

查询,这是 Bulk-Only 协议规定的.所以我们之后会遇见 usb_stor_Bulk_max_lun()函数,它将负责获取这个 max lun.而我依然要声明一次,这个函数对我们 U 盘是没啥意义的,咱们这个值肯定是 0.

至此,get_transport()也结束了,和 get_device_info 一样.我们目前所看到的这些函数都不得不面对现实,对它们来说,凋谢是最终的结果,盛开只是一个过程...而对我们来说,要到达终点,那么和这些函数狭路相逢,终不能幸免.然而,不管这部分代码有多么重要,也不过是我们整个长途旅程中,来去匆匆的转机站,无论停留多久,始终要离去坐另一班机.

冬天来了,春天还会远吗?(五)

道不尽红尘舍恋诉不完人间恩恩怨怨.

看完了 get_transport()继续看 get_protocol()和 get_pipes().仍然是来自 drivers/usb/storage/usb.c 中:

```
647 /* Get the protocol settings */
648 static int get_protocol(struct us_data *us)
649 {
650         switch (us->subclass) {
651         case US_SC_RBC:
652                 us->protocol_name = "Reduced Block Commands (RBC)";
653                 us->proto_handler = usb_stor_transparent_scsi_command;
654                 break;
655
656         case US_SC_8020:
657                 us->protocol_name = "8020i";
658                 us->proto_handler = usb_stor_ATAPI_command;
659                 us->max_lun = 0;
660                 break;
661
662         case US_SC_QIC:
663                 us->protocol_name = "QIC-157";
664                 us->proto_handler = usb_stor_qic157_command;
665                 us->max_lun = 0;
666                 break;
667
668         case US_SC_8070:
669                 us->protocol_name = "8070i";
670                 us->proto_handler = usb_stor_ATAPI_command;
671                 us->max_lun = 0;
672                 break;
673
674         case US_SC_SCSI:
675                 us->protocol_name = "Transparent SCSI";
```

```
676                 us->proto_handler = usb_stor_transparent_scsi_command;
677                 break;
678
679         case US_SC_UFI:
680                 us->protocol_name = "Uniform Floppy Interface (UFI)";
681                 us->proto_handler = usb_stor_ufi_command;
682                 break;
683
684 #ifdef CONFIG_USB_STORAGE_ISD200
685         case US_SC_ISD200:
686                 us->protocol_name = "ISD200 ATA/ATAPI";
687                 us->proto_handler = isd200_ata_command;
688                 break;
689 #endif
690
691         default:
692                 return -EIO;
693         }
694         US_DEBUGP("Protocol: %s\n", us->protocol_name);
695         return 0;
696 }
```

这段代码非常的浅显易懂.我相信即使去问上海火车站附近那些卖黑车的哥们儿,他们也能告诉你这段代码做了什么.就一件事,根据 us->subclass 来判断.对于 U 盘来说,spec 里边规定了,它的 subclass 是 US_SC_SCSI,所以这里就是两句赋值语句.一个是令 us 的 protocol_name 为"Transparent SCSI",另一个是令 us 的 proto_handler 为 usb_stor_transparent_scsi_command.后者又是一个函数指针,我们日后必将不可避免的遇到这个函数,暂且不表.

然后是 get_pipes().drivers/usb/storage/usb.c:

```
698 /* Get the pipe settings */
699 static int get_pipes(struct us_data *us)
700 {
701         struct usb_host_interface *altsetting =
702                 us->pusb_intf->cur_altsetting;
703         int i;
704         struct usb_endpoint_descriptor *ep;
705         struct usb_endpoint_descriptor *ep_in = NULL;
706         struct usb_endpoint_descriptor *ep_out = NULL;
707         struct usb_endpoint_descriptor *ep_int = NULL;
708
709         /*
710          * Find the endpoints we need.
711          * We are expecting a minimum of 2 endpoints - in and out (bulk).
712          * An optional interrupt is OK (necessary for CBI protocol).
713          * We will ignore any others.
```

```
714          */
715         for (i = 0; i < altsetting->desc.bNumEndpoints; i++) {
716                 ep = &altsetting->endpoint[i].desc;
717
718                 /* Is it a BULK endpoint? */
719                 if ((ep->bmAttributes & USB_ENDPOINT_XFERTYPE_MASK)
720                                 == USB_ENDPOINT_XFER_BULK) {
721                         /* BULK in or out? */
722                         if (ep->bEndpointAddress & USB_DIR_IN)
723                                 ep_in = ep;
724                         else
725                                 ep_out = ep;
726                 }
727
728                 /* Is it an interrupt endpoint? */
729                 else if ((ep->bmAttributes & USB_ENDPOINT_XFERTYPE_MASK)
730                                 == USB_ENDPOINT_XFER_INT) {
731                         ep_int = ep;
732                 }
733         }
734
735         if (!ep_in || !ep_out || (us->protocol == US_PR_CBI && !ep_int)) {
736                 US_DEBUGP("Endpoint sanity check failed! Rejecting dev.\n");
737                 return -EIO;
738         }
739
740         /* Calculate and store the pipe values */
741         us->send_ctrl_pipe = usb_sndctrlpipe(us->pusb_dev, 0);
742         us->recv_ctrl_pipe = usb_rcvctrlpipe(us->pusb_dev, 0);
743         us->send_bulk_pipe = usb_sndbulkpipe(us->pusb_dev,
744                 ep_out->bEndpointAddress & USB_ENDPOINT_NUMBER_MASK);
745         us->recv_bulk_pipe = usb_rcvbulkpipe(us->pusb_dev,
746                 ep_in->bEndpointAddress & USB_ENDPOINT_NUMBER_MASK);
747         if (ep_int) {
748                 us->recv_intr_pipe = usb_rcvintpipe(us->pusb_dev,
749                         ep_int->bEndpointAddress &
USB_ENDPOINT_NUMBER_MASK);
750                 us->ep_bInterval = ep_int->bInterval;
751         }
752         return 0;
753 }
```

这个函数应该可以说是我们这几个无聊的函数中最后一个了,但它也是相对来说最复杂的一个.请容我慢慢给您道来.702 行,us->pusb_intf,可还记得,在 associate_dev 中赋得值,如不记得请回过去查一下.

没错,us->pusb_intf 就是我们故事中最开始一再提到的那个 interface(指针).而它的成员 cur_altsetting,就是当前的 setting,或者说设置.在讲 associate_dev 的时候也已经遇到过,是一个 struct usb_host_interface 的结构体指针.现在这里用另一个指针临时代替一下, altsetting.接下来会用到它的成员,desc 和 endpoint.回顾 struct usb_host_interface,可以看到,它这两个成员,struct usb_interface_descriptor desc,和 struct usb_host_endpoint *endpoint.其中,desc 不用多说,正是这个 interface 的接口描述符,而 endpoint 这个指针记录的是几个 endpoint,它们以数组的形式被存储,而 endpoint 指向数组头.这些冬冬都是在 usb core 枚举的时候就设置好了,我们无需操任何心,只需拿来用就是了.这里给出 struct usb_host_endpoint 的定义,来自 include/linux/usb.h:

```
43 /* host-side wrapper for parsed endpoint descriptors */
44 struct usb_host_endpoint {
45          struct usb_endpoint_descriptor   desc;
46
47          unsigned char *extra;    /* Extra descriptors */
48          int extralen;
49 };
```

接着定义了几个 struct usb_endpoint_descriptor 的结构体指针.顾名思义,这就是对应 endpoint 的描述符的.其定义来自于 include/linux/usb_ch9.h:

```
260 /* USB_DT_ENDPOINT: Endpoint descriptor */
261 struct usb_endpoint_descriptor {
262          __u8   bLength;
263          __u8   bDescriptorType;
264
265          __u8   bEndpointAddress;
266          __u8   bmAttributes;
267          __u16 wMaxPacketSize;
268          __u8   bInterval;
269
270          // NOTE:   these two are _only_ in audio endpoints.
271          // use USB_DT_ENDPOINT*_SIZE in bLength, not sizeof.
272          __u8   bRefresh;
273          __u8   bSynchAddress;
274 } __attribute__ ((packed));
```

至此,四大描述符一一亮相,在继续讲之前,我们先来小结一下:究竟什么是描述符?每个 USB 设备都有这四大描述符,不过我们拿 U 盘来说.听说过 Flash Memory 吗?Intel,三星,这些都是做 Flash Memory 的,当然通常人们就简称 Flash. Flash 在 U 盘中扮演什么角色?Flash 是用来给用户存储数据的,而 U 盘中的 Flash 就相当于 PC 机中的硬盘,存储数据主要就靠它.那么除了给用户存储数据以外,设备自己还需要存储一些设备本身固有的东西,比如设备姓甚名谁?谁生产的?还有一些信息,比如该设备有几种配置,有几个接口,等等许多特性,这些东西怎么办?复旦大学 97 电工四大才子之一,我在 Intel 的老师加师兄曾经这样对我说:这个世界上,除了 Flash memory 外,还有一个咚咚叫做 EEPROM,也是用来存储的,它是 EEPROM 的前身, 而 Flash 是基于 EEPROM 技术发展起来的一种低成本的 ROM 产品. EEPROM 和 Flash 相同,都是需要电擦除,但 EEPROM 可以按字节擦除,而不向 Flash 那样一次擦除一个 block,这样在只需改动很少数

据的情况下使用 EEPROM 就很方便了.因此 EEPROM 的这一特性,它的电路要复杂些,集成度不高,一个 bit 需要两个管子,一个用来储存电荷信息,一个充当开关.所以 EEPROM 的成本高,Flash 简化了一些电路,成本降低了很多.因此,通常,USB 设备里边,会有一个 Flash 芯片,会有一个 EEPROM 芯片,Flash 给客户存储数据,而 EEPROM 用来存储设备本身的信息.这就是为什么当我们把 Flash 芯片卖给 Motorola 之后,客户看到的手机厂商是摩托罗拉而不是我们 Intel,因为我们虽然在做 Flash 的时候把我们的厂商 ID 写在了 Flash 上,但是最终的成品对外来看,提供的信息都是来自 EEPROM,所以当你把 USB 设备通过 USB 接口连到电脑上去,那么电脑上如果能显示厂家,那么一定是最终的包装厂家,而不可能是里边那块 Flash 的厂家.而 EEPROM 里边写什么?按什么格式写?这正是 usb spec 规定的,这种格式就是一个个的描述符的格式.设备描述符,配置描述符,接口描述符,端点描述符,以及其它一些某一些类别的设备特有的描述符,比如 hub 描述符.这些东西都是很规范的,尤其对于这四种标准的描述符,每个 usb 设备都是规规矩矩的支持的,所以 usb core 层可以用一段相同的代码把它们都给读出来,而不用再让我们设备驱动程序去自己读了,这就是权力集中的好处,反正大家都要做的事情,干脆让上头一起做了好了,这样的领导真是好啊!

715 到 733 行,循环, bNumEndpoints 就是接口描述符中的成员,表示这个接口有多少个端点,不过这其中不包括 0 号端点,0 号端点是任何一个 usb 设备都必须是提供的,这个端点专门用于进行控制传输,即它是一个控制端点.正因为如此,所以即使一个设备没有进行任何设置,usb 主机也可以开始跟它进行一些通信,因为即使不知道其它的端点,但至少知道它一定有一个 0 号端点,或者说一个控制端点.此外,通常 usb mass storage 会有两个 bulk 端点,用于 bulk 传输,即所谓的批量传输.我们日常的读写 U 盘里的文件,就是属于批量传输,所以毫无疑问,对于 mass storage 设备来说,bulk 传输是它的主要工作方式,道理很简单,我们使用 U 盘就是用来读写文件的,谁没事天天去读它的这种描述符那种描述符呢,吃错药了?和这些描述符打交道无非就是为了帮助我们最终实现读写文件的工作,这才是每一个 usb 存储设备真正的使命.

于是我们来这段循环到底在干嘛, altsetting->endpoint[i].desc,对照 struct usb_host_endpoint 这个结构体的定义,可知,desc 正是一个 struct usb_endpoint_descriptor 的变量.咱们刚刚定义了四个这种结构体的指针,ep,ep_in,ep_out,ep_int,很简单,就是用来记录端点描述符的,ep_in 用于 bulk-in,ep_out 用于 bulk-out,ep_int 用于记录中断端点(如果有的话).而 ep,只是一个临时指针.

我们看 struct usb_endpoint_descriptor,它的成员中, bmAttributes 表示属性,总共 8 位,其中 bit1 和 bit0 共同称为 Transfer Type,即传输类型,即 00 表示控制,01 表示等时,10 表示批量,11 表示中断.而 719 行我们看到, USB_ENDPOINT_XFERTYPE_MASK 这个宏定义于 include/linux/usb_ch9.h 中:

```
286 #define USB_ENDPOINT_XFERTYPE_MASK      0x03    /* in bmAttributes */
287 #define USB_ENDPOINT_XFER_CONTROL       0
288 #define USB_ENDPOINT_XFER_ISOC          1
289 #define USB_ENDPOINT_XFER_BULK          2
290 #define USB_ENDPOINT_XFER_INT           3
```

懂一点 C 语言的人就不难理解,719 行就是判断这个端点描述符描述的是不是一个 Bulk 端点,如果是,继续比较,我们先看 bEndpointAddress,这个 struct usb_endpoint_descriptor 中的另一个成员,也是 8 个 bit,或者说 1 个 byte,其中 bit7 表示的是这个端点的方向,0 表示 OUT,1 表示 IN,OUT 与 IN 是对主机而言.OUT 就是从主机到设备,IN 就是从设备到主机.而宏 USB_DIR_IN 仍然来自 include/linux/usb_ch9.h

```
25 /*
26  * USB directions
```

```
27  *
28  * This bit flag is used in endpoint descriptors' bEndpointAddress field.
29  * It's also one of three fields in control requests bRequestType.
30  */
31 #define USB_DIR_OUT                     0               /* to device */
32 #define USB_DIR_IN                      0x80            /* to host */
```

所以这里意思很明显,就是为了让 ep_in 和 ep_out 指向该指的 endpoint descriptor.

729 就不用再说了,如果这还看不懂的话,可以考虑去复旦的绝情谷找到那棵老树上吊自杀了,可惜如今学校里不断搞建设,绝情谷可能已经不复存在了.729 这一个 else if 的作用就是如果这个端点是中断端点,那么就让 ep_int 指向它.我们说了,每一类 usb 其上面有多少端点有何种端点都是不确定的,都得遵守该类设备的规范,而 usb mass storage 的规范说了,一个 usb mass storage 设备至少应该有两个 bulk 端点,除此之外,那个控制端点显然是必须的,毋庸置疑,另外,可能会有一个中断端点,这种设备支持 CBI 协议,即 Control/Bulk/Interrupt 协议.我们也说过了,U 盘遵守的是 Bulk-only 协议,它不需要有中断端点.

735 到 738 这段代码,没啥好说的,就是判断是否 ep_in 或者 ep_out 不存在,或者是遵守 CBI 协议但是没有中断端点,这些都是不合理的,当然就会出错啰!

剩下一小段代码,我们下节再看.需要说的是,这个函数结束之后我们将开始最精彩的部分,它就是伟大的 usb_stor_acquire_resources().黑暗即将过去,黎明已经带我们上路.让我们共同期待吧.同时,我们小结一下,此前我们花了很大的篇幅来为 usb_stor_acquire_resources()做铺垫,那我们来回顾一下,究竟做了哪些事情?

首先我们从 storage_probe 出发,一共调用了五个函数,它们是 assocaite_dev,get_device_info,get_transport,get_protocol,get_pipes.我们这样做的目的是什么?很简单,就是为了建立一个数据结构,它就是传说中的 struct us_data,它的名字叫做 us.我们把她建立了起来,为她申请了内存,为她的各个元素赋了值,目的就是为了让以后我们可以很好的利用她.这五个函数都不难,你一定也会写.难的是如何去定义 struct us_data,别忘了这个数据结构是写代码的同志们专门为 usb-storage 模块而设计的.子曾经曰过,所谓编程,无非就是数据结构加上算法.没错,这个定义于 drivers/usb/storage/usb.h 中的数据结构长达 60 行,她正是整部戏的主角.关于她的成员,我们还有很多没遇到,不过别急,后面会遇到的.好了,虽然 get_pipes 还有一小段没讲,但是我们可以提前和这 5 个函数说再见了,席慕蓉说过,若不得不分离,也要好好的说声再见,也要在心里存着一份感谢,谢谢她给你一份记忆.

通往春天的管道

1990 年,两伊战争,电台里报道早间新闻,播音员说:各位听众朋友,昨天伊拉克截断了科威特的两条输卵管道.(输油管道)

此时,树无语天无语人无语.

一年后,公元 1991 年,一个芬兰人写了一个叫做 Linux 的操作系统,他也觉得这位播音员很有趣,给听众朋友们带来了欢乐.于是为了纪念这件经典的口误,这个芬兰人在 Linux 中引入了管道这么一个概念,并且他把管道用在很多地方,文件系统中,设备驱动中,于是后来我们看到在 Linux 中有了各种各样的管道,不过,没有

输油管道,更没有输卵管道.但是相同的是,所有的管道都是用来传输东西的,只不过有些管道传输的是实实在在的物质,而有些管道传输的是数据.

眼下我们在 usb 代码中看到的管道就是用来传输数据,用来通信. 通信是双方的,不可能自言自语对吧.而在 usb 的通信中,一方肯定是主机,即 host,另一方是什么?是设备吗?在你少不更事的岁月里,你可以说是设备,你可以说主机和设备进行通信.但是,现在,当你已经长大,当你已经成熟,在这样一个激情燃烧的岁月里,你应该说得更确切一点.真正和主机进行通信的是设备内的端点.关于端点,我们也可以专业一点说,从硬件上来看它是实实在在存在的,它被实现为一种 FIFO,支持多少个端点是接口芯片的一个重要指标.而从概念上来说,端点是主机和 usb 设备之间通信流的终点.主机和设备可以进行不同种类的通信,或者说数据传输,首先,设备连接在 usb 总线上,usb 总线为了分辨每一个设备,给每一个设备编上号,然后为了实现多种通信,设备方于是提供了端点,端点多了,自然也要编上号,而让主机最终和端点去联系.

鲁迅先生说得好,世上本没有管道,端点多了,也就有了管道.

这句话说得相当好,非常有道理,两个字,犀利!没错,我们知道,usb 的世界里,有四种管道,它们对应 usb 世界中的四种传输方式,控制传输对应控制管道,中断传输对应中断管道,批量传输对应批量管道,等时传输对应等时管道.每一个 usb 世界中的设备要在 usb 世界里生存,就得有它生存的管道.而管道的出现,正是为了让我们分辨端点,或者说连接端点.记得网友唐伯虎点蚊香曾经说过,主机与端点之间的数据链接就称为管道.

比如说,复旦大学,有一个主校区,也可以说是教学区,(当然也包括一些实验室),上课什么的都得去教学区,把教学区看作主机,那么与其相对的是,另外有很多学生宿舍楼,宿舍楼多了,就给每个楼编上号,比如 1 号楼,2 号楼,...,36 号楼,..,每幢楼算一个设备,比如说你在淘宝网上买了一套阿玛尼外套(当然,肯定是假的,也就一两百的那种),你让人家给你快递过来,人家问你你住哪?你说你住复旦大学,但如果你只说你在复旦大学,那么送快递那哥们可能先得赶到复旦大学正门,或者学生宿舍区的正门,然后人家肯定就得问,你是哪幢楼哪间房?比如你说你是 36 号楼 201,好,那么像 201 这么一个数字呢,就对应端点号,最终那套外套要到达的就是端点 201,而不仅仅是 36 号楼,对吧,假如人家要是送到 36 号楼下就把外套给扔地上了你肯定得跟他急.那么在这个例子里,复旦主校区是主机,每幢宿舍楼算一个设备,你住的那间宿舍就算端点.那么 pipe 呢?pipe 很难与现实中的某一实物对应,不能说她是复旦正门通往宿舍的某条路,而应该按别的方式理解.她包含很多东西,你可以把她比作快递的货物上面贴得那张标签,比如她上面写了收货人的地址,包括多少号楼多少号房,在 usb 里面,就是设备号和端点号,知道了这两个号,货物就能确定它的目的地,而 usb 主机就能知道和她通信的是哪个端点.而 pipe 除了包含着两个号以外,还包含其它一些信息,比如她包含了通信的方向,也就是说,你能从那张标签上得知 36 号楼 201 这个是收件人呢还是寄件人,虽然现实中不需要写明,因为快递员肯定知道你是收件人,他没事才不会和寄件人联系呢.再比如,pipe 里还包含了 pipe 的类型,比如你快递是从深圳递过来,那么怎么递就得看了,快递公司肯定提供不同类型的服务,有的快有的慢,有的可能还有保险什么的,看你出多少钱,让你选择不同的服务类型,同样 pipe 也如此,因为 usb 设备的端点有不同的类型,所以 pipe 里就包含了一个字段来记录这一点,这个字段称为 pipe type.好,让我们来看看实际的 pipe 吧.

来看这些宏,头一个闪亮登场的是 usb_sndctrlpipe,定义于 include/linux/usb.h 中,咱们先把这一堆冬冬相关的代码给列出来.

```
1104 static inline unsigned int __create_pipe(struct usb_device *dev, unsigned int
endpoint)
1105 {
1106         return (dev->devnum << 8) | (endpoint << 15);
1107 }
```

```
1108
1109 /* Create various pipes... */
1110 #define usb_sndctrlpipe(dev,endpoint)    ((PIPE_CONTROL << 30) |
__create_pipe(dev,endpoint))
1111 #define usb_rcvctrlpipe(dev,endpoint)    ((PIPE_CONTROL << 30) |
__create_pipe(dev,endpoint) | USB_DIR_IN)
1112 #define usb_sndisocpipe(dev,endpoint)    ((PIPE_ISOCHRONOUS << 30) |
__create_pipe(dev,endpoint))
1113 #define usb_rcvisocpipe(dev,endpoint)    ((PIPE_ISOCHRONOUS << 30) |
__create_pipe(dev,endpoint) | USB_DIR_IN)
1114 #define usb_sndbulkpipe(dev,endpoint)    ((PIPE_BULK << 30) |
__create_pipe(dev,endpoint))
1115 #define usb_rcvbulkpipe(dev,endpoint)    ((PIPE_BULK << 30) |
__create_pipe(dev,endpoint) | USB_DIR_IN)
1116 #define usb_sndintpipe(dev,endpoint)     ((PIPE_INTERRUPT << 30) |
__create_pipe(dev,endpoint))
1117 #define usb_rcvintpipe(dev,endpoint)     ((PIPE_INTERRUPT << 30) |
__create_pipe(dev,endpoint) | USB_DIR_IN)
```

先看 1110 行,把这个宏展开,PIPE_CONTROL 也是宏,也是定义于同一文件中,include/linux/usb.h,

```
1040 /* ------------------------------------------------------------------------- */
1041
1042 /*
1043  * Calling this entity a "pipe" is glorifying it. A USB pipe
1044  * is something embarrassingly simple: it basically consists
1045  * of the following information:
1046  *  - device number (7 bits)
1047  *  - endpoint number (4 bits)
1048  *  - current Data0/1 state (1 bit) [Historical; now gone]
1049  *  - direction (1 bit)
1050  *  - speed (1 bit) [Historical and specific to USB 1.1; now gone.]
1051  *  - max packet size (2 bits: 8, 16, 32 or 64) [Historical; now gone.]
1052  *  - pipe type (2 bits: control, interrupt, bulk, isochronous)
1053  *
1054  * That's 18 bits. Really. Nothing more. And the USB people have
1055  * documented these eighteen bits as some kind of glorious
1056  * virtual data structure.
1057  *
1058  * Let's not fall in that trap. We'll just encode it as a simple
1059  * unsigned int. The encoding is:
1060  *
1061  *  - max size:           bits 0-1          [Historical; now gone.]
1062  *  - direction:          bit 7             (0 = Host-to-Device [Out],
1063  *                                           1 = Device-to-Host [In] ...
1064  *                                          like endpoint bEndpointAddress)
```

```
1065  *   - device:            bits 8-14        ... bit positions known to uhci-hcd
1066  *   - endpoint:          bits 15-18       ... bit positions known to uhci-hcd
1067  *   - Data0/1:           bit 19           [Historical; now gone. ]
1068  *   - lowspeed:          bit 26           [Historical; now gone. ]
1069  *   - pipe type:         bits 30-31       (00 = isochronous, 01 = interrupt,
1070  *                                          10 = control, 11 = bulk)
1071  *
1072  * Why? Because it's arbitrary, and whatever encoding we select is really
1073  * up to us. This one happens to share a lot of bit positions with the UHCI
1074  * specification, so that much of the uhci driver can just mask the bits
1075  * appropriately.
1076  */
1077
1078 /* NOTE:   these are not the standard USB_ENDPOINT_XFER_* values!! */
1079 #define PIPE_ISOCHRONOUS                    0
1080 #define PIPE_INTERRUPT                      1
1081 #define PIPE_CONTROL                        2
1082 #define PIPE_BULK                           3
```

咱们知道 usb 有四种传输方式,等时传输,中断传输,控制传输,批量传输.一个设备能支持这四种传输中的哪一种或者哪几种是设备本身的属性,在硬件设计的时候就确定了,比如一个纯粹的 u 盘,她肯定是支持 Bulk 传输和 Control 传输的.不同的传输要求有不同的端点,所以对于 u 盘来说,她一定会有 Bulk 端点和 Control 端点,于是就得使用相应的 pipe 来跟不同的端点联系.在这里我们看到了四个宏,其中,PIPE_ISOCHRONOUS 就是标志等时通道,PIPE_INTERRUPT 就是中断通道,PIPE_CONTROL 就是控制通道,PIPE_BULK 就是 BULK 通道.

另外__create_pipe 也是一个宏,由上面她的定义可以看出她为构造一个宏提供了设备号和端点号.在内核里使用一个 unsigned int 类型的变量来表征一个 pipe,其中 8~14 位是设备号,即 devnum,15~18 位是端点号,即 endpoint.而咱们还看到有这么一个宏,USB_DIR_IN,她是用来在 pipe 里面标志数据传输方向的,一个管道要么是只能输入要么是只能输出,鱼和熊掌不可兼得也.在 include/linux/usb_ch9.h 中有的:

```
24
25 /*
26  * USB directions
27  *
28  * This bit flag is used in endpoint descriptors' bEndpointAddress field.
29  * It's also one of three fields in control requests bRequestType.
30  */
31 #define USB_DIR_OUT                     0               /* to device */
32 #define USB_DIR_IN                      0x80            /* to host */
```

在 pipe 里面,第 7 位(bit 7)是表征方向的.所以这里 0x80 也就是说让 bit 7 为 1,这就表示传输方向是由设备向主机的,也就是所谓的 in,而如果这一位是 0,就表示传输方向是由主机向设备的,也就是所谓的 out.而正是因为 USB_DIR_OUT 是 0,而 USB_DIR_IN 是 1,所以我们看到定义管道的时候只有用到了 USB_DIR_IN,而没有用到 USB_DIR_OUT,因为她是 0,任何数和 0 相或都没有意义.

这样,咱们就知道了,get_pipes 函数里 741,742 行就是为 us 的控制输入和控制输出管道赋了值,管道是单向的,但是有一个例外,那就是控制端点,控制端点是双向的,比如你 36 号楼 201 这个端点既可以是往外寄东西,也可以是别人给你寄东西而作为收件人地址.而 usb 规范规定了,每一个 usb 设备至少得有一个控制端点,其端点号为 0.其它端点有没有得看具体设备而定,但这个端点是放之四海而皆准的,不管你是什么设备,只要你是 usb 这条道上的,那你就得遵守这么一个规矩,没得商量.所以我们看到 741,742 行里传递的 endpoint 变量值为 0.显然其构造的两个管道就是对应这个 0 号控制端点的.而接下来几行,就是构造 bulk 管道和中断管道(如果有中断端点的话).

对于 bulk 端点和中断端点(如果有的话),在她们的端点描述符里有这么一个字段,bEndpointAddress,这个字段共八位,但是她包含了挺多信息的,比如这个端点是输入端点还是输出端点,比如这个端点的地址,(总线枚举的时候给她分配的),以及这个端点的端点号,不过要取得她的端点号得用一个掩码 USB_ENDPOINT_NUMBER_MASK,让 bEndpointAddress 和 USB_ENDPOINT_NUMBER_MASK 相与就能得到她的端点号.(就好比一份藏头诗,你得按着特定的方法才能读懂她,而这里特定的方法就是和 USB_ENDPOINT_NUMBER_MASK 这个掩码相与就行了.)

750 行,对于中断端点,您还得使用端点描述符中的 bInterval 字段,表示端点的中断请求间隔时间.

至此,get_pipes 函数结束了,信息都保存到了 us 里面.下面 us 该发挥她的作用了.回到 storage_probe()函数,998 行,把 us 作为参数传递给了 usb_stor_acquire_resources()函数.而这个函数才是故事的高潮.每一个有识之士在看过这个函数之后就会豁然开朗,都会感慨,一下子就看到了春天,看到了光明,原来 Linux 中的设备驱动程序就是这么工作的啊!

```
 996
 997         /* Acquire all the other resources and add the host */
 998         result = usb_stor_acquire_resources(us);
 999         if (result)
1000                 goto BadDevice;
1001         result = scsi_add_host(us->host, &intf->dev);
1002         if (result) {
1003                 printk(KERN_WARNING USB_STORAGE
1004                         "Unable to add the scsi host\n");
1005                 goto BadDevice;
1006         }
```

我们来看 usb_stor_acquire_resources 函数.它被定义于 drivers/usb/storage/usb.c 中:

```
 755 /* Initialize all the dynamic resources we need */
 756 static int usb_stor_acquire_resources(struct us_data *us)
 757 {
 758         int p;
 759
 760         us->current_urb = usb_alloc_urb(0, GFP_KERNEL);
 761         if (!us->current_urb) {
 762                 US_DEBUGP("URB allocation failed\n");
 763                 return -ENOMEM;
```

```
764         }
765
766         /* Lock the device while we carry out the next two operations */
767         down(&us->dev_semaphore);
768
769         /* For bulk-only devices, determine the max LUN value */
770         if (us->protocol == US_PR_BULK) {
771                 p = usb_stor_Bulk_max_lun(us);
772                 if (p < 0) {
773                         up(&us->dev_semaphore);
774                         return p;
775                 }
776                 us->max_lun = p;
777         }
778
779         /* Just before we start our control thread, initialize
780          * the device if it needs initialization */
781         if (us->unusual_dev->initFunction)
782                 us->unusual_dev->initFunction(us);
783
784         up(&us->dev_semaphore);
785
786         /*
787          * Since this is a new device, we need to register a SCSI
788          * host definition with the higher SCSI layers.
789          */
790         us->host = scsi_host_alloc(&usb_stor_host_template, sizeof(us));
791         if (!us->host) {
792                 printk(KERN_WARNING USB_STORAGE
793                         "Unable to allocate the scsi host\n");
794                 return -EBUSY;
795         }
796
797         /* Set the hostdata to prepare for scanning */
798         us->host->hostdata[0] = (unsigned long) us;
799
800         /* Start up our control thread */
801         p = kernel_thread(usb_stor_control_thread, us, CLONE_VM);
802         if (p < 0) {
803                 printk(KERN_WARNING USB_STORAGE
804                        "Unable to start control thread\n");
805                 return p;
806         }
807         us->pid = p;
```

```
808
809         /* Wait for the thread to start */
810         wait_for_completion(&(us->notify));
811
812         return 0;
813 }
```

待到山花浪漫时,她在丛中笑.一个悟性高的人应该一眼就能从这个函数中找出那行在丛中笑的代码来,没错,她就是 801 行,kernel_thread,这个函数造就了许多的经典的 Linux 内核模块,正是因为她的存在,Linux 中某些设备驱动程序的编写变得非常简单.可以说,对某些设备驱动程序来说,kernel_thread 几乎是整个 driver 的灵魂,或者说是该 Linux 内核模块的灵魂.不管她隐藏的多么深,她总像漆黑中的萤火虫,那样的鲜明,那样的出众.甚至不夸张的说,对于很多模块来说,只要找到 kernel_thread 这一行,基本上你就知道这个模块是怎么工作的了.

传说中的 URB

有人问,怎么写个驱动写这么久啊?有完没完啊?此水何时休?此恨何时已?

的确,一路走来,大家都不容易,但既然已经走到今天,我们能做的也只有是坚持下去.十年之前,我不认识你,你不属于我,但十年之后我依然记得那一年(1997 年),我的一个中学校友的那一句:不管前面是地雷阵还是万丈深渊,我(们)都将一往无前,义无反顾,鞠躬尽瘁,死而后已.这个人叫朱镕基,毕业于长沙市一中.

usb_stor_acquire_resources,从名字上来看,获取资源.什么资源?网名为爬上墙头等红杏的朋友不禁表示出了一些好奇,之前不是申请了一大堆内存了吗?写个 usb 设备驱动程序怎么那么麻烦啊?不是专门为 usb mass storage 设备准备了一个 struct us_data 这么一个结构体了吗?不是说故事已经到高潮了吗?

周润发说得好,我还刚上路呢.没错,如果你以为看到这里你已经对 usb 设备驱动程序有了足够的认识,认为接下来的代码已经没有必要再分析了,那么,我只想说,上帝创造世界的计划中,未必包括使你会写 usb 设备驱动程序.

的确,别看 usb_stor_acquire_resources 的代码不多,每一行都有每一行的故事.本节我们只讲其中的一行代码,没错,就是一行代码,因为我们需要隆重推出一个名词,一个响当当的名字,她就是传说中的 urb,全称 usb request block.usb 设备需要通信,要传递数据,就需要使用 urb,确切的说,应该是 usb 设备驱动程序使用 urb.实际上,作为 usb 设备驱动,它本身并不能直接操纵数据的传输,在 usb 这个大观园里,外接设备永远都是配角,真正的核心只是 usb core,而真正负责调度的是 usb host controller,这个您通常看不见的 usb 主机控制器芯片,他俨然是 usb 大观园中的大管家.设备驱动要发送信息,所需要做的是建立一个 urb 数据结构,并把这个数据结构交给核心层,而核心层会为所有设备统一完成调度,而设备在提交了 urb 之后需要做的,只是等待,等待,在那漫漫长夜中等待.别急,我们慢慢来.

760 行,一条赋值语句,等号左边,us->current_urb,等号右边,usb_alloc_urb 函数被调用.如果说 struct us_data 是 usb mass storage 中的主角,那么 struct urb 将毫无争议的成为整个 usb 子系统中的主角. Linux 中所有的 usb 设备驱动,都必然也必须要使用 urb.那么 urb 究竟长成什么样呢?她是如芙蓉姐姐那么婀娜多姿呢,又亦或是如林志玲那般无公害性感呢?在 include/linux/usb.h 中能找到她的靓照:

```
614 /**
615  * struct urb - USB Request Block
616  * @urb_list: For use by current owner of the URB.
617  * @pipe: Holds endpoint number, direction, type, and more.
618  *      Create these values with the eight macros available;
619  *      usb_{snd,rcv}TYPEpipe(dev,endpoint), where the TYPE is "ctrl"
620  *      (control), "bulk", "int" (interrupt), or "iso" (isochronous).
621  *      For example usb_sndbulkpipe() or usb_rcvintpipe().  Endpoint
622  *      numbers range from zero to fifteen.  Note that "in" endpoint two
623  *      is a different endpoint (and pipe) from "out" endpoint two.
624  *      The current configuration controls the existence, type, and
625  *      maximum packet size of any given endpoint.
626  * @dev: Identifies the USB device to perform the request.
627  * @status: This is read in non-iso completion functions to get the
628  *      status of the particular request.  ISO requests only use it
629  *      to tell whether the URB was unlinked; detailed status for
630  *      each frame is in the fields of the iso_frame-desc.
631  * @transfer_flags: A variety of flags may be used to affect how URB
632  *      submission, unlinking, or operation are handled.  Different
633  *      kinds of URB can use different flags.
634  * @transfer_buffer:  This identifies the buffer to (or from) which
635  *      the I/O request will be performed (unless URB_NO_TRANSFER_DMA_MAP
636  *      is set).  This buffer must be suitable for DMA; allocate it with
637  *      kmalloc() or equivalent.  For transfers to "in" endpoints, contents
638  *      of this buffer will be modified.  This buffer is used for the data
639  *      stage of control transfers.
640  * @transfer_dma: When transfer_flags includes URB_NO_TRANSFER_DMA_MAP,
641  *      the device driver is saying that it provided this DMA address,
642  *      which the host controller driver should use in preference to the
643  *      transfer_buffer.
644  * @transfer_buffer_length: How big is transfer_buffer.  The transfer may
645  *      be broken up into chunks according to the current maximum packet
646  *      size for the endpoint, which is a function of the configuration
647  *      and is encoded in the pipe.  When the length is zero, neither
648  *      transfer_buffer nor transfer_dma is used.
649  * @actual_length: This is read in non-iso completion functions, and
650  *      it tells how many bytes (out of transfer_buffer_length) were
651  *      transferred.  It will normally be the same as requested, unless
652  *      either an error was reported or a short read was performed.
653  *      The URB_SHORT_NOT_OK transfer flag may be used to make such
654  *      short reads be reported as errors.
655  * @setup_packet: Only used for control transfers, this points to eight bytes
656  *      of setup data.  Control transfers always start by sending this data
657  *      to the device.  Then transfer_buffer is read or written, if needed.
```

```
658  * @setup_dma: For control transfers with URB_NO_SETUP_DMA_MAP set, the
659  *      device driver has provided this DMA address for the setup packet.
660  *      The host controller driver should use this in preference to
661  *      setup_packet.
662  * @start_frame: Returns the initial frame for isochronous transfers.
663  * @number_of_packets: Lists the number of ISO transfer buffers.
664  * @interval: Specifies the polling interval for interrupt or isochronous
665  *      transfers.  The units are frames (milliseconds) for for full and low
666  *      speed devices, and microframes (1/8 millisecond) for highspeed ones.
667  * @error_count: Returns the number of ISO transfers that reported errors.
668  * @context: For use in completion functions.  This normally points to
669  *      request-specific driver context.
670  * @complete: Completion handler. This URB is passed as the parameter to the
671  *      completion function.  The completion function may then do what
672  *      it likes with the URB, including resubmitting or freeing it.
673  * @iso_frame_desc: Used to provide arrays of ISO transfer buffers and to
674  *      collect the transfer status for each buffer.
675  *
676  * This structure identifies USB transfer requests.  URBs must be allocated by
677  * calling usb_alloc_urb() and freed with a call to usb_free_urb().
678  * Initialization may be done using various usb_fill_*_urb() functions.  URBs
679  * are submitted using usb_submit_urb(), and pending requests may be canceled
680  * using usb_unlink_urb() or usb_kill_urb().
681  *
682  * Data Transfer Buffers:
683  *
684  * Normally drivers provide I/O buffers allocated with kmalloc() or otherwise
685  * taken from the general page pool.  That is provided by transfer_buffer
686  * (control requests also use setup_packet), and host controller drivers
687  * perform a dma mapping (and unmapping) for each buffer transferred.  Those
688  * mapping operations can be expensive on some platforms (perhaps using a dma
689  * bounce buffer or talking to an IOMMU),
690  * although they're cheap on commodity x86 and ppc hardware.
691  *
692  * Alternatively, drivers may pass the URB_NO_xxx_DMA_MAP transfer flags,
693  * which tell the host controller driver that no such mapping is needed since
694  * the device driver is DMA-aware.  For example, a device driver might
695  * allocate a DMA buffer with usb_buffer_alloc() or call usb_buffer_map().
696  * When these transfer flags are provided, host controller drivers will
697  * attempt to use the dma addresses found in the transfer_dma and/or
698  * setup_dma fields rather than determining a dma address themselves.  (Note
699  * that transfer_buffer and setup_packet must still be set because not all
700  * host controllers use DMA, nor do virtual root hubs).
701  *
```

```
 * Initialization:
 *
 * All URBs submitted must initialize the dev, pipe, transfer_flags (may be
 * zero), and complete fields.
 * The URB_ASYNC_UNLINK transfer flag affects later invocations of
 * the usb_unlink_urb() routine.  Note: Failure to set URB_ASYNC_UNLINK
 * with usb_unlink_urb() is deprecated.  For synchronous unlinks use
 * usb_kill_urb() instead.
 *
 * All URBs must also initialize
 * transfer_buffer and transfer_buffer_length.  They may provide the
 * URB_SHORT_NOT_OK transfer flag, indicating that short reads are
 * to be treated as errors; that flag is invalid for write requests.
 *
 * Bulk URBs may
 * use the URB_ZERO_PACKET transfer flag, indicating that bulk OUT transfers
 * should always terminate with a short packet, even if it means adding an
 * extra zero length packet.
 *
 * Control URBs must provide a setup_packet.  The setup_packet and
 * transfer_buffer may each be mapped for DMA or not, independently of
 * the other.  The transfer_flags bits URB_NO_TRANSFER_DMA_MAP and
 * URB_NO_SETUP_DMA_MAP indicate which buffers have already been mapped.
 * URB_NO_SETUP_DMA_MAP is ignored for non-control URBs.
 *
 * Interrupt URBs must provide an interval, saying how often (in milliseconds
 * or, for highspeed devices, 125 microsecond units)
 * to poll for transfers.  After the URB has been submitted, the interval
 * field reflects how the transfer was actually scheduled.
 * The polling interval may be more frequent than requested.
 * For example, some controllers have a maximum interval of 32 microseconds,
 * while others support intervals of up to 1024 microseconds.
 * Isochronous URBs also have transfer intervals.  (Note that for isochronous
 * endpoints, as well as high speed interrupt endpoints, the encoding of
 * the transfer interval in the endpoint descriptor is logarithmic.
 * Device drivers must convert that value to linear units themselves.)
 *
 * Isochronous URBs normally use the URB_ISO_ASAP transfer flag, telling
 * the host controller to schedule the transfer as soon as bandwidth
 * utilization allows, and then set start_frame to reflect the actual frame
 * selected during submission.  Otherwise drivers must specify the start_frame
 * and handle the case where the transfer can't begin then.  However, drivers
 * won't know how bandwidth is currently allocated, and while they can
 * find the current frame using usb_get_current_frame_number () they can't
```

```
746  * know the range for that frame number.  (Ranges for frame counter values
747  * are HC-specific, and can go from 256 to 65536 frames from "now".)
748  *
749  * Isochronous URBs have a different data transfer model, in part because
750  * the quality of service is only "best effort".  Callers provide specially
751  * allocated URBs, with number_of_packets worth of iso_frame_desc structures
752  * at the end.  Each such packet is an individual ISO transfer.  Isochronous
753  * URBs are normally queued, submitted by drivers to arrange that
754  * transfers are at least double buffered, and then explicitly resubmitted
755  * in completion handlers, so
756  * that data (such as audio or video) streams at as constant a rate as the
757  * host controller scheduler can support.
758  *
759  * Completion Callbacks:
760  *
761  * The completion callback is made in_interrupt(), and one of the first
762  * things that a completion handler should do is check the status field.
763  * The status field is provided for all URBs.  It is used to report
764  * unlinked URBs, and status for all non-ISO transfers.  It should not
765  * be examined before the URB is returned to the completion handler.
766  *
767  * The context field is normally used to link URBs back to the relevant
768  * driver or request state.
769  *
770  * When the completion callback is invoked for non-isochronous URBs, the
771  * actual_length field tells how many bytes were transferred.  This field
772  * is updated even when the URB terminated with an error or was unlinked.
773  *
774  * ISO transfer status is reported in the status and actual_length fields
775  * of the iso_frame_desc array, and the number of errors is reported in
776  * error_count.  Completion callbacks for ISO transfers will normally
777  * (re)submit URBs to ensure a constant transfer rate.
778  */
779 struct urb
780 {
781         /* private, usb core and host controller only fields in the urb */
782         struct kref kref;               /* reference count of the URB */
783         spinlock_t lock;                /* lock for the URB */
784         void *hcpriv;                   /* private data for host controller */
785         struct list_head urb_list;      /* list pointer to all active urbs */
786         int bandwidth;                  /* bandwidth for INT/ISO request */
787         atomic_t use_count;             /* concurrent submissions counter */
788         u8 reject;                      /* submissions will fail */
789
```

```
790         /* public, documented fields in the urb that can be used by drivers */
791         struct usb_device *dev;           /* (in) pointer to associated device */
792         unsigned int pipe;                /* (in) pipe information */
793         int status;                       /* (return) non-ISO status */
794         unsigned int transfer_flags;      /* (in) URB_SHORT_NOT_OK | ...*/
795         void *transfer_buffer;            /* (in) associated data buffer */
796         dma_addr_t transfer_dma;          /* (in) dma addr for transfer_buffer */
797         int transfer_buffer_length;       /* (in) data buffer length */
798         int actual_length;                /* (return) actual transfer length */
799         unsigned char *setup_packet;      /* (in) setup packet (control only) */
800         dma_addr_t setup_dma;             /* (in) dma addr for setup_packet */
801         int start_frame;                  /* (modify) start frame (ISO) */
802         int number_of_packets;            /* (in) number of ISO packets */
803         int interval;                     /* (modify) transfer interval (INT/ISO) */
804         int error_count;                  /* (return) number of ISO errors */
805         void *context;                    /* (in) context for completion */
806         usb_complete_t complete;          /* (in) completion routine */
807         struct usb_iso_packet_descriptor iso_frame_desc[0];       /* (in) ISO ONLY
*/
808 };
```

我们常常抱怨,Linux 内核源代码中注释太少了,以至于我们常常看不懂那些代码究竟是什么含义.但,urb 让开发人员做足了文章,结构体 struct urb 的定义不过 30 行,而说明文字却用了足足 160 余行.可见 urb 的江湖地位.有时候我真的怀疑这些写代码的人是不是以为自己的代码是按行计费的,这么一大串的注释他写得不累我们看得还累呢.当然我们这里贴出来主要还是为了保持原汁原味,另一方面这个注释也说得很清楚,对于每一个成员都解释了,而我们接下来将必然要用到 urb 的许多个成员.

此刻,我们暂时不去理会这个结构体每一个元素的作用,只需要知道,一个 urb 包含了执行 usb 传输所需要的所有信息.而作为驱动程序,要通信,就必须创建这么一个数据结构,并且为她赋好值,显然不同类型的传输,需要对 urb 赋不同的值,然后将她提交给底层,完了底层的 usb core 会找到相应的 usb host controller,从而具体去实现数据的传输,传输完了之后,usb host controller 会通知设备驱动程序.

总之我们知道,760 行就是调用 usb_alloc_urb()申请了一个 struct urb 结构体.关于 usb_alloc_urb()这个函数,我们不打算讲,它是 usb core 所提供的一个函数,来自 drivers/usb/core/urb.c,usb 开发人员的确是给足了 urb 的面子,专门把和这个数据结构相关的代码整理在这么一个文件中了.我们可以在 include/linux/usb.h 中找到这个函数的声明,

```
917 extern struct urb *usb_alloc_urb(int iso_packets, int mem_flags);
```

这个函数的作用很明显,就是为一个 urb 结构体申请内存,它有两个参数,其中第一个 iso_packets 用来在等时传输的方式下指定你需要传输多少个包,对于非等时模式来说,这个参数直接使用 0.另一个参数 mem_flags 就是一个 flag,表征申请内存的方式,这个 flag 将最终传递给 kmalloc 函数,我们这里传递的是 GFP_KERNEL,这个 flag 是内存申请中最常用的,我们之前也用过,在为 us 申请内存的时候.usb_alloc_urb 最终将返回一个 urb 指针,而 us 的成员 current_urb 也是一个 struct urb 的指针,所以就赋给它了,不过需要记住,usb_alloc_urb 除了申请内存以外,还对结构体作了初始化,结构体 urb 被初

始化为 0,虽然这里我们没有把这个函数的代码贴出来,但你也千万不要以为写代码的人跟你我似的,申请变量还能忘了初始化.同时,struct urb 中还有一个引用计数,以及一个自旋锁,这些也同样被初始化了.

所以,接下来我们就将和 us->current_urb 打交道了.如果你对 urb 究竟怎么用还有些困惑的话,可以看看 host controller 的代码,如果你不想看,那么我可以用一种你最能接受的方式告诉你,usb 是一种总线,是总线它就要通信,我们现实生活中真正要使用的是设备,但是光有设备还不足以实现 usb 通信,于是世界上有了 usb host controller(usb 主机控制器),它来负责统一调度,这就好比北京的交警,北京这个城市里真正需要的本来是车辆和行人,而光有车辆和行人,没有交警,那么这个城市里的车辆和行人必将乱套,于是诞生了交警这么一个行业,交警站在路口统一来管理调度那混乱的交通.假如车辆和行人可以完全自觉遵守某种规矩而来来往往于这个城市的每一个角落,每一个路口,那么交警就没有必要存在了.同样,假如设备能够完全自觉的传递信息,每一个数据包都能到达它应该去的地方,那么我们根本就不需要有主机控制器这么一个东西.然而,事实是,在北京,遵守交通规则的人不多,我相信每一个来过北京的人都会发现,在北京,闯红灯是一种时尚.我常常对自己说,如果哪一天我没有闯红灯,那么由此可以推导出来,这一天我一定没有出门.同样,在 usb 的世界中,设备也总是那么的不守规矩,我们必须要设计一个东西出来管理来控制所有的 usb 设备的通信,这样,主机控制器就横空出世了.

那么设备和主机控制器的分工又是如何呢?硬件实现上我们就不说了,说点具体的,Linux 中,设备驱动程序只要为每一次请求准备一个 urb 结构体变量,把它填充好,(就是说赋上该赋的值)然后它调用 usb core 提供的函数,把这个 urb 传递给 host controller,host controller 就会把各个设备驱动程序所提交的 urb 统一规划,去执行每一个操作.而这期间,usb 设备驱动程序通常会进入睡眠,而一旦 host controller 把 urb 要做的事情给做完了,它会调用一个函数去唤醒 usb 设备驱动程序,然后 usb 设备驱动程序就可以继续往下走了.这又好比我们学校里的师生关系.考试的时候,我们只管把试卷填好,然后我们交给老师,然后老师拿去批改试卷,这期间我们除了等待别无选择,等待老师改完了试卷,告诉了我们分数,我们又继续我们的生活.当然了,如果成绩不好,也许有些人的生活就不会继续了,在复旦,我们可以选择去曾经的绝情谷上吊,可以选择去第四教学楼顶层跳下来,在交大,则可以去跳思源湖.同样,usb 设备驱动程序也是如此,如果 urb 提交给 usb host 了,但是最终却没有成功执行,那么也许该 usb 设备驱动程序的生命也就提前结束.不过这都是后话,现在只要有个感性认识即可,稍后看到了就能更深刻的体会了,这种岗位分工的方式给我们编写设备驱动程序带来了巨大的方便.

心锁

如果大家没意见的话,我们继续 usb_stor_acquire_resources 函数.

761 至 764 行,这没啥好说的吧.就是刚才 urb 申请了之后判断是否申请成功了,如果指针为 NULL 那么就是失败了.直接返回-ENOMEM.别往下了.

767 行,哦,又一个家伙闪亮登场了,dev_semaphore,这是一个信号量,在 storage_probe 的最初始阶段我们曾经见过,当时有这么一句话,这就是调用一个宏 init_MUTEX 来初始化一个信号量,

```
943         init_MUTEX(&(us->dev_semaphore));
```

我们当时说了,等到用的时候再讲.不过现在的确是到用的时候了,不过,我还不想讲.曾经我天真的以为,只要学了谭浩强的那本 C 程序设计,即便不能写代码,也应该能够看懂代码.然而,后来我发现事实并非如此,世界没错,我错了.

首先,什么是信号量?毕业的时候,校园招聘中常常笔试面试会考信号量,会考死锁,不过通常考的会比较简单,更多的情况是考察一些基本概念,印象中曾经在 Sun 中国工程研究院的面试中被问起过,当然也有比较复杂一点的,要求结合具体的问题对某种算法进行合理性分析,比如 Intel 某年在上海交大的笔试题,考到了经典的哲学家进餐问题,不过幸运的是这种变态的笔试题我没遇上,我进 Intel 的时候只被问起解释冒泡排序法...

我们整个故事中有两个信号量,它们都是 us 的成员,一个是这个 dev_semaphore,另一个是 sema,在定义 struct us_data 的时候我们已经看到过,

```
111         struct semaphore        dev_semaphore;    /* protect pusb_dev */
156         struct semaphore        sema;             /* to sleep thread on   */
```

sema 也同样是在 storage_probe 的一开始就做了初始化,

```
944         init_MUTEX_LOCKED(&(us->sema));
```

设备驱动中最难的部分在于三个方面,一个是涉及到内存管理的代码,一个是涉及到进程管理的代码,另一个就是信号量和互斥锁或者别的锁的代码.这些部分如果不合理将容易导致系统崩溃,而信号量最容易导致的就是死锁.

网名为卖血上网的哥们说话了,那么到底什么是信号量?或者什么是互斥锁?

先说互斥锁.它诞生于这样一个背景.这个世界上,有些东西只能属于某一个人,或者说在一个时间里只能属于一个人,这你承认吧,比如一个女孩的心.当你要追求一个女孩时,你首先会去了解其人是否名花有主,若是否,你才会去追求,若已然有主,那么你只能放弃,或者准确的说,你只能等待.当然你可以很激昂的说,如果等待可以换来奇迹的话,我宁愿等下去,哪怕一年,抑或一生!然而,她爱的是他,终究不是你,所以你伤悲,流泪,却打不开她心中那把锁.这里你应该就能感觉到什么是互斥锁了,一个女孩如果心有所属,那么对你来说,就仿佛已有人在你之前给她上了一把锁,而钥匙,不在你这里.

那么你问代码中为什么要锁?Ok,我告诉你,如果你正在操作一个队列,比如一个队列一百个元素,你想把第七十个读出来,于是你去遍历这个队列,可是如果没有锁,那么可能你遍历的时候别人也可以操作这个队列,比如你马上就要找到第七十个了,可是,可是,注意了,可是,这个时候,说不定哪位哥们儿缺德,把第七十个数给删除了,那你不是白忙活了?所以,怎么办?设一个锁,每个人要想操作这个队列就得先获得这把锁,而一旦你获得了这把锁,你在操作这把锁期间,别人就不能操作,因为他要操作他就得先获得锁,而这个时候锁在你这里,所以他只能等待,等你结束了,释放了锁,他才可以去操作.那么互斥锁指的就是这种情况,一个资源只能同时被一个进程操作,互斥的字面意思也正是如此.互相排斥,就像爱情是自私的一样.

那什么是信号量?信号量和互斥锁略有不同.它允许一个时间里有几个进程同时访问一段资源.到底允许几个可以设定,这称为设定信号量的初值,如设定为 1,那就说明是同一时间只有一个进程可以访问,那么这就是互斥锁了,不过有的时候一个资源确实可以让几个进程访问,特别是读访问,你想一个文件可以被两个进程同时读,这不要紧吧,各读各的,谁也影响不了谁,只要大家都不写就是了.设定信号量的初值,比如设定 5,那么就是说,同一时间你就让最多 5 个进程同时去读这个文件.每个进程获取了信号量就把信号量的值减一,到第六个进程了,它去判断,发现值已经等于 0 了,于是它不能访问了,只能等待,等待别的进程释放锁.

不过也许,一把钥匙只能开一把锁,更能代表爱情的专一.所以实际上,Linux 内核代码中锁用得更多,而初值不为 1 的信号量用得相对来说不多.比如我们刚才看到的这两个信号量,都是属于当作互斥锁在用.因为他们的初值一个被设置成了 1(init_MUTEX),一个被设置成了 0(init_MUTEX_LOCKED).设置成一很好理解,就是一把互斥锁,只能容许一个进程去获得它,设置成 0 呢,就表示这把锁已经被锁住了,得有进程释放它才可以.我们这里 767 行这句 down(&us->dev_semaphore)和 up(&us->dev_semaphore)是一对,一个是获得锁,一个是释放锁.它们就是为了保护中间那段代码.我刚才说我不想讲这段代码,的确,现在讲为什么这里要用锁还为时过早,整个故事中 us->dev_semaphore 出现的次数不是很多,但是我们必须对整个代码都熟悉了才可能理解为什么要用这把锁,因为这些代码都是环环相扣的,不能孤立来看.所以,我们将在故事的收尾阶段来一次性来以高屋建瓴的方式看每一处 down(&us->dev_semaphore)和 up(&us->dev_semaphore)的使用原因.接下来我们看到了这把锁的时候,也将一并跳过不提.到最后再来看.需要说的是这里 down 和 up 这两个函数的作用分别就是去获得锁和释放锁.对于 down 来说,它每次判断一下信号量的值是否大于 0,若是,就进入下面的代码,同时将信号量的值减一,若否,就等待,或者说专业一点,进入睡眠.对于 down(),我们小时候那部<<挥剑问情>>的歌词很形象的描述了其行为,

男:挥剑问情,如果问是有情,也愿以身相许,以身相殉;

女:挥剑问情,如果问是无情,又怕回首别是一种伤心.

不过,我们这里看到了两把锁,除了 us->dev_semaphore 以外,另一个是 us->sema,现在还没有使用它,但是我们可以先说一下,us->sema 在整个故事中出现的次数不多,总共只有三次.加上这里提到的这句初始化为 0 的语句,一个出现了四次.所以我们在遇到它的时候不需要跳过,会详细的讲.因为很容易理解它为什么会用在那里.请你记住,这个信号量或者说这把锁是被初始化为 0 了,不是 1,它一开始就处于锁住的状态,到时候你就会知道为什么了.我们边走边看.

另外一个需要说一下的是,与 down ()相似的有一个叫做 down_interruptible ()的函数,它们的区别在于后者可被信号打断,而前者不可被信号打断,而一旦问是无情,那么他们将进入等待,或者专业一点说,进入睡眠,直到某一天...

我们将看到的获取 us->sema 的函数正是 down_interruptible.而释放锁的函数仍然可以用 up.

第一次亲密接触(一)

直到现在我们才将第一次真正的开始接触 usb 的四种数据传输之一,控制传输.应该说从这一刻开始,代码开始变得复杂了.不过不要怕,有我在.在这个美妙的夏夜,让我们剪一段月光,来解代码的霜.

769 至 777 行,做了一件事情,确定这个设备的 max lun.不要说你不知道什么是 max lun.不知道的回去跪主板吧,我很负责任的向你推荐我们 Intel 最新的 3 系列整合芯片组主板.

在 get_transport()函数中,我们针对各种情况给 us->max_lun 赋了值,但是我们当时就注意到了,唯独对于 Bulk-Only 的设备,当时并没有赋值,所以这里我们看到,对于 us->protocol 等于 US_PR_BULK 的情况,有一个专门的函数来获得这个设备的 max lun.网友女为悦己者_整_容好奇的问,为什么写代码的同志在这里对 Bulk-Only 的设备表现出一种”偏偏喜欢你”的态度呢?没什么特别的,党中央规定的.所有的 usb 设备遵守一个规范,这个规范叫做 usb spec,而 usb 设备分为很多种类,usb mass storage 是其中一类,而 mass storage 设备又分为很多子类,每个子类又有它自己的规范,比如 U 盘它所遵守的就是 usb

mass storage class bulk-only transport spec.而这个规范说得很清楚,对于这种设备,它的 lun 不是凭感觉就能判断的,你得发送一个命令给它,向它去查询,然后它会做出响应,这样你才能知道它究竟是几个 lun.这条命令就是"GET MAX LUN",详见 spec 3.2.

所以我们即使不认真看这个函数也可以知道究竟发生了什么.而且我们之前还说过,普通的 U 盘的 max lun 肯定是 0.对于那种读卡器才可能有多个 lun.不过我们还是不妨来深入的看一下这个函数,毕竟这个函数是在 drivers/usb/storage/transport.c 中,属于我们的辖区.当然更重要的是,此前我们一直没有真正见识过究竟一次 usb 传输是怎么回事,作为 usb 设备驱动程序究竟如何发起 usb 传输,而这个函数正好给了我们一次不错的机会.同时我们也将会知道,了解了一次控制传输之后,别的传输也会很容易理解.

首先我们知道这次传输的目的是为了获得一个数字,max lun,而获得这个数据的方式是发送命令,所以整个过程就是,你发送一个命令给设备,设备返回一个值给你,这么简单的传输叫什么?控制传输.很显然,地球人都知道,控制传输是 usb 四种传输方式中最简单的那种.来看具体代码:

```
908 /* Determine what the maximum LUN supported is */
909 int usb_stor_Bulk_max_lun(struct us_data *us)
910 {
911         int result;
912
913         /* issue the command */
914         result = usb_stor_control_msg(us, us->recv_ctrl_pipe,
915                                 US_BULK_GET_MAX_LUN,
916                                 USB_DIR_IN | USB_TYPE_CLASS |
917                                 USB_RECIP_INTERFACE,
918                                 0, us->ifnum, us->iobuf, 1, HZ);
919
920         US_DEBUGP("GetMaxLUN command result is %d, data is %d\n",
921                   result, us->iobuf[0]);
922
923         /* if we have a successful request, return the result */
924         if (result > 0)
925                 return us->iobuf[0];
926
927         /*
928          * Some devices (i.e. Iomega Zip100) need this -- apparently
929          * the bulk pipes get STALLed when the GetMaxLUN request is
930          * processed.   This is, in theory, harmless to all other devices
931          * (regardless of if they stall or not).
932          */
933         if (result == -EPIPE) {
934                 usb_stor_clear_halt(us, us->recv_bulk_pipe);
935                 usb_stor_clear_halt(us, us->send_bulk_pipe);
936         }
937
938         /*
```

```
939         * Some devices don't like GetMaxLUN.  They may STALL the control
940         * pipe, they may return a zero-length result, they may do nothing at
941         * all and timeout, or they may fail in even more bizarrely creative
942         * ways.  In these cases the best approach is to use the default
943         * value: only one LUN.
944         */
945        return 0;
946 }
```

代码不长,不过并不容易.首先 914 行, usb_stor_control_msg()函数被调用,这也是我们自己定义的函数,所以也得讲,同样来自 drivers/usb/storage/transport.c:

```
209
210 /*
211  * Transfer one control message, with timeouts, and allowing early
212  * termination.  Return codes are usual -Exxx, *not* USB_STOR_XFER_xxx.
213  */
214 int usb_stor_control_msg(struct us_data *us, unsigned int pipe,
215             u8 request, u8 requesttype, u16 value, u16 index,
216             void *data, u16 size, int timeout)
217 {
218        int status;
219
220        US_DEBUGP("%s: rq=%02x rqtype=%02x value=%04x index=%02x
len=%u\n",
221                  __FUNCTION__, request, requesttype,
222                  value, index, size);
223
224        /* fill in the devrequest structure */
225        us->cr->bRequestType = requesttype;
226        us->cr->bRequest = request;
227        us->cr->wValue = cpu_to_le16(value);
228        us->cr->wIndex = cpu_to_le16(index);
229        us->cr->wLength = cpu_to_le16(size);
230
231        /* fill and submit the URB */
232        usb_fill_control_urb(us->current_urb, us->pusb_dev, pipe,
233                  (unsigned char*) us->cr, data, size,
234                  usb_stor_blocking_completion, NULL);
235        status = usb_stor_msg_common(us, timeout);
236
237        /* return the actual length of the data transferred if no error */
238        if (status == 0)
239             status = us->current_urb->actual_length;
```

```
240         return status;
241 }
```

这里相对麻烦一点的函数是 usb_stor_msg_common,仍然是我们自己定义的函数,所以我们又得继续往下一层看,这么多层函数调用的确挺让人看了头晕,一个人总要走陌生的路,看陌生的风景,听陌生的歌,然后在某个不经意的瞬间,你会发现,原本费尽心机想要弄清楚的函数就这么把自己弄糊涂了.然而,就像世上的每一条路都是弯的一样,每一个模块都会有曲折的函数调用,写代码的哥们儿为了表现自己一流的编剧水平,永远不会让我们一路平稳的看完整个模块的.除了面对,我们别无选择,毕竟从我们年幼时,摇篮就告诉我们,人生是不平静的,也是动荡的,所以为何不微笑着面对这些麻烦?这个函数仍旧是来自 drivers/usb/storage/transport.c:

```
132 /* This is the common part of the URB message submission code
133  *
134  * All URBs from the usb-storage driver involved in handling a queued scsi
135  * command _must_ pass through this function (or something like it) for the
136  * abort mechanisms to work properly.
137  */
138 static int usb_stor_msg_common(struct us_data *us, int timeout)
139 {
140         struct completion urb_done;
141         struct timer_list to_timer;
142         int status;
143
144         /* don't submit URBs during abort/disconnect processing */
145         if (us->flags & ABORTING_OR_DISCONNECTING)
146                 return -EIO;
147
148         /* set up data structures for the wakeup system */
149         init_completion(&urb_done);
150
151         /* fill the common fields in the URB */
152         us->current_urb->context = &urb_done;
153         us->current_urb->actual_length = 0;
154         us->current_urb->error_count = 0;
155         us->current_urb->status = 0;
156
157         /* we assume that if transfer_buffer isn't us->iobuf then it
158          * hasn't been mapped for DMA.  Yes, this is clunky, but it's
159          * easier than always having the caller tell us whether the
160          * transfer buffer has already been mapped. */
161         us->current_urb->transfer_flags =
162                         URB_ASYNC_UNLINK | URB_NO_SETUP_DMA_MAP;
163         if (us->current_urb->transfer_buffer == us->iobuf)
164                 us->current_urb->transfer_flags |= URB_NO_TRANSFER_DMA_MAP;
165         us->current_urb->transfer_dma = us->iobuf_dma;
```

```
166       us->current_urb->setup_dma = us->cr_dma;
167
168       /* submit the URB */
169       status = usb_submit_urb(us->current_urb, GFP_NOIO);
170       if (status) {
171             /* something went wrong */
172             return status;
173       }
174
175       /* since the URB has been submitted successfully, it's now okay
176        * to cancel it */
177       set_bit(US_FLIDX_URB_ACTIVE, &us->flags);
178
179       /* did an abort/disconnect occur during the submission? */
180       if (us->flags & ABORTING_OR_DISCONNECTING) {
181
182             /* cancel the URB, if it hasn't been cancelled already */
183             if (test_and_clear_bit(US_FLIDX_URB_ACTIVE, &us->flags)) {
184                   US_DEBUGP("-- cancelling URB\n");
185                   usb_unlink_urb(us->current_urb);
186             }
187       }
188
189       /* submit the timeout timer, if a timeout was requested */
190       if (timeout > 0) {
191             init_timer(&to_timer);
192             to_timer.expires = jiffies + timeout;
193             to_timer.function = timeout_handler;
194             to_timer.data = (unsigned long) us;
195             add_timer(&to_timer);
196       }
197
198       /* wait for the completion of the URB */
199       wait_for_completion(&urb_done);
200       clear_bit(US_FLIDX_URB_ACTIVE, &us->flags);
201
202       /* clean up the timeout timer */
203       if (timeout > 0)
204             del_timer_sync(&to_timer);
205
206       /* return the URB status */
207       return us->current_urb->status;
208 }
```

好了,代码贴完了,现在让我们一点一点去认识这段代码.记住我们最终就是为了看清楚 usb_stor_Bulk_max_lun()这个函数究竟是怎么获得 max lun 的.

第一次亲密接触(二)

对于设备驱动程序而言,控制传输要做的事情很简单,向 usb core 提交一个 urb,这个 urb 中间包含了一个命令,或者说控制请求,因为命令更适合于我们后来要讲的某个重要的概念.这里我们要发送的就是 GET MAX LUN.我们调用了一个函数, usb_stor_control_msg,其作用从名字上也可以看出,发送控制信息,即控制请求.做一件事情要遵守一件事情的规矩,发送控制信息必须按照一定的格式,发出去了人家才能看得懂.就像你要给远方的恋人寄一封信,你要在信封上按基本格式填写一些东西,你的信才能被寄出去对吗.

我们结合 usb_stor_control_msg 函数本身以及我们调用它的时候其具体上下文包括实际传递给它的参数来读这段代码,看看这个格式究竟是如何的.首先第一个参数 us,这个不用多说了吧.星星还是那个星星,月亮也还是那个月亮,山也还是那座山,梁也还是那到梁,us 自然总还是那个 us.而 us 中有一个成员,叫做 cr,见 struct us_data 的定义:

```
149         struct usb_ctrlrequest  *cr;            /* control requests     */
```

她是 struct usb_ctrlrequest 结构指针, usb 规范规定了一个控制请求的格式为一个 8 个字节的数据包,而 struct usb_ctrlrequest 正是专门为此而准备的 8 个字节的一个变量,所以控制传输中总会用到她.她的定义在 include/linux/usb_ch9.h 中:

```
 86 /**
 87  * struct usb_ctrlrequest - SETUP data for a USB device control request
 88  * @bRequestType: matches the USB bmRequestType field
 89  * @bRequest: matches the USB bRequest field
 90  * @wValue: matches the USB wValue field (le16 byte order)
 91  * @wIndex: matches the USB wIndex field (le16 byte order)
 92  * @wLength: matches the USB wLength field (le16 byte order)
 93  *
 94  * This structure is used to send control requests to a USB device.  It matches
 95  * the different fields of the USB 2.0 Spec section 9.3, table 9-2.  See the
 96  * USB spec for a fuller description of the different fields, and what they are
 97  * used for.
 98  *
 99  * Note that the driver for any interface can issue control requests.
100  * For most devices, interfaces don't coordinate with each other, so
101  * such requests may be made at any time.
102  */
103 struct usb_ctrlrequest {
104         __u8 bRequestType;
105         __u8 bRequest;
106         __le16 wValue;
107         __le16 wIndex;
```

```
108         __le16 wLength;
109 } __attribute__ ((packed));
```

不过需要说明一点,在 usb spec 中,不叫 cr,而叫 setup packet,在 struct urb 里边就有这么一个名字一样的成员,

```
799         unsigned char *setup_packet;    /* (in) setup packet (control only) */
```

因为在 Linux 中 usb 的世界里一个通用的数据结构是 urb,很多函数都是以 urb 为参数的,所以稍后我们实际上还是会让 urb 中的 setup_packet 指针指向这个 usb_ctrlrequest 结构体的.毕竟我们最终要提交的就是 urb,而不是 cr.并且 cr 只是出现在控制传输,而 urb 却是四大传输都要用到的.

Ok,在 usb mass storage bulk only transport 协议里面,规定的很清楚,要发送 GET MAX LUN 请求,必须把 bmRequestType 设置为 device to host,class,interface,同时把 bRequest 设置为 254(FEh),即咱们这里的 0xfe,而 wValue 设置为 0,wIndex 设置为接口的编号,最后 wLength 设置为 1.

需要解释一下,什么叫 bmRequestType 设置 device to host,class,interface?实际上,usb 2.0 规范里面指出,所有的 usb 设备都会响应主机的一些请求,这些请求必须来自主机,通过设备的默认控制管道发出.(即 0 号端点所对应的那个管道)这些请求,或者说这些 request,是通过控制传输来实现的,请求以及请求所需的参数都是通过 setup packet 来发送的.主机负责建立好这个 setup packet.(也就是咱们刚才的那个 cr,后来的 setup_packet.)每个 setup packet 包含 8 个 bytes.她们的含义如下:

byte0: bmRequestType,注意,在刚才咱们代码中数据结构 struct ctrlrequest 里边是写的 bRequestType,但是她们对应的是相同的冬冬.而之所以 usb 协议里写成 bmRequestType,是因为她实际上又是一个位图(m 表示 map),也就是说,尽管她只有 1 个 byte,但是她仍然是当作 8 位来用.她的 8 位的含义是:

D7: 控制传输数据传输的方向,0 表示方向为主机到设备,1 表示方向为设备到主机.(有时控制传输除了发命令阶段以外,可能还有数据传输阶段,此处表示的是在数据传输那个阶段的传输方向.)

D6...5: 请求的类型,0 称为标准类型(Standard),1 称为 Class,2 称为 Vendor,3 则是保留的,不能使用.

D4...0: 接收者,0 表示设备,1 表示接口,2 表示端点,3 表示其它,4...31 都是保留的,不能使用.

所以,bmRequestType 被设 device to host,class,interface,表示,数据传输阶段传输方向是设备到主机,请求的类型是 Class,意思是,请求类型可以有好几种,首先 usb spec 本身定义了很多种标准请求,所有的 usb 设备都必须支持这些标准请求,或者或标准 request,另一方面,一类设备或者厂商自己也可以定义自己的额外的一些请求,显然这里 get max lun 就是 usb mass storage 这个类,或者说这个 class 所定义的请求,因此,请求类型设置为 class,而 request 接收者可以是设备,也可以是接口,还可以是端点,对 get max lun 这个命令来说,她是针对一个 u 盘的,所以她的接收者应该是 interface.

byte1: bRequest,1 个 byte,指定了是哪个请求.每一个请求都有一个编号,咱们这里是 GET MAX LUN,其编号是 FEh.

byte2...3: wValue,2 个 bytes,不同请求有不同的值,咱们这里刚才已经说了,必须为 0.

byte4...5: wIndex,2 个 bytes,不同请求有不同的值,咱们这里要求被设置为是 interface number.

byte6...7: wLength,2 个 bytes,如果接下来有数据传输阶段,则这个值表征了数据传输阶段传输多少个 bytes,没啥好说的,这个值在 GET MAX LUN 请求中被规定为 1,也就是说返回 1 个 byte 即可.

结合函数调用的实参和函数的定义,可见,对于这个 cr 来说,其 bRequest 被赋值为 US_BULK_GET_MAX_LUN,bRequestType 被赋值为 USB_DIR_IN|USB_TYPE_CLASS|USB_RECIP_INTERFACE,而 wValue 被设为 0,wIndex 被设置为 us->ifnum, 推开记忆的门,我们发现在 associate_dev()函数中,us->ifnum 被赋值为 intf->cur_altsetting->desc.bInterfaceNumber,即她确实对应了 interface number. wLength 被赋值为 1.设置好 cr 之后,就可以把她传递给 urb 的 setup_packet 了.对比上面的 usb_stor_control_msg()函数中第 232 行,和下面函数 usb_fill_control_urb()的定义,即可看出 urb 的 setup_packet 指针指向了这个 cr.具体来看 usb_fill_control_urb()函数,这个函数定义于 include/linux/usb.h 中:

```
812 /**
813  * usb_fill_control_urb - initializes a control urb
814  * @urb: pointer to the urb to initialize.
815  * @dev: pointer to the struct usb_device for this urb.
816  * @pipe: the endpoint pipe
817  * @setup_packet: pointer to the setup_packet buffer
818  * @transfer_buffer: pointer to the transfer buffer
819  * @buffer_length: length of the transfer buffer
820  * @complete: pointer to the usb_complete_t function
821  * @context: what to set the urb context to.
822  *
823  * Initializes a control urb with the proper information needed to submit
824  * it to a device.
825  */
826 static inline void usb_fill_control_urb (struct urb *urb,
827                                          struct usb_device *dev,
828                                          unsigned int pipe,
829                                          unsigned char *setup_packet,
830                                          void *transfer_buffer,
831                                          int buffer_length,
832                                          usb_complete_t complete,
833                                          void *context)
834 {
835         spin_lock_init(&urb->lock);
836         urb->dev = dev;
837         urb->pipe = pipe;
838         urb->setup_packet = setup_packet;
839         urb->transfer_buffer = transfer_buffer;
840         urb->transfer_buffer_length = buffer_length;
841         urb->complete = complete;
842         urb->context = context;
843 }
```

很显然,她就是为特定的 pipe 填充一个 urb(最初 urb 申请的时候被初始化为 0 了嘛不是).对比函数调用和函数定义,可知, 这个 pipe 被设为了 us->recv_ctrl_pipe,即接收控制管道,也就是说专门为设备向主机发送数据而设置的管道.而这个 urb 就是 us->current_urb,并且除了 urb->setup_packet(unsigned char 类型的指针)指向了 us->cr 之外,urb->transfer_buffer(void 指针)指向了 us->iobuf,urb->transfer_buffer_length(int 类型)被赋值为 1,urb->complete(usb_complete_t 类型)被赋值为 usb_stor_blocking_completion.

此处特意提一下 usb_complete_t 类型,在 include/linux/usb.h 中,有这么一行,

612 typedef void (*usb_complete_t)(struct urb *, struct pt_regs *);

这里用了 typedef 来简化声明,不熟悉 typedef 功能的可以去查一下,typedef 的强大使得以下两种声明作用相同:

一种是:

```
void (*func1)(struct urb *,struct pt_regs *);
void (*func2)(struct urb *,struct pt_regs *);
void (*func3)(struct urb *,struct pt_regs *);
```

另一种是:

```
typedef void (*usb_complete_t)(struct urb *, struct pt_regs *);
usb_complete_t func1;
usb_complete_t func2;
usb_complete_t func3;
```

看出来了吧,如果要声明很多个函数指针,她们的参数又都很复杂,那么显然使用 typedef 一次,就可以一劳永逸了,以后声明就很简单了.所以,咱们也就知道实际上,urb 中的 complete 是一个函数指针,她被设置为指向函数 usb_stor_blocking_completion().关于 usb_stor_blocking_completion()咱们暂且不提,等到用的时候再说.

看完了 usb_fill_control_urb,咱们再回到 usb_stor_control_msg()函数,接下来是下一个函数,usb_stor_msg_common(),注意了,usb_fill_control_urb 这个函数只填充了 urb 中的几个元素,struct urb 里面包含了很多东西,不过有一些设置是共同的,所以下面用 usb_stor_msg_common()函数来设置,可以看出给这个函数传递的参数只有两个,一个就是 us,另一个是 timeout(传给她的值是 HZ),我们继续进入到这个函数中来把她看个清清楚楚明明白白真真切切.天空收容每一片云彩,不论其美丑,所以天空宽阔无边;大地拥抱每一寸土地,不论其贫富,所以大地广袤无垠;海洋接纳每一条河流,不论其大小,所以海洋广阔无边;而这个函数支持各种传输,作为 usb-storage 模块,无论其采用何种传输方式,无论传输数据量的多少,最终都一定要调用这个函数,所以我们必须承认这个函数很伟大.

鉴于这个函数的重要性,它适用于各种信息的传送,而不仅仅是控制传送,日后在 bulk 传输中我们还将遇上它,我们将在下一节专门来分析.但是在看这个函数之前,有些事情必须要心里有数,那就是作为设备驱动程序,只需要提交一个 urb 就可以了,剩下的事情 usb core 会去处理,有了结果它会通知我们.而提交 urb,usb core 为我们准备了一个函数,usb_submit_urb()不管我们使用什么传输方式,我们都只要调用这个函数即可,在此之前,我们需要做的只是准备好这么一个 urb,把 urb 中各相关的成员填充好,然后就 ok 了.而这 usb_stor_msg_common 正是这样做的.而显然,不同的传输方式其填写 urb 的方式也不同.

最后我们需要指出一点,这里我们的 cr 是一个指针,有同志会问这个指针申请过内存了吗?答案是肯定的,忆往昔峥嵘岁月,曾几何时,我们在函数 associate_dev()中就见到了 us->cr,并且用 usb_buffer_alloc 为其申请了内存,当时我们就讲过的.

第一次亲密接触(三)

让我们单刀直入,进入 usb_stor_msg_common()函数.

首先看 145 行,让 us->flags 和 ABORTING_OR_DISCONNECTING 相与,ABORTING_OR_DISCONNECTING 宏定义于 drivers/usb/storage/usb.h 中:

```
78 /* Dynamic flag definitions: used in set_bit() etc. */
79 #define US_FLIDX_URB_ACTIVE     18  /* 0x00040000  current_urb is in use  */
80 #define US_FLIDX_SG_ACTIVE      19  /* 0x00080000  current_sg is in use   */
81 #define US_FLIDX_ABORTING       20  /* 0x00100000  abort is in progress   */
82 #define US_FLIDX_DISCONNECTING  21  /* 0x00200000  disconnect in progress
*/
83 #define ABORTING_OR_DISCONNECTING       ((1UL << US_FLIDX_ABORTING) |
\
84                                          (1UL << US_FLIDX_DISCONNECTING))
85 #define US_FLIDX_RESETTING      22  /* 0x00400000  device reset in progress
*/
86 #define US_FLIDX_TIMED_OUT      23  /* 0x00800000  SCSI midlayer timed
out  */
```

她只是一个 flag,咱们知道,每一个 usb mass storage 设备,会有一个 struct us_data 的数据结构,即 us,所以,在整个 probe 的过程来看,她相当于一个"全局"的变量,因此咱们可以使用一些 flags 来标记一些事情.比如,此处,对于提交 urb 的函数来说,显然她不希望设备此时已经处于放弃或者断开的状态,因为那样就没有必要提交 urb 了嘛不是.

而下一个函数 init_completion(),只是一个队列操作函数,她被定义于 include/linux/completion.h 中:

```
 13 struct completion {
14          unsigned int done;
15          wait_queue_head_t wait;
16 };
24 static inline void init_completion(struct completion *x)
25 {
26          x->done = 0;
27          init_waitqueue_head(&x->wait);
28 }
```

她只是调用了 init_waitqueue_head 去初始化一个等待队列.而 struct completion 的定义也在上面已经列出.关于 init_waitqueue_head 咱们将在下面的故事中专门进行描述.

而接下来,都是在设置 us 的 current_urb 结构,咱们看 161 行,transfer_flags 被设置成了 URB_ASYNC_UNLINK | URB_NO_SETUP_DMA_MAP,其中 URB_ASYNC_UNLINK 表明 usb_unlink_urb()函数将被异步调用,不懂异步调用也没有关系,因为内核的发展是如此的迅速,这一点完全可以和上海的发展速度有得一拼,最新的内核中,人们已经很少用这个宏了,因为 usb_unlink_urb 已经是

异步调用的函数了,如果是同步调用,则可以使用另一个函数 usb_kill_urb().这两个函数的作用就是取消一个 urb 请求.同步调用用于这样一种情况,即函数执行过程中可以进入睡眠,满足一定条件再醒来继续执行,而异步调用则不会睡眠,那么这里之所以要设置 URB_ASYNC_UNLINK,其目的就在于让 usb_unlink_urb 是异步的执行,原因是,有的时候我们取消一个 urb request 的时候是处在一种不能睡眠的上下文,比如后面我们会看到的处理超时的函数.我们在互联网上下载一个很大文件的时候,常常会遇到超时的情况,然后我们的请求就被中止了.对于 usb 系统同样有这个问题,我们会给一个 urb 设置超时,提交上去到了一定时间还没能传输好的话就意味着可能传输有问题,这种情况我们通常也会把这个 urb 给取消掉.于是这里有两种情况,第一,我们提交了 urb 之后就不管事了,我们等待,怎么等待?睡眠,我们的进程进入睡眠,换句话说在我们这个上下文情景中也就是 storage_probe 函数睡了.如果不超时,那么万事大吉,urb 执行完了之后我们的进程被唤醒,继续往下走.那么天下平安.但是第二种情况是,我们进入了睡眠,可是时间到了,urb 还没有执行完,那么超时函数会被执行,它就会去取消这个 urb,如果说这个超时函数也可以睡眠,那就不得了了,天下大乱了,都睡了,谁干活?整个驱动不就瘫痪了!所以,这种情况下我们要调用的函数是不能睡眠的,也就是说必须是异步的.而 usb_unlink_urb 内部会通过判断是否有 URB_ASYNC_UNLINK 这么一个 flag 被设置在了 urb->transfer_flags 中,来选择是异步执行还是同步执行,实际上它的同步执行就是调用 usb_kill_urb 而已.不过刚才说的,最新版的内核,比如 2.6.21,它里边就把这两个函数彻底分开了,不用设这么一个 flag,usb_unlink_urb 就是异步,usb_kill_urb 就是同步.

而 URB_NO_SETUP_DMA_MAP 表明,如果使用 DMA 传输,则 urb 中 setup_dma 指针所指向的缓冲区是 DMA 缓冲区,而不是 setup_packet 所指向的缓冲区.接下来再或上 URB_NO_TRANSFER_DMA_MAP 则表明,如果本 urb 有一个 DMA 缓冲区需要传输,则该缓冲区是 transfer_dma 指针所指向的那个缓冲区,而不是 transfer_buffer 指针所指向的那一个缓冲区.换句话说,如果没设置这两个 DMA 的 flag,那么 usb core 就会使用 setup_packet 和 transfer_buffer 作为数据传输的缓冲区,然后下面两行就是把 us 的 iobuf_dma 和 cr_dma 赋给了 urb 的 transfer_dma 和 setup_dma.(157 至 160 的注释表明,只要 transfer_buffer 被赋了值,那就假设有 DMA 缓冲区需要传输,于是就去设 URB_NO_TRANSFER_DMA_MAP.)关于 DMA 这一段,因为比较难理解,所以我们多说几句.

首先,这里是两个 DMA 相关的 flag,一个是 URB_NO_SETUP_DMA_MAP,而另一个是 URB_NO_TRANSFER_DMA_MAP.主意这两个是不一样的,前一个是专门为控制传输准备的,因为只有控制传输需要有这么一个 setup 阶段需要准备一个 setup packet.而我们把让 setup_packet 指向了 us->cr 别忘了我们当初为 us->cr 申请内存的时候用的是下面这句:

```
449         /* Allocate the device-related DMA-mapped buffers */
450         us->cr = usb_buffer_alloc(us->pusb_dev, sizeof(*us->cr),
451                         GFP_KERNEL, &us->cr_dma);
```

别忘了这里的 us->cr_dma,这个函数虽然返回值是赋给了 us->cr,但与此同时,在 us->cr_dma 中记录的可以该地址所映射的 dma 地址,那么刚才这里设置了 URB_NO_SETUP_DMA_MAP 这么一个 flag,就说明,如果是 DMA 方式的传输,那么 usb core 就应该使用 us->cr_dma 里边的冬冬去进行 dma 传输,而不要用 us->cr 里边的冬冬了.换句话说,也就是 urb 里边的 setup_dma 而不是 setup_buffer.

同样 transfer_buffer 和 transfer_dma 的关系也是如此,我们当初同样用类似的方法申请了 us->iobuf 的内存:

```
457         us->iobuf = usb_buffer_alloc(us->pusb_dev, US_IOBUF_SIZE,
458                         GFP_KERNEL, &us->iobuf_dma);
```

这里就有 us->iobuf 和 us->iobuf_dma 这两个咚咚,但是我们注意到,163 和 164 行,我们在设置 URB_NO_TRANSFER_DMA_MAP 这个 flag 的时候,先做了一次判断,判断 us->current_urb->transfer_buffer 是否等于 us->iobuf,这是什么意思呢?我们在什么地方对 transfer_buffer 赋过值? 答案是 usb_fill_control_urb 中,我们把 us->iobuf 传递了过去,它被赋给了 urb->transfer_buffer,这样做就意味着我们这里将使用 DMA 传输,所以这里就设置了这个 flag,倘若我们不希望进行 DMA 传输,那很简单,我们在调用 usb_stor_msg_common 之前,不让 urb->transfer_buffer 指向 us->iobuf 就是了,反正这都是我们自己设置的,别人管不着.需要知道的是,transfer_buffer 是给各种传输方式中真正用来数据传输的,而 setup_packet 仅仅是在控制传输中发送 setup 的包的,控制传输除了 setup 阶段之外,也会有数据传输阶段,这一阶段要传输数据还是得靠 transfer_buffer,而如果使用 dma 方式,那么就是使用 transfer_dma.

Ok,下一句,169 行,终于到了提交 urb 这一步了,usb_submit_urb 得到调用,作为 usb 设备驱动程序,我们不需要知道这个函数究竟在做什么,只要知道怎么使用就可以了,无需关注代码背后的哲学.它的定义在 drivers/usb/core/urb.c 中,我们得知道它有两个参数,一个就是要提交的 urb,一个是内存申请的 flag,这里我们使用的是 GFP_NOIO,意思就是不能在申请内存的时候进行 IO 操作,道理很简单,咱们这个是存储设备,调用 usb_submit_urb 很可能是因为我们要读些磁盘或者 U 盘,那这种情况如果申请内存的函数又再一次去读写磁盘,那就有问题了,什么问题?嵌套呗.什么叫申请内存的函数也会读写磁盘?玩过 Linux 的人不会不知道 swap 吧,交换分区,干嘛要交换啊,可不就是因为内存不够么.使用磁盘作为交换分区不就方便了,所以申请内存的时候可能要的内存在磁盘上,那就得交换回来.这不就读写磁盘了么?所以我们为了读写硬盘而提交 urb,那么这个过程中就不能再次有 IO 操作了,这样做的目的是为了杜绝嵌套死循环.

于是我们调用了 169 行就可以往下走了,剩下的事情 usb core 和 usb host 会去处理,至于这个函数本身的返回值,如果一切正常,status 将是 0.所以这里判断如果 status 不为 0 那么就算出错了.

177 行,一个 urb 被提交了之后,通常我们会把 us->flags 中置上一个 flag, US_FLIDX_URB_ACTIVE, 让我们记录下这个 urb 的状态是活着的.

180 行,这里我们再次判断 us->flags,看是不是谁置了 aborting 或者 disconnected 的 flag.稍后我们会看到谁会置这些 flag,显然如果已经置了这些 flag 的话,咱们就没必要往下了,这个 urb 可以 cancel 了.

190 行,一个新的故事将被引出,这就是伟大的时间机制.

能冲刷一切的,除了眼泪,就是时间.所以 Linux 中引入了时间机制.

第一次亲密接触(四)

金城武说:不知道从什么时候开始,在什么东西上面都有个日期,秋刀鱼会过期,肉罐头会过期,连保鲜纸都会过期,我开始怀疑,在这个世界上,还有什么东西是不会过期的?

有时候我也被这个问题所困扰,我不知道是我不明白还是这世界变化太快.连 Linux 中都引入了过期这么一个概念.说文雅一点就是超时.设置一个时间,如果时间到了该做的事情还没有做完,那么某些事情就会发生.

比如,咱们需要做这样一些事情,定个闹钟,比如咱们需要烤蛋糕,现在是 8 点 30,而咱们要烤 45 分钟,所以咱们希望闹钟 9 点一刻响,然后当时间到了,闹钟就如咱们期待的一样,响个不停.在计算机中,咱们也需要做这样的事情,有些事情,咱们需要时间控制,特别是网络,通信,等等,凡是涉及数据传输的事儿,就得考虑超时,换句话说,定个闹钟,你要是在这个给定的时间里还没做好你该做的事情,那么停下来,别做了,肯定有问题,比如,咱们如果烤蛋糕 45 分钟,发现蛋糕一点香味都没有,颜色也没变,那肯定有问题,别烤了,先检查一下烤箱是不是坏了,是不是停电了,等等.而具体到咱们这里,需要用一个闹钟,或者叫专业一点,定时器,如果时间到了,就执行某个函数,这个功能 Linux 内核的时间机制已经实现了,咱们只需要按"说明书"调用相应的接口函数即可.看代码,190 行,如果 timeout>0,也就是说需要设置闹钟,那么首先需要定义一个 struct timer_list 结构体的变量,咱们这里定义的变量叫做 to_timer(在 usb_stor_msg_common 一开始就定义了的),然后用 init_timer()函数和 add_timer()函数来真正实现设置闹钟,init_timer()是初始化,然后设置好之后调用 add_timer 才能让闹钟生效.具体怎么设置的呢?在 add_timer()之前,为 to_timer.expires 赋值为 jiffies+timeout,to_timer.function 赋值为 timeout_handler,to_timer.data 赋值为 us.这表示,超时时间点为当前时间加上一个 timeout,(jiffies,Linux 内核中赫赫有名的全局变量,表示当前时间),timeout 咱们前面调用 usb_stor_msg_common 的时候给设置成了 HZ,也就是 1 秒.当时间到了之后,timeout_handler 函数会被执行,而 us 作为参数传递给她.不妨来看一下 timeout_handler 函数吧,她定义于 drivers/usb/storage/transport.c 中:

```
119 /* This is the timeout handler which will cancel an URB when its timeout
120  * expires.
121  */
122 static void timeout_handler(unsigned long us_)
123 {
124         struct us_data *us = (struct us_data *) us_;
125
126         if (test_and_clear_bit(US_FLIDX_URB_ACTIVE, &us->flags)) {
127                 US_DEBUGP("Timeout -- cancelling URB\n");
128                 usb_unlink_urb(us->current_urb);
129         }
130 }
```

看得出,其实也没做什么,就是清除 US_FLIDX_URB_ACTIVE flag,然后调用 usb_unlink_urb()函数撤销当前这个 urb.还记得刚才说的那个同步异步了吗?这正是刚才说的那个异步的情形,显然此刻这个函数不能睡眠,否则整个 driver 就挂了...

紧接着,199 行,非常重要的一句 wait_for_completion(&urb_done),这句话会使本进程进入睡眠.别忘了刚才我们那句 init_completion(&urb_done),urb_done 是一个 struct completion 结构体变量,这个定义在 usb_stor_msg_common()函数的第一行就出现了.显然 completion 是 Linux 中同步机制的一个很重要的结构体.与 wait_for_completion 对应的一个函数是 complete().其用法和作用是这样的:首先我们要用 init_completion 初始化一个 struct completion 的结构体变量,然后调用 wait_for_completion()这样当前进程就会进入睡眠,处于一种等待状态,而另一个进程可能会去做某事,当它做完了某件事情之后,它会调用 complete()函数,一旦它调用这个 complete 函数,那么刚才睡眠的这个进程就会被唤醒.这样就实现了一种同步机制,或者叫等待机制.那么我们来看 complete 函数在哪里被调用的,换句话说,咱们这里一旦睡去,何时才能醒来.

还记得在调用 usb_fill_control_urb()填充 urb 的时候咱们设置了一个 urb->complete 指针吗?没错,当时咱们就看到了,urb->complete=usb_stor_blocking_completion,这相当于向 usb host controller driver 传达了一个信息.所以,当 urb 传输完成了之后,usb host controller 会唤醒她,但不会直接唤醒她,而是通过执行之前设定的 urb 的 complete 函数指针所指向的函数,即调用 usb_stor_blocking_completion()函数去真正唤醒她.usb_stor_blocking_completion()函数定义于 drivers/usb/storage/transport.c 中:

```
109 /* This is the completion handler which will wake us up when an URB
110  * completes.
111  */
112 static void usb_stor_blocking_completion(struct urb *urb, struct pt_regs *regs)
113 {
114         struct completion *urb_done_ptr = (struct completion *)urb->context;
115
116         complete(urb_done_ptr);
117 }
```

这个函数就两句话,但她调用了 complete()函数,urb_done_ptr 就被赋为 urb->context,而 urb->context 是什么? usb_stor_msg_common()函数中,152 行,可不就是把刚初始化好的 urb_done 赋给了它么?这个函数可是 Linux 内核的核心函数,不要问她从哪里来,她会告诉你她来自内核底层,没错,她的户口在 kernel/sched.c,很显然,她就是唤醒刚才睡眠的那个进程.换言之,到这,wait_for_completion()将醒来,从而继续往下走.

如果你足够好奇,你会问如果超时,那么 timeout_handler 会被调用,于是 usb_unlink_urb 会被调用,然后呢?其实 usb_stor_blocking_completion 还是会被调用,而且会设置 urb->status 以告诉大家这个 urb 被 cancel 了.

下面只剩下几行代码了.首先是 clear_bit()清除 US_FLIDX_URB_ACTIVE,表明这个 urb 不再是 active 了.因为该干的事都干完了,就好比您的包裹已经寄到了,那显然您填的那个单子就没有用了.至少她上面应该有标志表明这份单子对应的包裹已经送过了,不要再送了.如果是超时了,那么也是一样的,urb 都被 cancel 了,当然就不用设为 active 了.

然后下一行,如果这时 timeout 还大于 0,那么说明刚才您设的那个超时闹钟还没到过期,而您该做的事情却已经做完了,所以这个闹钟就不需要设了,就好比邮局承诺您三天寄到,完了您记住了,三天她要没寄到,您就去索赔,所以您自个儿就订了个闹钟,三天真到期了您就可以去索赔,但是如果人家两天就给您寄到了,那您这个闹钟就没意义了嘛不是,所以这样您就得取消这个闹钟,省得她那弦老紧绷着,这里您也得删除刚才那个 to_timer,这样您可以调用 Linux 内核为您提供的函数 del_timer_sync(),她的参数就是刚才这个 to_timer 的地址.最后一句,usb_stor_msg_common()函数终于该返回了,return us->current_urb->status,返回的就是 urb 的 status.于是我们总算是可以离开这个函数了.

返回之后,又回到 usb_stor_control_msg()中来,如果 status 是 0,那么说明成功传输了,对于成功传输的情况,urb 的 actual_length 将被赋值为实际传输长度,然后 usb_stor_control_msg()也返回了,要么是实际长度,要么就是不成功的具体 status.因此我们也不得不离开这个函数.诚然,快乐要有悲伤作陪,雨过应该就有天晴.但是如果雨后还是雨,如果忧伤之后还是忧伤.请让我们从容面对这离别之后的离别.

于是,走过了千山万水,经历了千辛万苦,咱们再一次回到了久违的 usb_stor_Bulk_max_lun()函数.这样一次真正的控制传输就这么开始就这么结束了.

接下来,我们继续看,控制传输的结果返回给了 result,我们说过,单纯的 U 盘一般来说这个结果总是 0,即它必然只有一个 lun.这里判断的是 result 大于 0,道理很简单,result 是一个 int 型的数据,而返回的给它的实际上是 iobuf[0]这么一个 char 类型的变量,所以是字符'0',保存成 int 型当然就大于 0 了,所以这里打印出来结果是 0,但是 result 实际上是大于 0 的.而 945 行我们同样注意到,如果 result 是一些奇怪的值,正如注释所说,有的设备它就不认 GetMaxLUN 这个命令,就像在她的心里潜伏着一个深渊,扔下巨石也发不出声音来,或者它干脆就返回一个 0 长度的结果来,那么这种情况那么我们就只能把这种设备当作只有一个 LUN 了,所以也就返回 0 得了.

不过 933 到 935 这三行需要说一下.这也是专门为一些变态的设备准备的.一般的设备用不着.只是 usb_stor_clear_halt 这个函数是我们自己定义的,并且今后我们也会用到,所以我们还是讲一下.不过,在这个函数中,我们将再一次见到控制传输,但是毕竟不再是我们的第一次亲密接触了,所以,虽然我们依然还是在 usb_stor_Bulk_max_lun 中,但还是让我们下一节在讲吧.

最后需要解释一下的是,像 init_timer(),add_timer(),del_timer_sync()这几个函数都是 Linux 内核中的核心函数,包括结构体 struct timer_list,他们都来自 include/linux/timer.h 和 kernel/timer.c 中,我们只需要知道调用就可以了,不用知道究竟怎么实现的,只需要知道这样设置了超时的话,我们注册的超时处理函数就会被执行.而至于它究竟如何去调用的,如何计时如何去判断超时这些内核自会处理,不用我们担心,我们瞎操心也没用.对于内核来说,时间是怎样划破她的皮肤,只有她自己最清楚.而对于写设备驱动的人来说,这些核心代码就像是天空中的云朵,你看着它往某一个方向飘,却什么也做不了.正如令狐冲所说的那样:"有些事情本身我们无法控制,只好控制自己."

将控制传输进行到底

其实 usb_stor_clear_halt 这个函数的作用很简单,就是 spec 里边规定了,usb 设备中,有两类端点,必须具有一个叫做 Halt 的特征,啥是 Halt?查金山词霸去,中断,停止,暂停,怎么解释呢,你把手机关了,就不能给超级女生发短信投票了吧,你把电脑关了,就不能上黄色网站了吧,你把电视机关了,就不能看中国之队在亚洲杯上的精彩表演了吧.对于 usb 设备来说,其中端端点和 Bulk 端点就有这么一个特征,叫做 Halt,其实就是寄存器里的某一位,设为 1,就表示设置了 Halt 的特征,那就是表示这个端点不工作了.要想让端点重新工作,很简单,把这一位设置为 0 就可以了.

关于 Halt,用我们行话说,这叫做一个 feature,其实就是一个特征,坊间更喜欢说 feature.而 usb 设备实际上有很多 feature.确切的说,有的 feature 是算 device 的 feature,有些是 interface 的 feature,有些是 endpoint 的 feature.而 Halt 是 endpoint 的 feature. usb spec 规定了一些请求,比如 SET_FEATURE,以及 CLEAR_FEATURE,顾名思义,就是设置一个 feature 或者清除一个 feature.那么我们这里发生的是 clear halt,实际上就是执行 CLEAR_FEATURE,清除 halt 这个 feature.刚才通过 usb_stor_Bulk_max_lun()函数我们已经看到了对于一次控制传输,我们作为设备驱动需要做哪些工作,这里和刚才的区别仅仅在于,刚才发送的请求是 GET MAX LUN,而现在要发送的请求是 CLEAR FEATURE.另一方面呢,GET MAX LUN 是 usb mass storage spec 专门给它们这一小类设备定义的,而 CLEAR FEATURE 那是所有的 usb 设备都通用的,因为它是 usb spec 所规定的.

那么什么时候我们需要调用这个函数呢?先不说我们这里的上下文,实际上 usb spec 规定了,对于设备的 bulk 端点,每当设备在 reset 之后,需要清除 halt 这个 feature 然后端点才能正常工作.所以之后我们会看到,在 reset 相关的函数里我们会调用这个函数,那么我们此刻所遇到的这个函数是处于什么情景呢?如果不看注释你就能看懂,那么我只能说,你他妈的太有才了!开源社区需要你这样伟大的自由主义战士!注释里说得很清楚,有些变态的设备,它就是不跟你按常理出牌,人家能正常响应 GetMaxLUN 这个 request,它偏要耍个性,就是不认 spec,你发送 GetMaxLUN 请求过来,它不予回复,它出现 STALL 的特点,什么是 STALL?其实就是 Halt,端点挂起,或者通俗一点理解,就是死机了.所以,毫无疑问,我们要把这个 halt 给清掉,否则设别没有办法工作了.

Ok,是时候该看看函数内部了,这是一个定义于 drivers/usb/storage/transport.c 中的函数.usb_stor_clear_halt():

```
243 /* This is a version of usb_clear_halt() that allows early termination and
244  * doesn't read the status from the device -- this is because some devices
245  * crash their internal firmware when the status is requested after a halt.
246  *
247  * A definitive list of these 'bad' devices is too difficult to maintain or
248  * make complete enough to be useful.  This problem was first observed on the
249  * Hagiwara FlashGate DUAL unit.  However, bus traces reveal that neither
250  * MacOS nor Windows checks the status after clearing a halt.
251  *
252  * Since many vendors in this space limit their testing to interoperability
253  * with these two OSes, specification violations like this one are common.
254  */
255 int usb_stor_clear_halt(struct us_data *us, unsigned int pipe)
256 {
257         int result;
258         int endp = usb_pipeendpoint(pipe);
259
260         if (usb_pipein (pipe))
261                 endp |= USB_DIR_IN;
262
263         result = usb_stor_control_msg(us, us->send_ctrl_pipe,
264                 USB_REQ_CLEAR_FEATURE, USB_RECIP_ENDPOINT,
265                 USB_ENDPOINT_HALT, endp,
266                 NULL, 0, 3*HZ);
267
268         /* reset the endpoint toggle */
269         usb_settoggle(us->pusb_dev, usb_pipeendpoint(pipe),
270                 usb_pipeout(pipe), 0);
271
272         US_DEBUGP("%s: result = %d\n", __FUNCTION__, result);
273         return result;
274 }
```

258 行,usb_pipeendpoint,定义于 include/linux/usb.h 中,

```
1091 #define usb_pipeendpoint(pipe)  (((pipe) >> 15) & 0xf)
```

很简单,右移 15 位,然后与 0xf 相与,得到的自然就是原来 pipe 里边的 15 至 18 位. 我们曾经讲过,一个 pipe 的 15 位至 18 位是 endpoint 号,(一共 16 个 endpoint,)所以很显然,这里就是得到 endpoint 号. 然后把她赋给了 endp.然后 usb_pipein()也定义于同一文件中,

```
1088 #define usb_pipein(pipe)          ((pipe) & USB_DIR_IN)
1089 #define usb_pipeout(pipe)         (!usb_pipein(pipe))
```

显然,就是判断她是不是 IN 的管道.如果是 IN,那么她返回 1,反之,返回 0.usb_pipeout 则相反.261 行,如果是,就或上.

263 行,再一次调用 usb_stor_control_msg 来传递信息了.USB_REQ_CLEAR_FEATURE 对应的 usb spec 的一个标准请求命令 CLEAR_FEATURE(即凡是 usb 设备就应该支持的命令),表示清除一个设备的某种特征,而 USB_ENDPOINT_HALT 则对应 usb 的端点特征,每个端点都有这么一个特征,ENDPOINT_HALT,她指出端点是否处于停止状态.CLEAR_FEATURE 命令用来清除该端点的停止状态.说明了 CLEAR_FEATURE 清除的是端点的特征.结合 usb_stor_control_msg 形参实参来看,usb spec(Table 9-3)规定对于这个请求,wValue 要被设置为被 Feature Selector,赋值为 USB_ENDPOINT_HALT,即选择的 Feature 是 ENDPOINT_HALT,而 wIndex 要被设置为指定一个 Endpoint,参考 usb2.0 规范,在指定一个 Endpoint 时 wIndex 的格式,可知,低四位为端点号(D3~D0),D7 为方向,(IN/OUT),其余各位为保留位.实际上赋值为 endp,正是包含了方向和端点号这两个信息.wLength 要求被设置为 0,data 设置为 NULL,这些都没错.超时设了 3s.酱紫,就可以清除这个 Endpoint 的 ENDPOINT_HALT 这个 flag.关于 usb_stor_control_msg 我们当然就不用再讲了,忘记了的回头去看吧,反正一样的天一样的脸,一样的函数就在你面前.唯一不同的只是传递的参数不同罢了,也许这就是曾经沧海难为水吧,我们的人生也是如此,只能是一条不归路,走上去,就回不了头,谁也没有办法重走一遍曾经的路.

需要特别注意一下,上次 GETMaxLUN 调用 usb_stor_control_msg 的时候,我们倒数第四个参数是设了 0,而这里我们传递了一个 endp,这是因为不同的请求 spec 里边规定好了的,虽然这两个命令控制的对象不一样,但是作为控制传输,主机总是和控制端点在发生关系.并不因为这里是清楚 bulk 端点的 Halt Feature 就要发送给 bulk 端点,控制传输永远都只是发生在主机和控制端点之间.而真正要控制 bulk 端点,正是通过我们这里这个 endp 这么一传递,设备自然就知道该干嘛了.

接下来,269 行,又是一个定义于 include/linux/usb.h 中的宏,

```
1101 #define usb_settoggle(dev, ep, out, bit) ((dev)->toggle[out] =
((dev)->toggle[out] & ~(1 << (ep))) | ((bit) << (ep)))
```

dev 是 struct usb_device 结构体指针,直到现在才知道,struct usb_device 结构体中有 unsigned int toggle[2]这么一个数组,这个数组有两个元素,对应 endpoint 的 IN 和 OUT.拿 OUT 来说,每一个 endpoint 在这里占一位触发位,在 usb 控制传输的数据传输时,每一个包的头部是交替的,有两种包头,DATA0 和 DATA1,为了保证传输的正确性,一次用 DATA0,一次用 DATA1,一旦哪次没有交叉,host 就知道出错了.这里所谓的 usb_settoggle,就是对指定的 ep 所对应的那个 toggle 位 reset 成 0,然后如果

bit 不为 0,则把 bit 左移到 ep 对应的那位再和 toggle 或上,也就是说,把这个 toggle 位 reset 成'bit',比如 bit 为 1,那么就是 reset 成 1,如果 bit 为 0,那么就是 reset 成 0,大多数情况下 reset 都是复位成 0,但有时也会不是 0,这些都得看心情而定了.((dev)->toggle[out] & ~(1<<(ep)))就是把 1 左移 ep 位,比如 ep 为 3,那么就是得到了 1000,然后取反,得到 0111,(当然高位还有更多个 1),然后(dev)->toggle[out]和 0111 相与,这就是使得 toggle[out]的第 3 位清零而其她位都不变.然后咱们这里 bit 传递进来的是 0,所以就不起什么作用,还是 reset 成 0.总之,269 行做的事情就是把指定的 Endpoint 的和指定的 pipe 对应的那位 toggle 位给清零.)

当然细心的人会看一下 spec,spec 里面说了,对于使用 data toggle 的 endpoint,不管其 halt feature 是否被设置了,总之只要你调用 Clear Feature,那么其 data toggle 总是会被初始化为 Data0).所以有人就奇怪了,既然调用 Clear Feature 就已经把 data toggle 位初始化为 0 了,那这里为什么还要再次作一次 set toggle 呢?

事实上是这样的,其实这个世界上有两个 toggle bits,不是两个 toggle bit,是两个 toggle bit_s_,单复数别看错了,其实设备里边是有一个 toggle bits,而我们这里软件层次上,也定义了 toggle bits,这个 toggle bits 是给 host 用的,设备里边的那个 toggle bits 在 clear feature 之后,没错,是被初始化成 Data0 了,但是 host 这边他也想记录下这么一个序列,所以写代码的哥们儿就定义了这个一个数组,而这里调用 set toggle 的目的无非就是想让这个数组和设备中物理上的那个 toggle bits 保持同步.

到这里这个函数也就结束了,返回的是这次控制传输的结果,不过我们注意到调用这个函数的上下文,并没有人会 care 这个返回值,也许这里再判断返回值的意义不大了吧,本来就是在处理出错的代码中.

至此, usb_stor_Bulk_max_lun 这个函数也终于要返回了.于是我们终于,终于再一次回到了 usb_stor_acquire_resources 函数中,我容易吗?别说我了,歌神张学友在看到 Linux 内核代码如此复杂也不得不感慨说,这代码是一张无边无际的网,轻易就将我困在网中央,我越陷越深越迷惘,路越走越远越漫长,如何我才能锁住这个函数...

776 行,令 us->max_lun 等于刚才 usb_stor_Bulk_max_lun()的返回值.接下来,我们将看到一行具有划时代意义的代码.从此我们唱着东方红,走进新时代,这就是伟大的 S-C-S-I.

横空出世的 scsi

世界上最遥远的距离,
不是生与死,
而是我就站在你面前,
你却不知道我爱你.
世界上最遥远的距离,
不是我就站在你面前,
你却不知道我爱你,
而是明明知道彼此相爱,
却不能在一起.
世界上最遥远的距离,
不是明明知道彼此相爱,
却不能在一起,

而是明明无法抵挡这股想念,
却还得故意装作丝毫没有把你放在心里.
世界上最遥远的距离,
不是明明无法抵挡这股想念,
却还得故意装作丝毫没有把你放在心里,
而是用自己冷默的心,
对爱你的人掘了一条无法跨越的沟渠.

曾经天真的以为代码看到这里,该出场的函数也都出场了,该出场的数据结构也都出场了,不会有什么新鲜的冬冬了.曾经天真的以为我们即将知道整个驱动是如何工作的了.未曾想到,我们距离完全了解整个故事还有一光年. 挡在我们面前的,是 scsi. 的确,看一个 U 盘驱动不仅仅是要了解 usb 协议,还要懂 scsi 协议,要知道 U 盘它不仅仅是 usb 设备,它还是"盘",它要存储数据,所以才叫它 usb mass storage,所以才叫它海量存储,而 U 盘,它所遵循的传输协议叫 bulk-only 传输,它所遵循的指令集叫做 SCSI transparent command sets.换句话说,U 盘究竟怎么通信?使用 scsi 命令.你不懂 scsi 协议行吗?

没办法,如果你对 scsi 协议完全不了解.那么对不起,先抽点空熟悉一下 scsi 协议,熟悉一下 scsi 命令集吧.不要说你没有时间,雷锋同志说的好,时间就像乳沟,只要肯挤,总是有的.去 google 一把吧,去百度一把吧.我们在这里等你.

如果你真的不去看,那好吧,我假设你了解一点吧,什么是 scsi?无非也是一类总线.不过我们通常大多数普通人并不会接触 scsi,公司里边用得多,比如 scsi 硬盘,校园里边大学生通常不用 scsi 硬盘,用 ide 硬盘.每种硬盘都有它的市场.就像每个明星都有它的 fans 一样.于是我们知道这个世界上有玉米,有凉粉,有盒饭.那么常见的硬盘就是 scsi 硬盘和 ide 硬盘.scsi 硬盘属于 scsi 设备中的一种,有设备就有总线,有总线就有协议,所以我们知道了这个世界上有一种协议叫做 scsi 协议,就好比我们 usb 世界里有 usb 协议一样.时下流行的是 SCSI-2 协议.Linux 内核代码中自然也按这个协议来为 scsi 设备准备设备驱动程序.

关于 scsi,drivers 目录下面当然也有一个子目录是属于它的,那就是 drivers/scsi 目录.如果你有雅兴用 ls 命令看一下,你会发现这下面的文件那是相当的多.如果你真的很感兴趣,那么请从 Kconfig 文件和 Makefile 文件开始看起.去深入了解一下 Linux 整个 scsi 子系统是怎么工作的.我就不奉陪了.不过正如我们曾经介绍过的 2.6 中伟大的设备模型实现了这么一件事情,不管你是 pci 还是 usb 还是 scsi,都给你定义一条总线,然后总线上面两棵树,一棵是设备,一棵是驱动,对于设备这棵树,pci 有 pci 的扫描方法,usb 有 usb 的扫描方法,scsi 有 scsi 的扫描方法,总之这个过程被称为总线枚举,枚举完了之后设备这棵树就建立好了,同时 drivers 这棵树也会一步一步建立.每类设备有它自己的比较方法,要是合适,就把一个设备和一个驱动绑定起来,这样子,驱动程序提供的函数自然就会在需要的时候被调用,那么,谁来调用?

Ok,如果你是代码设计者,你会怎么处理?你打算如何为整个 scsi 系统规划代码?不知道?真的不知道?那么我真的羡慕你这么年轻就认识我了.不过,可惜,我也不知道.经过在复旦四年的大学教育,我已经被训练成了一名合格的人渣.这几年里我们关注的只是璩美凤的被偷拍事件,只是赵忠祥的录音事件,只是李金斗的嫖娼事件,只是阿丘的包二奶事件,却从未曾关注过自己应该学点什么,作为一名生在红旗下长在新中国的共产主义接班人,惭愧啊!

算了,不知道就不知道吧,让我们来思考一下.就像 usb 子系统那边一样,usb 那边有一部分核心代码,被称为 usb core,那么 scsi 这边自然也应该有这么一部分代码吧,也叫 scsi core,这你没意见吧.usb 那边弄了一个 usb host 目录,然后各种设备也分了类,比如 storage 设备,比如 input 设备,比如 serial 设备,比如 image 设备,那是因为 usb 的世界里有两个角色,一个是 host,一个是设备,那 scsi 这边是不是也可以这样

呢?先不说可不可以这样,事实情况是,没有这样,所有的冬冬都一股脑儿堆在 drivers/scsi/目录下面,有朝一日你要是混入了开发队伍中,你不妨提议把这个目录整理一下,别像现在这样,至少看上去整齐一点,乱七八糟不象话.不过也许开发者们有他们自己的理由吧,他们也许没有时间,那么你可以告诉他们雷锋同志是如何说的.开发者们把 scsi 设备分成了四类,于是他们写了四个模块来为这些设备做驱动程序.这四个模块是 sd_mod.ko,sr_mod.ko,st.ko,sg.ko.如果你正在 Linux 下使用 U 盘,那么用 lsmod 查看一下当前安装的模块,你一定会看到一个叫做 sd_mod 的模块,一定会看到一个叫做 scsi_mod 的模块,scsi_mod.ko 正是 scsi 的核心模块,即所谓的 scsi core.那么 scsi host 呢?host 同样也有一个模块,这就得看你具体用的是什么 host 了,行话管这个叫 HBA,即 host bus adapter.而相应的驱动程序就叫 Host Bus Adapter driver 了.正如在 usb 系统中,所有的总线上的活动都是以 host 为主的,而 scsi 也是如此,所有的设备也都是围着 host 转.即使地球不自转了,设备仍然要围着 host 转.不过你会很奇怪,即使你的机器里没有一个叫做 HBA 的冬冬,可是你的 U 盘还是能用啊,这是怎么回事?没错,怪事年年有,今年特别多,Linux 内核代码虽然繁华美丽,对我们来说,却常是朦胧不真实.设计者们是如何处理 usb-storage 和 scsi 的接口的呢?他们用代码虚拟了一个 scsi host.所以,如果你用 cat /proc/scsi/scsi 命令,就可以看看 scsi 设备中 U 盘是怎么被描述的.

下面是没有插 U 盘的一个例子:

```
localhost:~ # cat /proc/scsi/scsi
Attached devices:
Host: scsi0 Channel: 00 Id: 08 Lun: 00
  Vendor: DP       Model: BACKPLANE        Rev: 1.00
  Type:   Enclosure                        ANSI SCSI revision: 05
Host: scsi0 Channel: 02 Id: 00 Lun: 00
  Vendor: DELL     Model: PERC 5/i         Rev: 1.00
  Type:   Direct-Access                    ANSI SCSI revision: 05
Host: scsi1 Channel: 00 Id: 00 Lun: 00
  Vendor: SUN      Model: StorEdge 3510    Rev: 415F
  Type:   Enclosure                        ANSI SCSI revision: 03
Host: scsi1 Channel: 00 Id: 01 Lun: 00
  Vendor: SUN      Model: StorEdge 3510    Rev: 415F
  Type:   Direct-Access                    ANSI SCSI revision: 03
Host: scsi1 Channel: 00 Id: 01 Lun: 01
  Vendor: SUN      Model: StorEdge 3510    Rev: 415F
  Type:   Direct-Access                    ANSI SCSI revision: 03
Host: scsi1 Channel: 00 Id: 01 Lun: 02
  Vendor: SUN      Model: StorEdge 3510    Rev: 415F
  Type:   Direct-Access                    ANSI SCSI revision: 03
Host: scsi1 Channel: 00 Id: 01 Lun: 03
  Vendor: SUN      Model: StorEdge 3510    Rev: 415F
  Type:   Direct-Access                    ANSI SCSI revision: 03
Host: scsi1 Channel: 00 Id: 01 Lun: 04
  Vendor: SUN      Model: StorEdge 3510    Rev: 415F
  Type:   Direct-Access                    ANSI SCSI revision: 03
Host: scsi1 Channel: 00 Id: 01 Lun: 05
  Vendor: SUN      Model: StorEdge 3510    Rev: 415F
```

```
  Type:   Direct-Access                    ANSI SCSI revision: 03
Host: scsi1 Channel: 00 Id: 01 Lun: 06
  Vendor: SUN      Model: StorEdge 3510    Rev: 415F
  Type:   Direct-Access                    ANSI SCSI revision: 03
Host: scsi1 Channel: 00 Id: 01 Lun: 07
  Vendor: SUN      Model: StorEdge 3510    Rev: 415F
  Type:   Direct-Access                    ANSI SCSI revision: 03
```

而下面第二条是机器里有 U 盘的情况:

```
  localhost:~ # cat /proc/scsi/scsi
Attached devices:
Host: scsi0 Channel: 00 Id: 08 Lun: 00
  Vendor: DP       Model: BACKPLANE        Rev: 1.00
  Type:   Enclosure                        ANSI SCSI revision: 05
Host: scsi0 Channel: 02 Id: 00 Lun: 00
  Vendor: DELL     Model: PERC 5/i         Rev: 1.00
  Type:   Direct-Access                    ANSI SCSI revision: 05
Host: scsi1 Channel: 00 Id: 00 Lun: 00
  Vendor: SUN      Model: StorEdge 3510    Rev: 415F
  Type:   Enclosure                        ANSI SCSI revision: 03
Host: scsi1 Channel: 00 Id: 01 Lun: 00
  Vendor: SUN      Model: StorEdge 3510    Rev: 415F
  Type:   Direct-Access                    ANSI SCSI revision: 03
Host: scsi1 Channel: 00 Id: 01 Lun: 01
  Vendor: SUN      Model: StorEdge 3510    Rev: 415F
  Type:   Direct-Access                    ANSI SCSI revision: 03
Host: scsi1 Channel: 00 Id: 01 Lun: 02
  Vendor: SUN      Model: StorEdge 3510    Rev: 415F
  Type:   Direct-Access                    ANSI SCSI revision: 03
Host: scsi1 Channel: 00 Id: 01 Lun: 03
  Vendor: SUN      Model: StorEdge 3510    Rev: 415F
  Type:   Direct-Access                    ANSI SCSI revision: 03
Host: scsi1 Channel: 00 Id: 01 Lun: 04
  Vendor: SUN      Model: StorEdge 3510    Rev: 415F
  Type:   Direct-Access                    ANSI SCSI revision: 03
Host: scsi1 Channel: 00 Id: 01 Lun: 05
  Vendor: SUN      Model: StorEdge 3510    Rev: 415F
  Type:   Direct-Access                    ANSI SCSI revision: 03
Host: scsi1 Channel: 00 Id: 01 Lun: 06
  Vendor: SUN      Model: StorEdge 3510    Rev: 415F
  Type:   Direct-Access                    ANSI SCSI revision: 03
Host: scsi1 Channel: 00 Id: 01 Lun: 07
  Vendor: SUN      Model: StorEdge 3510    Rev: 415F
  Type:   Direct-Access                    ANSI SCSI revision: 03
```

Host: scsi3 Channel: 00 Id: 00 Lun: 00
  Vendor: Leidisk Model: USB Flash Drive Rev:
  Type: Direct-Access ANSI SCSI revision: 02

看出区别了吗?没错,最下面多了一项,这是一个 Leidisk 生产的 U 盘.这里本没有一个真实的 host,但是写代码的高手们却让你觉得也许似乎大概有,甚至连 scsi core 都被欺骗了一样.那么这些高手们是如何实现的呢?让我们来慢慢的看,让我们来看看这个神奇的 host 究竟是如何出现的,看看它是否像如今的房价一样,是否像林志玲的胸一样,看着坚挺,实际里面全是泡沫...

谁是最变态的结构体

scsi 子系统里的设备使用 scsi 命令来通信,scsi spec 定义了一大堆的命令,spec 里称这个为命令集,即所谓的 command set.其中一些命令是每一个 scsi 设备都必须支持的,另一些命令则是可选的.而作为 U 盘,它所支持的是 scsi transparent command set,所以它基本上就是支持所有的 scsi 命令了,不过我们其实并不关心任何一个具体的命令,只需要了解一些最基本的命令就是了.比如我们需要知道,所有的 scsi 设备都至少需要支持以下这四个 scsi 命令:INQUIRY,REQUEST SENSE,SEND DIAGNOSTIC,TEST UNIT READY.一会我们在代码中会遇见其中的几个,暂且不表.另外对于磁盘设备,它还需要支持另外一些命令,比如读方面的 READ 命令,写方面的 WRITE 命令,又比如我们经常做的格式化操作,它就对应 FORMAT UNIT 命令.对于磁盘这样的设备,SCSI 协议里边称它为 direct-access devices.这就是为什么你刚才在 cat /proc/scsi/scsi 的输出中能看到一个"Type:Direct-Access"这么一项.

知道了 scsi 总线上使用 scsi 命令来通信,那么我们下一步需要知道 scsi host 的作用,它主要就是负责发送命令给设备,然后设备就去执行命令.所以 scsi host 也被称为 initiator(发起者),而 scsi 设备被称为 target(目的地).

那么我们就知道,如果我们没有 scsi host,但是我们有遵守 scsi 协议接受 scsi 命令的 device,那怎么办?谁来发起命令?没有硬件我们用软件,命令是谁传递过来的?应用层?或者 scsi core?不管是谁,只要我们能够把上层的命令传递给设备,那就 Ok 了对不对?scsi 核心层把一切都做好了,我们只要为一个 scsi host 申请相应的数据结构,让命令来了能够发送给设备,能够让设备接收到命令,那就万事大吉了对不对?或者说整个 usb-storage 的真正的功能也就实现了对不对?到这里我们就可以开始继续来看我们的代码了.别忘了我们还在 usb_stor_acquire_resources()函数中,只不过刚刚讲完 usb_stor_Bulk_max_lun()函数而已.

781 行,us->unusual_dev->initFunction 是什么?不要说你一点印象也没有.在分析 unusual_devs.h 文件的时候曾经专门举过例子的,说有些设备需要一些初始化函数,它就定义在 unusual_devs.h 文件中,而我们通过 UNUSUAL_DEV 的定义已经把这些初始化函数给赋给了 us->unusual_dev 的 initFunction 指针了.所以这时候,在传输开始之前,我们判断,是不是有这样一个函数,即这个函数指针是否为空,如果不为空,很好办,执行这个函数就是了.比如当时我们举例子的时候说的那两个惠普的 CD 刻录机就有个初始化函数 init_8200e,那么就让它执行好了.当然,一般的设备肯定不需要这么一个函数.至于传递给这个函数的参数,在 struct us_unusual_dev 结构体定义的时候,就把这个函数需要什么样的参数定义好了,需要的就是一个 struct us_data *,那么很自然,传递的就是 us.

然后 790 行,scsi_host_alloc 就是 scsi 子系统提供的函数,它的作用就是申请一个 scsi host 相应的数据结构.不过我们要注意的是它的参数,尤其是第一个参数, &usb_stor_host_template,其实这是一个

struct scsi_host_template 的结构体指针,从这一刻开始,我们需要开始了解 drivers/usb/storage/scsiglue.c 这个文件了,glue 就是胶水的意思,与 scsi 相关联的代码我们就都准备在这个文件里了.

usb_stor_host_template 的定义就在 drivers/usb/storage/scsiglue.c 中,

```
416 /*
417  * this defines our host template, with which we'll allocate hosts
418  */
419
420 struct scsi_host_template usb_stor_host_template = {
421         /* basic userland interface stuff */
422         .name =                         "usb-storage",
423         .proc_name =                    "usb-storage",
424         .proc_info =                    proc_info,
425         .info =                         host_info,
426
427         /* command interface -- queued only */
428         .queuecommand =                 queuecommand,
429
430         /* error and abort handlers */
431         .eh_abort_handler =             command_abort,
432         .eh_device_reset_handler =      device_reset,
433         .eh_bus_reset_handler =         bus_reset,
434
435         /* queue commands only, only one command per LUN */
436         .can_queue =                    1,
437         .cmd_per_lun =                  1,
438
439         /* unknown initiator id */
440         .this_id =                      -1,
441
442         .slave_alloc =                  slave_alloc,
443         .slave_configure =              slave_configure,
444
445         /* lots of sg segments can be handled */
446         .sg_tablesize =                 SG_ALL,
447
448         /* limit the total size of a transfer to 120 KB */
449         .max_sectors =                  240,
450
451         /* merge commands... this seems to help performance, but
452          * periodically someone should test to see which setting is more
453          * optimal.
454          */
```

```
455         .use_clustering =               1,
456
456
457         /* emulated HBA */
458         .emulated =                      1,
459
460         /* we do our own delay after a device or bus reset */
461         .skip_settle_delay =            1,
462
463         /* sysfs device attributes */
464         .sdev_attrs =                   sysfs_device_attr_list,
465
466         /* module management */
467         .module =                        THIS_MODULE
468 };
```

如果您觉得眼前这个结构体变量的定义或者说初始化很复杂,那么您错了.因为下面您将看到一个更变态的数据结构的定义,她就是传说中的 struct scsi_host_template,她是 scsi 子系统定义的结构体,来自 include/scsi/scsi_host.h 文件:

```
40 struct scsi_host_template {
41         struct module *module;
42         const char *name;
43
44         /*
45          * Used to initialize old-style drivers.  For new-style drivers
46          * just perform all work in your module initialization function.
47          *
48          * Status:  OBSOLETE
49          */
50         int (* detect)(struct scsi_host_template *);
51
52         /*
53          * Used as unload callback for hosts with old-style drivers.
54          *
55          * Status: OBSOLETE
56          */
57         int (* release)(struct Scsi_Host *);
58
59         /*
60          * The info function will return whatever useful information the
61          * developer sees fit.  If not provided, then the name field will
62          * be used instead.
63          *
64          * Status: OPTIONAL
```

```
65          */
66         const char *(* info)(struct Scsi_Host *);
67
68         /*
69          * Ioctl interface
70          *
71          * Status: OPTIONAL
72          */
73         int (* ioctl)(struct scsi_device *dev, int cmd, void __user *arg);
74
75         /*
76          * The queuecommand function is used to queue up a scsi
77          * command block to the LLDD.  When the driver finished
78          * processing the command the done callback is invoked.
79          *
80          * If queuecommand returns 0, then the HBA has accepted the
81          * command.  The done() function must be called on the command
82          * when the driver has finished with it. (you may call done on the
83          * command before queuecommand returns, but in this case you
84          * *must* return 0 from queuecommand).
85          *
86          * Queuecommand may also reject the command, in which case it may
87          * not touch the command and must not call done() for it.
88          *
89          * There are two possible rejection returns:
90          *
91          *   SCSI_MLQUEUE_DEVICE_BUSY: Block this device temporarily, but
92          *   allow commands to other devices serviced by this host.
93          *
94          *   SCSI_MLQUEUE_HOST_BUSY: Block all devices served by this
95          *   host temporarily.
96          *
97          * For compatibility, any other non-zero return is treated the
98          * same as SCSI_MLQUEUE_HOST_BUSY.
99          *
100         * NOTE: "temporarily" means either until the next command for#
101         * this device/host completes, or a period of time determined by
102         * I/O pressure in the system if there are no other outstanding
103         * commands.
104         *
105         * STATUS: REQUIRED
106         */
107        int (* queuecommand)(struct scsi_cmnd *,
108                             void (*done)(struct scsi_cmnd *));
```

```
109
110         /*
111          * This is an error handling strategy routine.  You don't need to
112          * define one of these if you don't want to - there is a default
113          * routine that is present that should work in most cases.  For those
114          * driver authors that have the inclination and ability to write their
115          * own strategy routine, this is where it is specified.  Note - the
116          * strategy routine is *ALWAYS* run in the context of the kernel eh
117          * thread.  Thus you are guaranteed to *NOT* be in an interrupt
118          * handler when you execute this, and you are also guaranteed to
119          * *NOT* have any other commands being queued while you are in the
120          * strategy routine. When you return from this function, operations
121          * return to normal.
122          *
123          * See scsi_error.c scsi_unjam_host for additional comments about
124          * what this function should and should not be attempting to do.
125          *
126          * Status: REQUIRED      (at least one of them)
127          */
128         int (* eh_strategy_handler)(struct Scsi_Host *);
129         int (* eh_abort_handler)(struct scsi_cmnd *);
130         int (* eh_device_reset_handler)(struct scsi_cmnd *);
131         int (* eh_bus_reset_handler)(struct scsi_cmnd *);
132         int (* eh_host_reset_handler)(struct scsi_cmnd *);
133
134         /*
135          * This is an optional routine to notify the host that the scsi
136          * timer just fired.  The returns tell the timer routine what to
137          * do about this:
138          *
139          * EH_HANDLED:           I fixed the error, please complete the command
140          * EH_RESET_TIMER:       I need more time, reset the timer and
141          *                       begin counting again
142          * EH_NOT_HANDLED        Begin normal error recovery
143          *
144          * Status: OPTIONAL
145          */
146         enum scsi_eh_timer_return (* eh_timed_out)(struct scsi_cmnd *);
147
148         /*
149          * Before the mid layer attempts to scan for a new device where none
150          * currently exists, it will call this entry in your driver.  Should
151          * your driver need to allocate any structs or perform any other init
152          * items in order to send commands to a currently unused target/lun
```

```
153          * combo, then this is where you can perform those allocations.   This
154          * is specifically so that drivers won't have to perform any kind of
155          * "is this a new device" checks in their queuecommand routine,
156          * thereby making the hot path a bit quicker.
157          *
158          * Return values: 0 on success, non-0 on failure
159          *
160          * Deallocation:   If we didn't find any devices at this ID, you will
161          * get an immediate call to slave_destroy().   If we find something
162          * here then you will get a call to slave_configure(), then the
163          * device will be used for however long it is kept around, then when
164          * the device is removed from the system (or * possibly at reboot
165          * time), you will then get a call to slave_destroy().   This is
166          * assuming you implement slave_configure and slave_destroy.
167          * However, if you allocate memory and hang it off the device struct,
168          * then you must implement the slave_destroy() routine at a minimum
169          * in order to avoid leaking memory
170          * each time a device is tore down.
171          *
172          * Status: OPTIONAL
173          */
174         int (* slave_alloc)(struct scsi_device *);
175
176         /*
177          * Once the device has responded to an INQUIRY and we know the
178          * device is online, we call into the low level driver with the
179          * struct scsi_device *.   If the low level device driver implements
180          * this function, it *must* perform the task of setting the queue
181          * depth on the device.   All other tasks are optional and depend
182          * on what the driver supports and various implementation details.
183          *
184          * Things currently recommended to be handled at this time include:
185          *
186          * 1.   Setting the device queue depth.   Proper setting of this is
187          *      described in the comments for scsi_adjust_queue_depth.
188          * 2.   Determining if the device supports the various synchronous
189          *      negotiation protocols.   The device struct will already have
190          *      responded to INQUIRY and the results of the standard items
191          *      will have been shoved into the various device flag bits, eg.
192          *      device->sdtr will be true if the device supports SDTR messages.
193          * 3.   Allocating command structs that the device will need.
194          * 4.   Setting the default timeout on this device (if needed).
195          * 5.   Anything else the low level driver might want to do on a device
196          *      specific setup basis...
```

```
197          * 6.   Return 0 on success, non-0 on error.   The device will be marked
198          *      as offline on error so that no access will occur.   If you return
199          *      non-0, your slave_destroy routine will never get called for this
200          *      device, so don't leave any loose memory hanging around, clean
201          *      up after yourself before returning non-0
202          *
203          * Status: OPTIONAL
204          */
205         int (* slave_configure)(struct scsi_device *);
206
207         /*
208          * Immediately prior to deallocating the device and after all activity
209          * has ceased the mid layer calls this point so that the low level
210          * driver may completely detach itself from the scsi device and vice
211          * versa.   The low level driver is responsible for freeing any memory
212          * it allocated in the slave_alloc or slave_configure calls.
213          *
214          * Status: OPTIONAL
215          */
216         void (* slave_destroy)(struct scsi_device *);
217
218         /*
219          * This function determines the bios parameters for a given
220          * harddisk.   These tend to be numbers that are made up by
221          * the host adapter.   Parameters:
222          * size, device, list (heads, sectors, cylinders)
223          *
224          * Status: OPTIONAL
225          */
226         int (* bios_param)(struct scsi_device *, struct block_device *,
227                         sector_t, int []);
228
229         /*
230          * Can be used to export driver statistics and other infos to the
231          * world outside the kernel ie. userspace and it also provides an
232          * interface to feed the driver with information.
233          *
234          * Status: OBSOLETE
235          */
236         int (*proc_info)(struct Scsi_Host *, char *, char **, off_t, int, int);
237
238         /*
239          * Name of proc directory
240          */
```

```
241         char *proc_name;
242
243         /*
244          * Used to store the procfs directory if a driver implements the
245          * proc_info method.
246          */
247         struct proc_dir_entry *proc_dir;
248
249         /*
250          * This determines if we will use a non-interrupt driven
251          * or an interrupt driven scheme,  It is set to the maximum number
252          * of simultaneous commands a given host adapter will accept.
253          */
254         int can_queue;
255
256         /*
257          * In many instances, especially where disconnect / reconnect are
258          * supported, our host also has an ID on the SCSI bus.  If this is
259          * the case, then it must be reserved.  Please set this_id to -1 if
260          * your setup is in single initiator mode, and the host lacks an
261          * ID.
262          */
263         int this_id;
264
265         /*
266          * This determines the degree to which the host adapter is capable
267          * of scatter-gather.
268          */
269         unsigned short sg_tablesize;
270
271         /*
272          * If the host adapter has limitations beside segment count
273          */
274         unsigned short max_sectors;
275
276         /*
277          * dma scatter gather segment boundary limit. a segment crossing this
278          * boundary will be split in two.
279          */
280         unsigned long dma_boundary;
281
282         /*
283          * This specifies "machine infinity" for host templates which don't
284          * limit the transfer size.  Note this limit represents an absolute
```

```
285          * maximum, and may be over the transfer limits allowed for
286          * individual devices (e.g. 256 for SCSI-1)
287          */
288 #define SCSI_DEFAULT_MAX_SECTORS          1024
289
290         /*
291          * True if this host adapter can make good use of linked commands.
292          * This will allow more than one command to be queued to a given
293          * unit on a given host.  Set this to the maximum number of command
294          * blocks to be provided for each device.  Set this to 1 for one
295          * command block per lun, 2 for two, etc.  Do not set this to 0.
296          * You should make sure that the host adapter will do the right thing
297          * before you try setting this above 1.
298          */
299         short cmd_per_lun;
300
301         /*
302          * present contains counter indicating how many boards of this
303          * type were found when we did the scan.
304          */
305         unsigned char present;
306
307         /*
308          * true if this host adapter uses unchecked DMA onto an ISA bus.
309          */
310         unsigned unchecked_isa_dma:1;
311
312         /*
313          * true if this host adapter can make good use of clustering.
314          * I originally thought that if the tablesize was large that it
315          * was a waste of CPU cycles to prepare a cluster list, but
316          * it works out that the Buslogic is faster if you use a smaller
317          * number of segments (i.e. use clustering).  I guess it is
318          * inefficient.
319          */
320         unsigned use_clustering:1;
321
322         /*
323          * True for emulated SCSI host adapters (e.g. ATAPI)
324          */
325         unsigned emulated:1;
326
327         /*
328          * True if the low-level driver performs its own reset-settle delays.
```

```
329          */
330         unsigned skip_settle_delay:1;
331
332         /*
333          * Countdown for host blocking with no commands outstanding
334          */
335         unsigned int max_host_blocked;
336
337         /*
338          * Default value for the blocking.  If the queue is empty,
339          * host_blocked counts down in the request_fn until it restarts
340          * host operations as zero is reached.
341          *
342          * FIXME: This should probably be a value in the template
343          */
344 #define SCSI_DEFAULT_HOST_BLOCKED         7
345
346         /*
347          * Pointer to the sysfs class properties for this host, NULL terminated.
348          */
349         struct class_device_attribute **shost_attrs;
350
351         /*
352          * Pointer to the SCSI device properties for this host, NULL terminated.
353          */
354         struct device_attribute **sdev_attrs;
355
356         /*
357          * List of hosts per template.
358          *
359          * This is only for use by scsi_module.c for legacy templates.
360          * For these access to it is synchronized implicitly by
361          * module_init/module_exit.
362          */
363         struct list_head legacy_hosts;
364 };
```

简直难以置信,一个结构体的定义加上注释从第 40 行到第 364 行,写代码的同志辛苦了,定义出这么一个变态的数据结构来,咱们有理由相信写代码的那位或者那些人有很多是单身,因为那句"不在寂寞中恋爱,就在寂寞中变态"在他们身上体现的淋漓尽致.但是没办法,咱们还得继续讲,时光的背影如此悠悠,往日的岁月又上心头,此时不讲更待何时呢?您若是还没明白什么是温柔,还想好好感受雪花绽放的气候,那您就先歇会吧,咱们先往下走了.其他也不想说什么了,从这个结构体的名称,scsi_host_template 来看,也基本知道,她是一种模版,很多情况都会被用到.她有很多元素,不过咱们 mass storage 里边定义的变量 usb_stor_host_template 并没有初始化所有的元素,只是初始化了其中一部分.实际上,scsi_host_template 正如其名字一样,代表的是 scsi host,通俗一点说,一个 scsi 卡,或者专业一点说

scsi 适配器,或者更专业的 scsi 主机适配器,或者 scsi 主机控制器,scsi 卡可以有多个,而在 Linux 中,每一个 scsi 卡对应一个数据结构,Scsi_Host(而 Linux 中将通过使用一个 scsi_host_template 结构指针为参数的函数来为 Scsi_Host 初始化),但是有些 Scsi_Host 对应的并非是真实的 scsi 卡,硬件上并不存在,但仍然需要一个 Scsi_Host,比如咱们的 u 盘,因为她得被模拟成 scsi 设备,所以得为她准备一个 scsi 卡,虚拟的 scsi 卡.回忆刚才 cat /proc/scsi/scsi 的输出,有 U 盘的情况,显示的 scsi 信息多了最后这么一段,而其次呢,Host 号也多了一个,原来 Host 只有 scsi0,scsi1,而现在多出来一个 scsi3.而这个 scsi3 其实就是一个虚拟的 scsi 卡.所以,为了创建这么一个虚拟的 scsi 卡的数据结构,咱们在 drivers/usb/storage/scsiglue.c 中定义了结构体变量 struct scsi_host_template usb_stor_host_template.

实际上 Scsi_Host 也是一个变态的数据结构,咱们将在下节结合刚才提到的函数 scsi_host_alloc 来讲解这些变态的数据结构.晕了吧,没办法,在这么多复杂的数据结构面前,也许只有琼瑶阿姨最近给<<又见一帘幽梦>>中的设计的那段经典的连续 13 个我晕的对白才能表达我们此刻的心情吧.

Scsi 数据结构-像雾像雨又像风

关于 scsi,咱想说的是,在 Linux 内核中,整个 scsi 子系统被分为三层.upper level,mid level,lower level,也许您看到这心里很烦,Linux 为什么这么麻烦呢.就像某位大侠所说的:真不明白,女孩买很多很多漂亮衣服穿,就是为了吸引男孩的目光,但男孩想看的,却是不穿衣服的女孩.实际上,Linux 开发者们把 scsi 子系统包装成很多层,是为了给您提供方便,但是您看代码的时候却会感觉很烦,要是没有那么多层该多好.

来说说这三层吧,upper level,用伟大的汉语来讲,就是最上层,她是和操作系统打交道的,比如您要是有一块 scsi 硬盘,那么您就需要使用 sd_mod.o 这么一个模块,她实际上是与硬件无关的,是纯粹的软件上的抽象出来的数据结构组建的模块.mid level,中层,实际上这层才是真正的核心层,江湖上人称 scsi-core,即 scsi 核心层,她提供了支持 scsi 的核心数据结构和函数,这一层对应的模块是 scsi_mod.o,系统中要想使用 scsi 设备,首先必须加载她,反过来,只有所有的 scsi 设备的模块都被卸载了才能够卸载她.陆游就曾如此形容 scsi 核心层,无意苦争春,一任群芳妒.

然后,是 lower level,底层.很不幸,如果您要写驱动,八成就是写的底层,正如现实中的我们一样,生活在社会的最底层.因为 upper level 和 mid level 都已经基本上确定了,她们和硬件没关系.能留给您做的事情只能是底层.现实就是这样,尤其对 80 后的来说,生存的压力让 80 后不可能再如 70 后那样,能做的只有面对现实.

既然您需要关注的是底层,那么底层和中层或者说 scsi 核心层是如何打交道的呢?首先您得知道,核心层提供了很多函数供底层使用,其中上一节见到的 scsi_host_alloc()函数正是核心层提供的,后面还将遇到的一些函数也来自中层,对于 scsi 中层提供的函数,咱们不需要去关心她究竟如何实现的,只看看她的声明即可.在 include/scsi/scsi_host.h 中,有这么一行,

```
527 extern struct Scsi_Host *scsi_host_alloc(struct scsi_host_template *, int);
```

可以看到这个函数的参数有一个是 struct scsi_host_template 结构体的指针,而她将返回一个 struct Scsi_Host 结构体的指针,回想一下,前面咱们调用这个函数的那句,

```
790             us->host = scsi_host_alloc(&usb_stor_host_template, sizeof(us));
```

凭一种男人的直觉,可以猜出,这个函数申请了一个 Scsi_Host 结构体,并返回指向她的指针赋给 us->host.从高一点的角度来说,这么一句话实际上就是在 scsi 核心层注册了一个 scsi 卡,当然这里的 scsi 卡是虚拟的.顺便来看看 us->host.她实际上就是一个 struct Scsi_Host 结构体的指针,而 struct Scsi_Host 又是一个变态的数据结构,她的定义在 include/scsi/scsi_host.h 中:

```
376 struct Scsi_Host {
377         /*
378          * __devices is protected by the host_lock, but you should
379          * usually use scsi_device_lookup / shost_for_each_device
380          * to access it and don't care about locking yourself.
381          * In the rare case of beeing in irq context you can use
382          * their __ prefixed variants with the lock held. NEVER
383          * access this list directly from a driver.
384          */
385         struct list_head          __devices;
386
387         struct scsi_host_cmd_pool *cmd_pool;
388         spinlock_t                free_list_lock;
389         struct list_head          free_list; /* backup store of cmd structs */
390         struct list_head          starved_list;
391
392         spinlock_t                default_lock;
393         spinlock_t                *host_lock;
394
395         struct semaphore          scan_mutex;/* serialize scanning activity */
396
397         struct list_head          eh_cmd_q;
398         struct task_struct      * ehandler;   /* Error recovery thread. */
399         struct semaphore        * eh_wait;    /* The error recovery thread waits
400                                                  on this. */
401         struct completion       * eh_notify; /* wait for eh to begin or end */
402         struct semaphore        * eh_action; /* Wait for specific actions on the
403                                                 host. */
404         unsigned int              eh_active:1; /* Indicates the eh thread is awake
and active if
405                                                 this is true. */
406         unsigned int              eh_kill:1; /* set when killing the eh thread */
407         wait_queue_head_t         host_wait;
408         struct scsi_host_template *hostt;
409         struct scsi_transport_template *transportt;
410         volatile unsigned short host_busy;    /* commands actually active on
low-level */
411         volatile unsigned short host_failed; /* commands that failed. */
412
```

```
413         unsigned short host_no;   /* Used for IOCTL_GET_IDLUN, /proc/scsi et al.
*/
414         int resetting; /* if set, it means that last_reset is a valid value */
415         unsigned long last_reset;
416
417         /*
418          * These three parameters can be used to allow for wide scsi,
419          * and for host adapters that support multiple busses
420          * The first two should be set to 1 more than the actual max id
421          * or lun (i.e. 8 for normal systems).
422          */
423         unsigned int max_id;
424         unsigned int max_lun;
425         unsigned int max_channel;
426
427         /*
428          * This is a unique identifier that must be assigned so that we
429          * have some way of identifying each detected host adapter properly
430          * and uniquely.   For hosts that do not support more than one card
431          * in the system at one time, this does not need to be set.   It is
432          * initialized to 0 in scsi_register.
433          */
434         unsigned int unique_id;
435
436         /*
437          * The maximum length of SCSI commands that this host can accept.
438          * Probably 12 for most host adapters, but could be 16 for others.
439          * For drivers that don't set this field, a value of 12 is
440          * assumed.   I am leaving this as a number rather than a bit
441          * because you never know what subsequent SCSI standards might do
442          * (i.e. could there be a 20 byte or a 24-byte command a few years
443          * down the road?).
444          */
445         unsigned char max_cmd_len;
446
447         int this_id;
448         int can_queue;
449         short cmd_per_lun;
450         short unsigned int sg_tablesize;
451         short unsigned int max_sectors;
452         unsigned long dma_boundary;
453
454         unsigned unchecked_isa_dma:1;
455         unsigned use_clustering:1;
```

```
456         unsigned use_blk_tcq:1;
457
458         /*
459          * Host has requested that no further requests come through for the
460          * time being.
461          */
462         unsigned host_self_blocked:1;
463
464         /*
465          * Host uses correct SCSI ordering not PC ordering. The bit is
466          * set for the minority of drivers whose authors actually read
467          * the spec ;)
468          */
469         unsigned reverse_ordering:1;
470
471         /*
472          * Host has rejected a command because it was busy.
473          */
474         unsigned int host_blocked;
475
476         /*
477          * Value host_blocked counts down from
478          */
479         unsigned int max_host_blocked;
480
481         /* legacy crap */
482         unsigned long base;
483         unsigned long io_port;
484         unsigned char n_io_port;
485         unsigned char dma_channel;
486         unsigned int  irq;
487
488
489         unsigned long shost_state;
490
491         /* ldm bits */
492         struct device           shost_gendev;
493         struct class_device     shost_classdev;
494
495         /*
496          * List of hosts per template.
497          *
498          * This is only for use by scsi_module.c for legacy templates.
499          * For these access to it is synchronized implicitly by
```

```
500          * module_init/module_exit.
501          */
502         struct list_head sht_legacy_list;
503
504         /*
505          * Points to the transport data (if any) which is allocated
506          * separately
507          */
508         void *shost_data;
509         struct class_device transport_classdev;
510
511         /*
512          * We should ensure that this is aligned, both for better performance
513          * and also because some compilers (m68k) don't automatically force
514          * alignment to a long boundary.
515          */
516         unsigned long hostdata[0]  /* Used for storage of host specific stuff */
517                 __attribute__ ((aligned (sizeof(unsigned long))));
518 };
```

需要提一下最后这个元素,hostdata[0],相信您感兴趣的是__attribute__,这是 gcc 的关键字.她的作用是,描述函数,变量,类型的属性,很显然谭浩强大哥的 C 程序设计中是不会介绍这玩艺儿的,事实上大多数人也用不着知道她.不过在 gcc 里面,她出现的很频繁,因为她有利于代码优化和代码检查,特别是当咱们编写的是一个要在各种硬件平台上运行的操作系统的时候,这些属性是相当的有必要的.通常__attribute__(单词 attribute 前后各两个 underscore,即下划线.)出现在定义一个变量/函数/类型的时候,她紧跟在变量/函数/类型定义的后面,此处咱们看到的__attribute__是紧跟在 unsigned long hostdata[0]这样一个数组(汗,数组居然只有一个元素)后面,数组的每一个元素是一个 unsigned long 的类型变量,__attribute__后面的括号里则表明了属性,gcc 一共支持十几种属性,其中 aligned 是一种,此外还有一些常用的,比如 packed,noreturn,而 packed 其实咱们前面见过,只是没有提起.现在讲一下 aligned 属性和 packed 属性.这两个都是和字节对齐有关的属性,前些年像 intel 和 microsoft 这些外企笔试面试题目中常考察字节对齐的冬冬,所以相信很多人对字节对齐并不陌生.简而言之,字节对齐是这么一回事,变量存放在内存中本来是无所谓怎么放的,因为怎么放都能访问到,但是不同的硬件访问内存的方式不一样,只有把变量按特定的规则存放在内存中,存取效率才会高,否则不按规则的存放变量将有可能降低变量存取效率.很多情况下,人们并不关心字节对齐,因为通常这些事情由编译器来处理,编译器很冰雪,她知道针对什么平台该如何对齐.但有时候,咱们需要显式的去设定对齐方式,因为有时候咱们对编译器的所作所为并不满意,或者咱们自己觉得自己指定的方式会更好,比如此处,在定义 hostdata 之前的几行注释,已经很清楚地说明了为啥咱们要显式的去指定对齐方式.具体来说,unsigned long hostdata[0] __attribute__ ((aligned (sizeof(unsigned long))))就是表示 hostdata[0]将以 sizeof(unsigned long)字节对齐,显然不同的硬件平台 sizeof(unsinged long)是不一样的.而之前咱们遇到过的那个定义于 include/linux/usb_ch9.h 中的 struct usb_device_descriptor 以及 struct usb_interface_descriptor 结构体,则在最后跟了这么一句:__attribute__ ((packed)),这个表示使用最小可能的对齐,从 packed 的字面意思也很清楚,紧凑一点,别瞎留空间,实际上这也就是给编译器的一个命令,告诉编译器,嘿,一会节省一点,别浪费空间啊.

好,介绍完 Scsi_Host 数据结构,咱们继续回到那曾经的 usb_stor_acquire_resources()函数中来,us->host 得到了她想要的,然后下面 798 行,只是一句赋值,把 us->host->hostdata[0]赋值为(unsigned long)us,这样做有什么用咱们后面遇到了再说.

总之,scsi_host_alloc()这么一调用,就是给咱们的 struct Scsi_Host 结构提申请了空间,真正要想模拟一个 scsi 的情景,需要三个函数,scsi_host_alloc(),scsi_add_host(),scsi_scan_host().只有调用了第二个函数之后,scsi 核心层才知道有这么一个 host 的存在,而只有第三个函数被调用了之后,真正的设备才被发现.这些函数的含义和它们的名字吻合的很好.不是吗?

最后需要指出的是,scsi_host_alloc()需要两个参数,第一个参数是 struct scsi_host_template 的指针,咱们当然给了它&usb_stor_host_template,而第二个参数实际上是被称为 driver 自己的数据,咱们传递的是 sizeof(us).这样子,scsi_host_alloc()中就会给咱们申请内存空间,即为 us 申请内存空间.不过有趣的是,us 我们早就申请好了空间,这里多申请一份是否有必要呢?注意到 struct Scsi_Host 里边不是有一个 hostdata 么,理由是这样的,struct us_data 这个冬冬是我们在 usb-storage 模块里边专门定义的,而一会我们和 scsi 层打交道,我们要注册一些函数,提供给 scsi 核心层,让核心层去调用,这些函数原型如何是 scsi 层说了算的,scsi 层准备了一些函数指针,我们只是把这些指针赋好值,scsi 层就知道在什么时候该调用哪个函数了.所以既然原型是人家提供的,那么人家肯定不知道我们会有一个 struct us_data 这么一个结构体,所以我们在定义函数的时候就不能把 struct us_data 当作一个参数,但是我们专门为自己这个模块准备的结构体又不可能不用,那怎么办?好办,scsi 核心层是认 struct Scsi_Host 这个结构体的,而这个结构体在设计的时候就专门准备了一个 unsigned long hostdata[0]来给别的设备驱动使用.那我们还客气什么,让这个指针指向我们的 us 就可以了.以后要用 us 再通过 hostdata[0]来获得就是了.所以刚才我们看到了 us->host->hostdata[0]被赋值为(unsigned long)us.这个 host 就是我们之后会一直用到的 struct Scsi_Host 结构体,hostdata[0]是一个 unsigned long 的变量,us 是一个指针,所以这就使得这个变量的值等于这个指针指向的地址.而之后我们会经常看见 scsiglue.c 中的函数里边使用如下的赋值语句:

```
struct us_data *us = (struct us_data *) host->hostdata[0];
```

所以意思就很明确了.指针被定义为原来 us 所指向的那个地址.所以看似又定义了一个 struct us_data *us,实际上只不过是原来是一个全局变量,现在是一个局部变量而已.至于在 scsi_host_alloc()中又申请了一段大小为 sizeof(us)的内存,既然已经有内存了就没有必要用它了,只是借用 hostdata 这个指针而已.当然,这样做不太合理,浪费内存,与我党长期以来坚持的艰苦朴素的作风是相违背的,在我党如今强调以艰苦奋斗为荣,以骄奢淫逸为耻的背景下,这段代码自然遭到了广大人民群众的质疑,因此毫无疑问,在最新版本的 Linux 内核中这段代码被修改了(用今天流行的话说就是被和谐掉了).有兴趣的同志们可以关注一下 2.6.22 版的内核.

最后的最后,补充一点,为什么 scsi_host_alloc 传递进来的是一个 struct scsi_host_template 指针,而返回的确是一个 struct Scsi_Host 指针?首先,这两个结构体包含很多相同的元素,但又不完全相同,它们协同工作,互相关联,但是各自起的作用不一样,struct Scsi_Host 结构体中有一个成员,就是一个 struct scsi_host_template 指针,它就指向和它相关联的那个 template,而 struct scsi_host_template 中很多个函数指针,它们的参数就是 struct Scsi_Host 的指针.所以这之间的关系千丝万缕,剪不断理还乱,藕断丝还连.一点不亚于曹禺先生的<<雷雨>>中那复杂的人物关系,唯一的差别只是这里没有乱伦关系罢了.好了,就这些,这种复杂的关系我们不需要去完全了解,我们要做的只是知道今后我们有且仅有一个 struct Scsi_Host 有且仅有一个 struct scsi_host_template 就可以了.

至此,我们终于走到了 usb_stor_acquire_resources()中第 801 行,即将见到这个千呼万唤始出来的内核精灵.

彼岸花的传说(一)

彼岸花,花语是悲伤的回忆.

很久很久以前,城市的边缘开满了大片大片的曼珠沙华,它的花香有一种魔力,可以让人想起自己前世的事情.守护曼珠沙华的是两个妖精,一个是花妖叫曼珠,一个是叶妖叫沙华.他们守侯了几千年,可是从来没有见过面,因为开花的时候,就没有叶子,有叶子的时候没有花.他们疯狂地想念着彼此,并被这种痛苦折磨着.终于有一天,他们决定违背神的规定偷偷地见一次面.那一年的曼珠沙华红艳艳的花被惹眼的绿色衬托着,开得格外妖冶美丽.

曼珠和沙华受到惩罚,被打入轮回,并被诅咒永远也不能在一起,生生世世在人世间受到磨难.从那以后,蔓珠沙华又叫彼岸花,意思是开放在天国的花,它的花的形状像一只只在向天堂祈祷的手掌,可是再也没有在这个城市出现过.每年的秋彼岸期间(春分前后三天叫春彼岸,秋分前后三天叫秋彼岸)她会开在黄泉路上,曼珠和沙华的每一次转世在黄泉路上闻到彼岸花的香味就能想起前世的的自己,然后发誓不分开,但只有在这一刻,因为他们会再次跌入诅咒的轮回,灵魂籍由着这花的指引,走向幽冥……

而为了纪念这个美丽而又忧伤的传说,Linux 内核中引入了守护进程,也正是与这个传说对应,守护进程也叫内核精灵,当然,如果你觉得人们都太迷信,而你是共产党员,是无神论者,那么 ok,你可以叫它为内核线程.事实上我也是无神论者,不过我只敢在白天承认这一点.我们来看具体的代码.

801 行,调用了 kernel_thread()函数,kernel_thread(usb_stor_control_thread, us, CLONE_VM),并将返回值用一个整型变量 p 来表示,如果从前您对内核本身不是很熟悉,那这个函数就会让您有点头疼了.这个函数将会创建一个内核线程,而函数 usb_stor_control_thread()将会执行,us 将是传递给她的参数,CLONE_VM 只是设定的一个 flag,对 Linux 内核不是很熟悉的话,可以将 kernel_thread 看作类似于 fork 的冬冬,什么?fork 您也不知道?好吧.不知道就不知道,只是依稀记得在那毕业求职的日子里,很多外企的笔试面试中都会问起 fork,问一些关于她的返回值的问题,印象中,2004 年维尔软件的校园招聘笔试中考过,后来我去 Sun 中国工程研究院面试也被问起过. 实际上,简单一点说,kernel_thread()这么一执行呢,就会有两个进程,一个是父进程,一个是子进程,子进程将会执行 usb_stor_control_thread(),而 us 是作为 usb_stor_control_thread 函数的参数(实参),CLONE_VM 标志表征父子进程之间共享地址空间,执行完 usb_stor_control_thread()之后,子进程就结束了,她会调用 exit()函数退出.而父进程继续顺着 usb_stor_acquire_resources()函数往下走,kernel_thread()函数对于父进程而言返回的是子进程的进程 id,所以 802 行先判断,若是返回值 p 小于 0,则说明出错了,否则,那就把 p 赋给 us 的元素 pid.

于是,咱们接下来必须再次兵分两路,分别跟踪父进程和子进程前进了.先看父进程,810 行,执行 wait_for_completion(&(us->notify)),如果您说 wait_for_completion 是谁?您早已经把她忘怀,那么她会很伤心的.记得咱们当初在提交 urb 的时候调用过她吗?咱们初始化了一个等待队列,urb->done,然后咱们提交了 urb,然后咱们通过调用 wait_for_completion(&urb->done)进入了等待,进入了睡眠,当 urb 被 usb core 处理完了,或者说被 urb host controller 处理完了,complete()函数将会被调用,从而唤醒咱们那个进入睡眠的进程.所以此处也一样,us->notify 是在很早很早以前 storage_probe()函数中初始化的,该函数的 945 行,init_completion(&(us->notify)),而 struct us_data 中,有这么一个元素 notify,是 struct completion 结构体变量,所以在这里,父进程将进入睡眠,谁来唤醒他?没错,您很聪明,正是子进

程,也正是在 usb_stor_control_thread()函数中,会有 complete(&(us->notify))这么一句,wait_for_completion()和 complete()是一对青梅竹马的函数,她们总是一起被用,一个设置进程进入睡眠,另一个则负责把这个进入睡眠的进程唤醒,如果您知道信号量的话,她们就类似于信号量中的 down()/up()函数对,只不过她们更加安全.

好,父进程先搁一搁,反正他进入睡眠了,没人唤醒他的话,他是不会往下走的.来看子进程,也就是 usb_stor_control_thread()函数,这个函数定义于 drivers/usb/storage/usb.c 中.咱们下一节再深入这个函数,现在,休息,休息一会...

彼岸花的传说(二)

如果让观众短信投票的话,usb_stor_control_thread()这个函数中的代码无疑是整个模块中最为精华的代码.我们只需要它中间 301 行那个 for(;;)就知道,这是一个死循环,即使别的代码都执行完了,即使别的函数都退出了,这个函数仍然像永不消逝的电波一般,经典常驻.显然,只有死循环才能代码永恒.才能代表忠诚.这是每一个守护者的职责.

usb_stor_control_thread(),其代码如下:

```
281 static int usb_stor_control_thread(void * __us)
282 {
283         struct us_data *us = (struct us_data *)__us;
284         struct Scsi_Host *host = us->host;
285
286         lock_kernel();
287
288         /*
289          * This thread doesn't need any user-level access,
290          * so get rid of all our resources.
291          */
292         daemonize("usb-storage");
293
294         current->flags |= PF_NOFREEZE;
295
296         unlock_kernel();
297
298         /* signal that we've started the thread */
299         complete(&(us->notify));
300
301         for(;;) {
302                 US_DEBUGP("*** thread sleeping.\n");
303                 if(down_interruptible(&us->sema))
304                         break;
305
306                 US_DEBUGP("*** thread awakened.\n");
```

```
307
308             /* lock the device pointers */
309             down(&(us->dev_semaphore));
310
311             /* if us->srb is NULL, we are being asked to exit */
312             if (us->srb == NULL) {
313                     US_DEBUGP("-- exit command received\n");
314                     up(&(us->dev_semaphore));
315                     break;
316             }
317
318             /* lock access to the state */
319             scsi_lock(host);
320
321             /* has the command timed out *already* ? */
322             if (test_bit(US_FLIDX_TIMED_OUT, &us->flags)) {
323                     us->srb->result = DID_ABORT << 16;
324                     goto SkipForAbort;
325             }
326
327             /* don't do anything if we are disconnecting */
328             if (test_bit(US_FLIDX_DISCONNECTING, &us->flags)) {
329                     US_DEBUGP("No command during disconnect\n");
330                     goto SkipForDisconnect;
331             }
332
333             scsi_unlock(host);
334
335             /* reject the command if the direction indicator
336              * is UNKNOWN
337              */
338             if (us->srb->sc_data_direction == DMA_BIDIRECTIONAL) {
339                     US_DEBUGP("UNKNOWN data direction\n");
340                     us->srb->result = DID_ERROR << 16;
341             }
342
343             /* reject if target != 0 or if LUN is higher than
344              * the maximum known LUN
345              */
346             else if (us->srb->device->id &&
347                             !(us->flags & US_FL_SCM_MULT_TARG)) {
348                     US_DEBUGP("Bad target number (%d:%d)\n",
349                               us->srb->device->id, us->srb->device->lun);
350                     us->srb->result = DID_BAD_TARGET << 16;
```

```
351                 }
352
353                 else if (us->srb->device->lun > us->max_lun) {
354                         US_DEBUGP("Bad LUN (%d:%d)\n",
355                                 us->srb->device->id, us->srb->device->lun);
356                         us->srb->result = DID_BAD_TARGET << 16;
357                 }
358
359                 /* Handle those devices which need us to fake
360                  * their inquiry data */
361                 else if ((us->srb->cmnd[0] == INQUIRY) &&
362                             (us->flags & US_FL_FIX_INQUIRY)) {
363                         unsigned char data_ptr[36] = {
364                             0x00, 0x80, 0x02, 0x02,
365                             0x1F, 0x00, 0x00, 0x00};
366
367                         US_DEBUGP("Faking INQUIRY command\n");
368                         fill_inquiry_response(us, data_ptr, 36);
369                         us->srb->result = SAM_STAT_GOOD;
370                 }
371
372                 /* we've got a command, let's do it! */
373                 else {
374                         US_DEBUG(usb_stor_show_command(us->srb));
375                         us->proto_handler(us->srb, us);
376                 }
377
378                 /* lock access to the state */
379                 scsi_lock(host);
380
381                 /* indicate that the command is done */
382                 if (us->srb->result != DID_ABORT << 16) {
383                         US_DEBUGP("scsi cmd done, result=0x%x\n",
384                                    us->srb->result);
385                         us->srb->scsi_done(us->srb);
386                 } else {
387 SkipForAbort:
388                         US_DEBUGP("scsi command aborted\n");
389                 }
390
391                 /* If an abort request was received we need to signal that
392                  * the abort has finished.   The proper test for this is
393                  * the TIMED_OUT flag, not srb->result == DID_ABORT, because
394                  * a timeout/abort request might be received after all the
```

```
395                      * USB processing was complete. */
396                     if (test_bit(US_FLIDX_TIMED_OUT, &us->flags))
397                             complete(&(us->notify));
398
399                     /* finished working on this command */
400 SkipForDisconnect:
401                     us->srb = NULL;
402                     scsi_unlock(host);
403
404                     /* unlock the device pointers */
405                     up(&(us->dev_semaphore));
406             } /* for (;;) */
407
408             /* notify the exit routine that we're actually exiting now
409              *
410              * complete()/wait_for_completion() is similar to up()/down(),
411              * except that complete() is safe in the case where the structure
412              * is getting deleted in a parallel mode of execution (i.e. just
413              * after the down() -- that's necessary for the thread-shutdown
414              * case.
415              *
416              * complete_and_exit() goes even further than this -- it is safe in
417              * the case that the thread of the caller is going away (not just
418              * the structure) -- this is necessary for the module-remove case.
419              * This is important in preemption kernels, which transfer the flow
420              * of execution immediately upon a complete().
421              */
422             complete_and_exit(&(us->notify), 0);
423 }
```

284 行,定义了一个 Scsi_Host 的指针 host,令她指向 us->host,也就是刚刚用 scsi_host_alloc()申请的那个 Scsi_Host 结构体变量.

292 行,daemonize("usb-storage"),其实,这句话才是真正创建精灵进程的,daemonize()函数来自内核的核心位置,kernel/exit.c 中,她完成了这么一件事情,把一个普通的进程转换成为了精灵进程,不过此处咱们可以不去深究精灵进程的原理,甚至咱们可以认为这句话没有做任何事情,只是从此之后咱们 ps 命令一看能够看到有一个叫做 usb-storage 的进程.比如下面所看到的:

```
localhost:~ # ps -el
F S   UID   PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
4 S     0     1     0  0  76   0 -   195 -      ?        00:00:02 init
1 S     0     2     1  0 -40   - -     0 migrat ?        00:00:00 migration/0
1 S     0     3     1  0  94  19 -     0 ksofti ?        00:00:00 ksoftirqd/0
1 S     0    18     1  0  70  -5 -     0 worker ?        00:00:00 events/0
1 S     0    26     1  0  71  -5 -     0 worker ?        00:00:00 khelper
```

```
1 S     0    27     1   0  70  -5 -      0 worker ?        00:00:00 kthread
1 S     0    45    27   0  72  -5 -      0 worker ?        00:00:00 kacpid
1 S     0   229    27   0  80   0 -      0 pdflus ?        00:00:00 pdflush
1 S     0   230    27   0  75   0 -      0 pdflus ?        00:00:00 pdflush
1 S     0   231     1   0  76   0 -      0 kswapd ?        00:00:00 kswapd0
1 S     0   961    27   0  70  -5 -      0 scsi_e ?        00:00:00 scsi_eh_0
1 S     0  1033    27   0  70  -5 -      0 scsi_e ?        00:00:00 scsi_eh_1
1 S     0  1045    27   0  71  -5 -      0 scsi_e ?        00:00:00 scsi_eh_2
1 S     0  1047    27   0  70  -5 -      0 worker ?        00:00:00 scsi_wq_2
5 S     0  1262     1   0  72  -4 -   1774 -       ?        00:00:02 udevd
1 S     0  1939    27   0  70  -5 -      0 hub_th ?        00:00:00 khubd
1 S     0  7804    27   0  70  -5 -      0 scsi_e ?        00:00:00 scsi_eh_3
1 S     0  7805    27   0  70  -5 -      0 -       ?       00:00:00 usb-storage
4 S     0 13905 13902   0  75   0 -   2430 wait    pts/1    00:00:00 bash
0 R     0 19098 13905   0  77   0 -    821 -       pts/1    00:00:00 ps
```

显然,您在终端按 ctrl-c 是不可能中止这个 usb-storage 进程的.这是精灵进程诸多特性中的一个,她运行于后台.试一下 kill -9 加进程号,看你能杀死她不?(系统崩溃了我可不负责哦...)

294 行,这里为目前的进程设置一个 flag,PF_NOFREEZE,在整个内核代码中,这个 flag 也只出现过几次.这个 flag 是与电源管理相关的,2.6 的内核为了实现与 Windows 相似的一个功能,Hibernate,也就是"冬眠",(别说您不知道,Windows 关机选项里面有"关机","重启","注销","Stand by",以及"Hibernate").在内核编译菜单里面,Power managerment options 中,有一个选项 Software Suspend,也就是内核编译选项中的 CONFIG_SOFTWARE_SUSPEND,选择了她使得机器可以被 suspended.显然咱们不用 care 她. 但是这里之所以要设置这个 flag,是因为 suspend 要求把内存里的冬冬写到磁盘上,而一个进程设置了这个 flag 就表明它在 suspend 的时候不会被冻住,用行话来讲就是,they're not refrigerated during a suspend.freeze 就是冷冻,冻住的意思,过去分词 frozen 是形容词,冻结的,冰冻的,其实就是让进程睡眠.所以总的来说,这里的做法就是说,即使系统 suspend 了,这个进程,或者准确地说,这个内核线程,也不应该进入睡眠.

要执行这两个操作需要执行 lock_kernel()/unlock_kernel()这一对函数.然后 299 行,执行 complete 唤醒前面那节的父进程.而子进程并不退出,她继续行走,她无怨无悔的行走,只不过余秋雨先生是为追寻人类文明足迹而进行的域外旅程,而此处子进程(执)行的她对内核的守护.她像天使一般,守护着心爱的人.

于是咱们也继续跟着她行走,299 行, complete(&(us->notify)),这正是和刚才在父进程里看到的那句使进程进入睡眠的 wait_for_completion(&(us->notify))相对应,这里自然是唤醒父进程,我们先继续看一下子进程,稍候马上去看父进程.

301 行,一个 for 语句死循环,尽管外面的世界很精彩,但是咱们去看看 for 里面的世界也不妨.

303 行,down_interruptible()函数,事实上 302 行的注释已经告诉咱们,thread 将进入睡眠了...,也许她累了.down_interruptible 的参数是&us->sema,不陌生吧,我们之前讲信号量讲互斥锁的时候就已经提过了 us->sema.所以这里很简单,就是想获得这把锁,但是别忘了,我们当初就介绍过,这把锁一开始就被初始化为 0 了,也就是说它属于那种指腹为婚的情形,一到这个世界来就告诉别人自己已经是名花有主了.因

此,这里只能进入睡眠,等待一个 up()函数去释放锁.谁会调用 up()函数呢?暂时先不管它,我们先关注一下父进程,毕竟我们自己进入了睡眠,而之前我们把父进程唤醒了.

彼岸花的传说(三)

遥想公瑾当年,小乔出嫁了,雄姿英发.
羽扇纶巾,谈笑间,樯橹灰飞烟灭.
故国神游,多情应笑我,早生华发,
人生如梦,一樽还酹江月.

的确,人生如梦,设计 Linux 代码的人想必非常认可这种观点,因为他们已然把这种思想融入到了代码中去,所以在代码里我们常看到睡眠,唤醒,睡眠,唤醒...而作为当年的大学生,Linus 想必也很认同大学生活的现状,即大学生活就是睡觉,只是有的人两个人睡,有的人一个人睡.

前面已经说了,父进程在函数 usb_stor_acquire_resources()里边,810 行,wait_for_completion(&(us->notify)),进入睡眠,而刚才咱们在子进程里已经看到,complete(&(us->notify))被调用,于是父进程被唤醒,回到 usb_stor_acquire_resources()函数中,往下走,812 行,无它,唯返回耳.返回了 0.于是咱们终于回到了 storage_probe()函数中来.

1001 行,scsi_add_host()函数被执行,之前申请的 us->host 被作为参数传递给她,同时,intf->dev 也被传递给她,这个冬冬是被用来注册 sysfs 的.在 scsi_host_alloc 之后,必须执行 scsi_add_host(),这样,scsi 核心层才能够知道有这么一个 host 存在.scsi_add_host()成功则返回 0,否则返回出错代码.如果一切顺利,咱们将走到 1009 行,别急,先把代码贴出来,这就是 storage_probe()函数的最后一小段了:

```
1007
1008            /* Start up the thread for delayed SCSI-device scanning */
1009            result = kernel_thread(usb_stor_scan_thread, us, CLONE_VM);
1010            if (result < 0) {
1011                    printk(KERN_WARNING USB_STORAGE
1012                            "Unable to start the device-scanning thread\n");
1013                    scsi_remove_host(us->host);
1014                    goto BadDevice;
1015            }
1016
1017            return 0;
1018
1019            /* We come here if there are any problems */
1020 BadDevice:
1021            US_DEBUGP("storage_probe() failed\n");
1022            usb_stor_release_resources(us);
1023            dissociate_dev(us);
1024            return result;
1025 }
1026
```

又一次见到了 kernel_thread,不需要更多解释,这里自然还是创建一个内核守护进程,只不过这次是 usb_stor_scan_thread,而上次是 usb_stor_control_thread.usb_stor_scan_thread()函数也是定义于 drivers/usb/storage/usb.c 中:

```
886 /* Thread to carry out delayed SCSI-device scanning */
887 static int usb_stor_scan_thread(void * __us)
888 {
889         struct us_data *us = (struct us_data *)__us;
890
891         /*
892          * This thread doesn't need any user-level access,
893          * so get rid of all our resources.
894          */
895         lock_kernel();
896         daemonize("usb-stor-scan");
897         unlock_kernel();
898
899         printk(KERN_DEBUG
900                 "usb-storage: device found at %d\n", us->pusb_dev->devnum);
901
902         /* Wait for the timeout to expire or for a disconnect */
903         if (delay_use > 0) {
904                 printk(KERN_DEBUG "usb-storage: waiting for device "
905                                 "to settle before scanning\n");
906 retry:
907                 wait_event_interruptible_timeout(us->scsi_scan_wait,
908                                 test_bit(US_FLIDX_DISCONNECTING, &us->flags),
909                                 delay_use * HZ);
910                 if (current->flags & PF_FREEZE) {
911                         refrigerator(PF_FREEZE);
912                         goto retry;
913                 }
914         }
915
916         /* If the device is still connected, perform the scanning */
917         if (!test_bit(US_FLIDX_DISCONNECTING, &us->flags)) {
918                 scsi_scan_host(us->host);
919                 printk(KERN_DEBUG "usb-storage: device scan complete\n");
920         }
921
922         complete_and_exit(&us->scsi_scan_done, 0);
923 }
```

显而易见,这里的 daemonize()又会让咱们再 ps -el 的命令输出中看到有 usb-stor-scan 这么一个精灵进程.

903 行,delay_use 哪来的?同一文件中,最开始的地方,定义了一个静态变量,

```
101 static unsigned int delay_use = 5;
102 module_param(delay_use, uint, S_IRUGO | S_IWUSR);
103 MODULE_PARM_DESC(delay_use, "seconds to delay before using a new device");
```

设置了 delay_use 为 5,而 module_param 是 LK 2.6 提供的一个宏,使得 delay_use 可以在模块被装载的时候设定.(如果不设,那么她自然就是这里的值 5,表示使用一个新的设备之前等待 5 秒延时.)为啥要 delay 啊?不为啥,只是因为天上的星星会流泪,地上的玫瑰会枯萎.您插进去的 u 盘也可能立刻又被您拔出来了,试想您插入以后一两秒之内又拔出来,那么咱们下面也不用耽误工夫再检测了嘛不是.

903 行,判断 delay_use>0,然后 907 行,wait_event_interruptible_timeout(),它的第一个参数是 us->scsi_scan_wait,请回答,us->scsi_scan_wait 是第一次露面吗?

混迹外企这么多日子, 我可以很负责任的很有诚意的用英语语重心长的告诉你: The answer is no. 在 storage_probe()函数的最初,在 us 的初始化时,scsi_scan_wait 被初始化了.947 行,init_waitqueue_head(&us->scsi_scan_wait),而在定义 struct us_data 时,有一个成员就是 scsi_scan_wait,即 wait_queue_head_t scsi_scan_wait,这些都是什么意思呢?

实际上 wait_event_interruptible_timeout()是一个宏,它代表着 Linux 中的一种等待机制,等待某个事件的发生,函数原型中,第一个参数是一个等待队列头,即 wait_queue_head_t 定义的变量,在 LK 2.6 中使用 init_waitqueue_head()函数初始化这个等待队列,然后第三个参数是设置超时,比如咱们这里设了 5s,这表示如果 5 秒到了,那么函数会返回 0,不管其它条件如何.第二个参数是一种等待的条件,或者说等待的事件,如果条件满足了,那么函数也会返回,条件要是不满足,那么这个进程会进入睡眠,不过 interruptible 表明了信号可以把她中断.一旦进入睡眠,那么有三种情况,一种是 wake_up 或者 wake_up_interruptible 函数被另一个进程执行,从而唤醒她,第二种是信号中断她,第三种就是刚才讲的超时,时间到了,自然就会返回.

那么这里具体来说,先判断 US_FLIDX_DISCONNECTING 这个 flag 有没有设,如果没有设才进入睡眠,否则就不需要浪费彼此的感情了.在进入睡眠之后,如果 5 秒之内您没有把 u 盘拔出来,那么 5 秒一到,函数返回 0,继续往下走,如果您在 5 秒之前拔出来了,那么后来咱们会讲,storage_disconnect()函数会执行,她会设置 US_FLIDX_DISCONNECTING 这个 flag,并且她会调用 wake_up(&us->scsi_scan_wait)来唤醒这里睡眠的进程,告诉他:"别等了,哥们儿,你没那种命!"这样函数就会提前返回,不用等到 5 秒再返回了.总之不管条件满不满足,5 秒之内肯定会返回,所以我们继续往下看.

910 行,看到 PF_FREEZE 这个标记,有点眼熟,不过上次咱们看到的是 PF_NOFREEZE,这些都是电源管理的内容,无需理会.对于 usb 设备,它可以进入 suspend 状态,那么如果设置了这个 flag,这里就让它进入吧,于是有人问了,这个 flag 我们从没设过啊,问题是这个 flag 不一定非得是我们来设,要知道这里可能等待了一段时间的,(delay_use),既然等待了一段时间,就有可能被认为是要进入睡眠,总线上就不会有活动,于是上头以为每人要理睬这个 usb 设备了,于是就会为它设置进入 suspend 的 flag.特别的情况,你小子吃错药了没事做,把 delay_use 给设成 1 光年,什么?光年不是时间单位?那好吧,那假设你把 delay_use 设成杨过喜欢的十六年,那你说这个设备是不是该进入 suspend 状态?或者说是不是该进入休眠状态?如果是,那么调用内核提供的函数, refrigerator()去休眠.

917 行,再次判断设备有没有被断开,如果还是没有,那么执行 scsi_scan_host()函数扫描,扫描然后就知道这个 host 或者说这个 scsi 卡上面接了什么设备(虽然咱们这个只是模拟的 scsi 卡),然后 cat /proc/scsi/scsi 才能看到您的 u 盘嘛不是.(scsi_scan_host()这里面发生了很多很多的故事,但这都是 scsi 那层的故事,咱们暂时先不去关注好不?)

然后 922 行,complete_and_exit 函数,她和 complete 函数还有一点点不一样,除了唤醒别人,还得结束自己(exit).她的户口在 kernel/exit.c 中:

```
842 NORET_TYPE void complete_and_exit(struct completion *comp, long code)
843 {
844         if (comp)
845                 complete(comp);
846
847         do_exit(code);
848 }
```

所以这里会先检查&us->scsi_scan_done,在 struct us_data 中,定义了一个成员 struct completion scsi_scan_done,而在 storage_probe()函数一开始为 us 进行初始化的时候,调用了 init_completion(&us->scsi_scan_done)为 us->scsi_scan_done 进行了初始化,如果此前 storage_disconnect()函数被调用了,那么她会调用 wait_for_completion(&us->scsi_scan_done)来进入睡眠并且等待咱们这里把她唤醒,此乃后话,暂且不表.待到分析 storage_disconnect 的时刻再来揭晓.但是如果根本就没有人调用 wait_for_completion(&scsi_scan_done),那么也就是说没有人睡眠,那么这个 complete 函数就什么也不做.然后 do_exit()函数不用多说,内核提供的函数,结束进程.也就是说,对于上面这个 scan 的精灵进程,到这里她就会结束退出了.可以看出它是一个短命的守护进程,真的不知该说天妒英才又还是该说红颜命薄呢.总之对于这个精灵进程来说,她的使命就是让你能在 cat /proc/scsi/scsi 中看到你的 U 盘,当然了,从此以后你在/dev 目录下面也就能看到你的设备了,比如/dev/sda.

再来看父进程,也就是 storage_probe(),在用 kernel_thread()创建了 usb_stor_scan_thread 之后,一切正常的话,storage_probe()也走到了尽头了.1017 行,return 0 了.终于,这个不老的传说也终于到了老的那一刻,一切都结束了,一切都烟消云散了.在这世上所有的事情都必须有个结束.--席慕蓉

彼岸花的传说(四)

江姐问:国民党推翻了么? 答:被阿扁推翻了,大家都成了好朋友.
董存瑞问:劳动人民还当牛做马么?答:都下岗了,不劳动了.
红色娘子军吴琼花来电话问:姐妹们都翻身得解放了吧? 答:思想解放了,都当小姐了.
杨子荣来电话问:土匪都剿灭了吧?答:都当公安了.
杨白劳来电话问:地主们都打倒了么? 答:都入党了.
马克思来电话问:资本家都消灭了么? 答:都进中央了.
上帝打电话问:看 Linux 那些事儿的人都回帖了吗? 答:没回的都在去见你的路上.

如果你把上面这些对话写成程序,烧录到芯片上,那么当外界询问设备的时候,设备就知道该怎么回答.如果你问姐妹们都翻身得解放了吧?设备就能回答你:思想解放了,都当小姐了.这就是 scsi 设备的基本工作方式.scsi 协议定义了一系列的命令.主机通过发送命令给设备来实现通信.

我们刚刚跟着 storage_probe()几乎完整的走了一遍,貌似一切都该结束了,可是你不觉得你到目前为止还根本没有看明白设备究竟怎么工作的吗?U 盘,不仅仅是 usb 设备,还是"盘",它还需遵守 usb mass storage 协议,还需遵守 transparent scsi 规范.从驱动程序的角度来看,它和一般的 scsi 磁盘差不多.正是因为如此,所以 U 盘的工作真正需要的是四个模块,usbcore,scsi_mod,sd_mod,以及咱们这里的 usb-storage,其中 sd_mod 恰恰就是 scsi disk 的驱动程序.没有它,你的 scsi 硬盘就别想在 Linux 下面转起来.看看它的代码,你就可以去电脑报投稿,发一篇"玩转 Linux 下的 scsi 硬盘"这样的文章.

那么我们从哪里开始去接触这些 scsi 命令呢?别忘了我们现在的主题,内核守护进程,别忘了我们曾经有一段代码只讲到一半就没讲了.没错,那就是 usb_stor_control_thread(),当初我们用 kernel_thread 创建它的时候就说了,从此以后一个进程变成两个进程,而我们刚才沿着 storage_probe 讲完的是父进程,父进程最终返回了,而子进程则没有那么简单,我们已经说过,usb_stor_control_thread()中的死循环注定了这个子进程是一个永恒的进程,只要这个模块还没有被卸载,或者说还没有被要求卸载,这个子进程就必将永垂不朽的战斗下去.于是让我们推开记忆的门,回过来看这个函数,当初我们讲到了 303 行,由于 us->sema 一开始就是锁着的,所以 down_interruptible 这里一开始就进入睡眠了,只有在接到唤醒的信号或者锁被释放了释放锁的进程来唤醒它它才会醒过来.那么谁来释放这把锁呢?

有两个地方,一个是这个模块要卸载了,这个我们稍后来看.另一个就是有 scsi 命令发过来了.scsi 命令从哪里发过来?很简单,scsi 核心层,硬件上来说,scsi 命令就是 scsi 主机到 scsi 设备,而从代码的角度来说,scsi 核心层要求为每一个 scsi host 实现一个 queuecommand 命令,每一次应用层发送来了 scsi 命令了,比如你去读写/dev/sda,最终 scsi 核心层就会调用与该 host 相对应 queuecommand,(确切地说是 struct Scsi_Host 结构体中的成员 struct scsi_host_template 中的成员指针 queuecommand,这是一个函数指针.)那么我们来看,当初我们定义的 struct scsi_host_template usb_stor_host_template,其中的确有一个 queuecommand,我们赋给它的值也是 queuecommand,即我们让 queuecommand 指向一个叫做 queuecommand 的函数,在 struct scsi_host_template 的定义中,函数指针的原型为:

```
107         int (* queuecommand)(struct scsi_cmnd *,
108                              void (*done)(struct scsi_cmnd *));
```

而我们所定义的 queuecommand()函数又在哪里呢?在 drivers/usb/storage/scsiglue.c 中:

```
165 /* queue a command */
166 /* This is always called with scsi_lock(srb->host) held */
167 static int queuecommand(struct scsi_cmnd *srb,
168                         void (*done)(struct scsi_cmnd *))
169 {
170         struct us_data *us = (struct us_data *)srb->device->host->hostdata[0];
171
172         US_DEBUGP("%s called\n", __FUNCTION__);
173         srb->host_scribble = (unsigned char *)us;
174
175         /* check for state-transition errors */
176         if (us->srb != NULL) {
177                 printk(KERN_ERR USB_STORAGE "Error in %s: us->srb = %p\n",
178                         __FUNCTION__, us->srb);
179                 return SCSI_MLQUEUE_HOST_BUSY;
```

```
180         }
181
182         /* fail the command if we are disconnecting */
183         if (test_bit(US_FLIDX_DISCONNECTING, &us->flags)) {
184                 US_DEBUGP("Fail command during disconnect\n");
185                 srb->result = DID_NO_CONNECT << 16;
186                 done(srb);
187                 return 0;
188         }
189
190         /* enqueue the command and wake up the control thread */
191         srb->scsi_done = done;
192         us->srb = srb;
193         up(&(us->sema));
194
195         return 0;
196 }
```

这个函数不长,它的使命也很简单,就是为了唤醒我们的那个沉睡中的守护进程,告诉他,爱不能再沉睡.

我们来仔细看一下,170 行,这种伎俩已经无需多说,当初我们第一次见到 hostdata[0]的时候就已经说得很清楚,它的出现就是为了给后面 scsi 相关的代码使用,而这里果然就用上了.

173 行,host_scrible 这个指针原本用来指向一段临时数据,或者叫中间结果暂存区,这行代码的直观作用就是令一个指针 host_scrible 的指针指向 us,但实际上这行代码是令人大跌隐形眼镜的,理由很简单,这是 Linux 内核代码中罕见的垃圾代码.因为 host_scrible 这个指针对我们来说没有任何作用,所以,最新的内核已经把这行代码删除了.当然,最初写这行代码也许只是因为作者看到台湾还有收复,祖国尚未统一,心情很郁闷,所以留这么一行代码以备忘,一旦将来祖国统一了就把它删掉.不过开源社区的其他同志大概不会同意,所以很自然这行代码没能保留下来.这从一个侧面也反映了开源社区还没能像我们中国,实现一片和谐,这种潜在的隐忧不得不令每一位有识之士为开源社区捏一把汗啊!然而我们需要说的是,host_scrible 虽然对我们的驱动没有用,但是在一些真正的 scsi host adpater 的驱动程序中,还是有多处使用了它,因为这个指针本来就是提供给底层驱动使用的.

176 行,判断 us->srb,事到如今,有些事瞒也瞒不住了,我们不得不去面对一个新的数据结构,它就是 struct scsi_cmnd.queuecommand()函数的第一个参数就是 struct scsi_cmnd 指针,而 struct us_data 中也有一个 struct scsi_cmnd *srb,也是一个指针.那么我们来看 struct scsi_cmnd,这个数据结构的意义很明显,就是代表一个 scsi 命令.她定义于 include/scsi/scsi_cmnd.h 中,

```
30 struct scsi_cmnd {
31         int      sc_magic;
32
33         struct scsi_device *device;
34         unsigned short state;
35         unsigned short owner;
36         struct scsi_request *sc_request;
```

```
37
38          struct list_head list;   /* scsi_cmnd participates in queue lists */
39
40          struct list_head eh_entry; /* entry for the host eh_cmd_q */
41          int eh_state;               /* Used for state tracking in error handlr */
42          int eh_eflags;              /* Used by error handlr */
43          void (*done) (struct scsi_cmnd *);        /* Mid-level done function */
44
45          /*
46           * A SCSI Command is assigned a nonzero serial_number when
internal_cmnd
47           * passes it to the driver's queue command function.   The serial_number
48           * is cleared when scsi_done is entered indicating that the command has
49           * been completed.   If a timeout occurs, the serial number at the moment
50           * of timeout is copied into serial_number_at_timeout.   By subsequently
51           * comparing the serial_number and serial_number_at_timeout fields
52           * during abort or reset processing, we can detect whether the command
53           * has already completed.   This also detects cases where the command has
54           * completed and the SCSI Command structure has already being reused
55           * for another command, so that we can avoid incorrectly aborting or
56           * resetting the new command.
57           */
58          unsigned long serial_number;
59          unsigned long serial_number_at_timeout;
60
61          int retries;
62          int allowed;
63          int timeout_per_command;
64          int timeout_total;
65          int timeout;
66
67          /*
68           * We handle the timeout differently if it happens when a reset,
69           * abort, etc are in process.
70           */
71          unsigned volatile char internal_timeout;
72
73          unsigned char cmd_len;
74          unsigned char old_cmd_len;
75          enum dma_data_direction sc_data_direction;
76          enum dma_data_direction sc_old_data_direction;
77
78          /* These elements define the operation we are about to perform */
79 #define MAX_COMMAND_SIZE          16
```

```
 80          unsigned char cmnd[MAX_COMMAND_SIZE];
 81          unsigned request_bufflen;         /* Actual request size */
 82
 83          struct timer_list eh_timeout;     /* Used to time out the command. */
 84          void *request_buffer;             /* Actual requested buffer */
 85
 86          /* These elements define the operation we ultimately want to perform */
 87          unsigned char data_cmnd[MAX_COMMAND_SIZE];
 88          unsigned short old_use_sg;        /* We save   use_sg here when requesting
 89                                            * sense info */
 90          unsigned short use_sg;   /* Number of pieces of scatter-gather */
 91          unsigned short sglist_len;        /* size of malloc'd scatter-gather list */
 92          unsigned short abort_reason;      /* If the mid-level code requests an
 93                                            * abort, this is the reason. */
 94          unsigned bufflen;         /* Size of data buffer */
 95          void *buffer;             /* Data buffer */
 96
 97          unsigned underflow;       /* Return error if less than
 98                                       this amount is transferred */
 99          unsigned old_underflow; /* save underflow here when reusing the
100                                     * command for error handling */
101
102          unsigned transfersize;   /* How much we are guaranteed to
103                                      transfer with each SCSI transfer
104                                      (ie, between disconnect /
105                                      reconnects.    Probably == sector
106                                      size */
107
108          int resid;               /* Number of bytes requested to be
109                                      transferred less actual number
110                                      transferred (0 if not supported) */
111
112          struct request *request;          /* The command we are
113                                               working on */
114
115 #define SCSI_SENSE_BUFFERSIZE    96
116          unsigned char sense_buffer[SCSI_SENSE_BUFFERSIZE];                /*
obtained by REQUEST SENSE
117                                                    * when CHECK CONDITION is
118                                                    * received on original command
119                                                    * (auto-sense) */
120
121          /* Low-level done function - can be used by low-level driver to point
122           *          to completion function.   Not used by mid/upper level code. */
```

```
    123         void (*scsi_done) (struct scsi_cmnd *);
    124
    125         /*
    126          * The following fields can be written to by the host specific code.
    127          * Everything else should be left alone.
    128          */
    129         struct scsi_pointer SCp;         /* Scratchpad used by some host adapters
*/
    130
    131         unsigned char *host_scribble;    /* The host adapter is allowed to
    132                                           * call scsi_malloc and get some memory
    133                                           * and hang it here.      The host adapter
    134                                           * is also expected to call scsi_free
    135                                           * to release this memory.  (The
memory
    136                                           * obtained by scsi_malloc is guaranteed
    137                                           * to be at an address < 16Mb). */
    138
    139         int result;              /* Status code from lower level driver */
    140
    141         unsigned char tag;       /* SCSI-II queued command tag */
    142         unsigned long pid;       /* Process ID, starts at 0 */
    143 };
```

我算是看明白了,凡是涉及 scsi 那边的数据结构就没有一个不变态,写代码的人成心就想吓唬我们.可是我们作为共产主义接班人,怎能被这些吓倒呢.不过还好,我们只要知道有这么一个数据结构就可以了,同时需要知道在 us 中有一个成员 srb,由它来指向 scsi 命令.知道这些足矣.继续说 176 行,看一下 us->srb 是不是为空,如果为空我们才可以继续往下走去唤醒那个守护进程,否则就说明人家那边之前的一个命令还没有执行完.你问为什么?到时候你完后看就知道了,人家那边执行完一个命令就会把它设为空.而作为第一次来说,显然 us->srb 为空,因为还没有任何人为它赋过值,只是初始化 us 的时候把所有元素都初始化为 0 了,所以这第一次来到这里的时候肯定是空.这里如果不为空就返回 SCSI_MLQUEUE_HOST_BUSY 给 scsi core,这样核心层就知道,人家这边 host 忙着呢,先不急着执行下面的命令.

183 行, US_FLIDX_DISCONNECTING 这个 flag 我们已经遇见多次了,已经无需多讲,现在只是不知道究竟是哪里设置了这个 flag,日后我们看到 storage_disconnect 就知道了.这里和以往一样,如果这个 flag 设置了,就别浪费大家的感情了,赶紧结束吧.设置 srb->result 让 scsi core 知道这里已经断开连接了.而 queuecommand 命令本身就返回 0.不过我们需要注意的是 186 行这个 done 函数,仔细看这个 done 是 queuecommand()函数的第二个参数,是一个函数指针,实际上 scsi core 调用 queuecommand 的时候传递的参数名字就叫做 scsi_done,这就是一个函数名,scsi 核心层定义了一个叫做 scsi_done 的函数,scsi 核心层要求当低层的驱动程序完成一个命令后要调用这个函数去通知 scsi 核心层,这实际上就相当于一种中断机制,scsi 核心层调用了 queuecommand()之后它就不管事了,它就去干别的了,等你底层的代码把这个 queuecommand 执行完了之后,或者准确地说当你底层把命令执行完了之后,你就调用 scsi_done 从而 scsi 核心层就知道你这个命令完成了,然后它就会接着做一些它该做的事情比如清理这个命令,或者别的一些收尾的工作.所以这里我们看到,如果设备已经设置了断开的 flag,那么这里就执行 done,如果没有断开那就在下面的 190 行设置 srb->scsi_done 等于这个 done,实际上就是等于 scsi_done,这

两个 scsi_done,一个是 struct scsi_cmnd *srb 的成员指针,一个是 scsi 核心层的函数名.虽然它们同名,但是是两个不同的冬冬,就好像满世界都是叫张伟,王伟,李伟,刘伟的一样,名字相同人不同,很正常.

最后,192 行,令 us->srb 等于这个 srb.而 193 行,这正是我们苦苦寻找的代码,正是这个 up(&us->sema),唤醒了我们的守护进程.之后,195 行,这个函数本身就结束了,像烟花熄灭,像夜空沉寂.而我们显然就该去看那个 usb_stor_control_thread()了.因为,她醒了,她终于醒了.

彼岸花的传说(五)

燕子去了,有再来的时候;杨柳枯了,有再青的时候;桃花谢了,有再开的时候;老婆离了,有再找的时候,孩子跑了,有回来的时候;煮熟的鸭子飞了,有飞回来的时候.一个函数没讲完就跳走了,有再回来的时候.其实,那些人,那些事,终究不曾远离.

于是,她再一次进入我们的视野.

她就是 usb_stor_control_thread().唤醒她的是来自 queuecommand 的 up(&(us->sema)),us->srb 被赋值为 srb,而 srb 是来自 scsi 核心层在调用 queuecommand 时候传递进来的参数.聚焦 usb_stor_control_thread(),309 行, 前面说过,关于 dev_semaphore 这把锁我们必须在看完整个模块之后再来总较高的角度来看,所以这里自然先跳过.

312 行,没啥好说的,如果传进来的是 NULL,那么还是退出吧.别无选择.(为什么命令是空就说明要结束了?因为如果 srb 是空,那么 scsi core 是不会调用 queuecommand 的,道理不用说了吧,scsi core 也不傻,都没有命令它还调用底层的函数干嘛,吃错药了?那么 queuecommand 没有调用的话是谁唤醒了这个守护进程?我们前面说了,唤醒它的有两个地方,一个是 queuecommand,另一个就是模块要卸载了,准确的说是 usb_stor_release_resources()函数,日后等我们遇到了这个函数再看吧,总之就是说当这个模块都要释放资源了,这时候 srb 肯定会被设为 NULL,然后这种情况下唤醒了我们这个守护进程,于是才有了这里这个判断语句.)

318 行,host 也是一把锁,这把锁我们也到最后再来看.暂且不表.

321 行至 331 行,又是判断两个 flag 有没有被设置,第二个不用多说了,看都看腻了,老是判断设备有没有被拔出,要是你的 U 盘插进去了永远不拔出来,那么你可以把这个 flag 相关的代码都删了,当然事实上是你不可能不拔出来,热插拔本来就是 usb 设备的一大特性.所以你就能理解写代码的哥们儿为什么这么煞费苦心的去判断设备是不是已经不在了.那么第一个 flag 呢,US_FLDX_TIMED_OUT,这个 flag 的含义也如其字面意义一样,超时了,超时的概念在计算机的世界里比比皆是.不过对于这个 flag,设置它的函数是 command_abort,这个函数也是咱们提供的,由 scsi 核心层去调用,由它那边负责计时,到了超时的时间它就调用 command_abort.我们稍候会看,先不急.

338 行,判断 srb 的一个成员 sc_data_direction, 先看 DMA_BIDIRECTIONAL 这个宏.这个宏定义于 include/linux/dma-mapping.h 中,

```
7 /* These definitions mirror those in pci.h, so they can be used
8  * interchangeably with their PCI_ counterparts */
9 enum dma_data_direction {
```

```
10          DMA_BIDIRECTIONAL = 0,
11          DMA_TO_DEVICE = 1,
12          DMA_FROM_DEVICE = 2,
13          DMA_NONE = 3,
14 };
```

在 scsi_cmnd 结构体里边,有这么两个成员,

```
75          enum dma_data_direction sc_data_direction;
76          enum dma_data_direction sc_old_data_direction;
```

这些被用来表征数据阶段数据传输的方向.DMA_TO_DEVICE 表示从主存到设备,DMA_FROM_DEVICE 表示从设备到主存.坊间有传闻说,DMA_NONE 则只被用于调试,一般不能使用否则将有可能导致内核崩溃.不过更准确一点是,usb mass storage 协议里边规定了双向传输是非法的,而一个命令传输零数据是合法的,比如 TEST_UNIT_READY 命令就不用传输数据.DMA_BIDIRECTIONAL 表示两个方向都有可能,换言之也就是不知道究竟是哪个方向.就比如您找某位半仙算命,而当您想考考他,问他您是什么时候生的,他却对您说,您不是上半年生的就是下半年生的.这样只能说明他什么都不知道.同理,338 行看到 srb 的 sc_data_direction 是 DMA_BIDIRECTIONAL 的时候,自然就当作出错了.因为不确定方向的话也就没法传输数据了嘛不是.

346 行, US_FL_SCM_MULT_TARG 这个 flag,表示设备支持多个 target,这里的意思很明显,对于那些不支持多个 target 的设备,其 us->srb->device->id 必须为 0,否则就有问题了.struct us_data 结构体中的成员 struct scsi_cmnd * srb,struct scsi_cmnd 结构体中有一成员 struct scsi_device * device,而 struct scsi_device 顾名思义,描述一个 scsi device,就像过去的 struct usb_device 用来描述 usb device 一样,这些写 Linux 代码的哥们儿也没别的技巧,就这几招.不用看也知道,struct scsi_device 又是一个变态的数据结构,还是那句话,仿佛不写一些变态数据结构来不足以体现 Linus 这帮子人是腕儿.具体来看,她来自 include/scsi/scsi_device.h 中:

```
38 struct scsi_device {
39          struct Scsi_Host *host;
40          struct request_queue *request_queue;
41
42          /* the next two are protected by the host->host_lock */
43          struct list_head     siblings;    /* list of all devices on this host */
44          struct list_head     same_target_siblings; /* just the devices sharing same
target id */
45
46          volatile unsigned short device_busy;     /* commands actually active on
low-level */
47          spinlock_t sdev_lock;              /* also the request queue_lock */
48          spinlock_t list_lock;
49          struct list_head cmd_list;       /* queue of in use SCSI Command structures
*/
50          struct list_head starved_entry;
51          struct scsi_cmnd *current_cmnd; /* currently active command */
52          unsigned short queue_depth;      /* How deep of a queue we want */
```

```
53          unsigned short last_queue_full_depth; /* These two are used by */
54          unsigned short last_queue_full_count; /* scsi_track_queue_full() */
55          unsigned long last_queue_full_time;/* don't let QUEUE_FULLs on the same
56                                                jiffie count on our counter, they
57                                                could all be from the same event. */
58
59          unsigned int id, lun, channel;
60
61          unsigned int manufacturer;        /* Manufacturer of device, for using
62                                             * vendor-specific cmd's */
63          unsigned sector_size;    /* size in bytes */
64
65          void *hostdata;           /* available to low-level driver */
66          char devfs_name[256];    /* devfs junk */
67          char type;
68          char scsi_level;
69          char inq_periph_qual;    /* PQ from INQUIRY data */
70          unsigned char inquiry_len;        /* valid bytes in 'inquiry' */
71          unsigned char * inquiry;          /* INQUIRY response data */
72          char * vendor;            /* [back_compat] point into 'inquiry' ... */
73          char * model;             /* ... after scan; point to static string */
74          char * rev;               /* ... "nullnullnullnull" before scan */
75          unsigned char current_tag;        /* current tag */
76          struct scsi_target        *sdev_target;    /* used only for single_lun */
77
78          unsigned int     sdev_bflags; /* black/white flags as also found in
79                                         * scsi_devinfo.[hc]. For now used only to
80                                         * pass settings from slave_alloc to scsi
81                                         * core. */
82          unsigned writeable:1;
83          unsigned removable:1;
84          unsigned changed:1;       /* Data invalid due to media change */
85          unsigned busy:1;          /* Used to prevent races */
86          unsigned lockable:1;      /* Able to prevent media removal */
87          unsigned locked:1;        /* Media removal disabled */
88          unsigned borken:1;        /* Tell the Seagate driver to be
89                                     * painfully slow on this device */
90          unsigned disconnect:1;   /* can disconnect */
91          unsigned soft_reset:1;   /* Uses soft reset option */
92          unsigned sdtr:1;          /* Device supports SDTR messages */
93          unsigned wdtr:1;          /* Device supports WDTR messages */
94          unsigned ppr:1;           /* Device supports PPR messages */
95          unsigned tagged_supported:1;      /* Supports SCSI-II tagged queuing */
96          unsigned simple_tags:1; /* simple queue tag messages are enabled */
```

```
 97          unsigned ordered_tags:1;/* ordered queue tag messages are enabled */
 98          unsigned single_lun:1;   /* Indicates we should only allow I/O to
 99                                    * one of the luns for the device at a
100                                    * time. */
101          unsigned was_reset:1;    /* There was a bus reset on the bus for
102                                    * this device */
103          unsigned expecting_cc_ua:1; /* Expecting a
CHECK_CONDITION/UNIT_ATTN
104                                       * because we did a bus reset. */
105          unsigned use_10_for_rw:1; /* first try 10-byte read / write */
106          unsigned use_10_for_ms:1; /* first try 10-byte mode sense/select */
107          unsigned skip_ms_page_8:1;       /* do not use MODE SENSE page 0x08
*/
108          unsigned skip_ms_page_3f:1;      /* do not use MODE SENSE page 0x3f */
109          unsigned use_192_bytes_for_3f:1; /* ask for 192 bytes from page 0x3f */
110          unsigned no_start_on_add:1;      /* do not issue start on add */
111          unsigned allow_restart:1; /* issue START_UNIT in error handler */
112          unsigned no_uld_attach:1; /* disable connecting to upper level drivers */
113          unsigned select_no_atn:1;
114          unsigned fix_capacity:1;         /* READ_CAPACITY is too high by 1 */
115
116          unsigned int device_blocked;     /* Device returned QUEUE_FULL. */
117
118          unsigned int max_device_blocked; /* what device_blocked counts down
from  */
119 #define SCSI_DEFAULT_DEVICE_BLOCKED      3
120
121          int timeout;
122
123          struct device            sdev_gendev;
124          struct class_device      sdev_classdev;
125
126          struct class_device      transport_classdev;
127
128          enum scsi_device_state sdev_state;
129          unsigned long            sdev_data[0];
130 } __attribute__((aligned(sizeof(unsigned long))));
```

这个结构体将在未来的日子里被我们多次提到.当然,此刻,我们只需要注意到 unsigned int id, lun, channel 这三个成员,这正是定位一个 scsi 设备必要的三个成员,一个 scsi 卡所控制的设备被划分为几层,先是若干个 channel,然后每个 channel 上有若干个 target,每个 target 用一个 target id 来表征,然后一个 target 可以有若干个 lun,而咱们这里判断的是 target id.对于不支持多个 target 的设备,她必须为 0.对于绝大多数 usb mass storage 设备来说,它们的 target id 肯定为 0,但是世界上总是有那么多怪事,有些设备厂家就是要标新立异,它就是要让你个设备支持多个 target,于是它就可以设置

US_FL_SCM_MULT_TARG 这么一个 flag,比如我们可以在 drivers/usb/storage/unusual_devs.h 中看到如下的定义:

```
UNUSUAL_DEV(  0x04e6, 0x0002, 0x0100, 0x0100,
              "Shuttle",
              "eUSCSI Bridge",
              US_SC_DEVICE, US_PR_DEVICE, usb_stor_euscsi_init,
              US_FL_SCM_MULT_TARG ),
```

然后 353 行,us->srb->device->lun 不应该大于 us->max_lun,这两个冬冬是什么区别?us->max_lun 是咱们年轻的时候使用 storage_probe 调用 usb_stor_Bulk_max_lun()函数来向 usb mass storage 设备获得的最大 LUN,比如 MAX LUN 等于 3,那么咱们这个设备支持的就是 4 个 LUN,即 0,1,2,3.而 us->srb->device->lun 则可以是这四个值中的任一个,看咱们传递进来的命令是要访问谁了.但她显然不可能超过 MAX LUN.

然后就是 359 行了.看到这么一个 flag-US_FL_FIX_INQUIRY,这又是 us->flags 中众多 flag 中的一个,一些定义于 drivers/usb/storage/unusal_devs.h 中的设备有这个 flag,事实上,通常大多数设备的 vendor name 和 product name 是通过 INQUIRY 命令来获得的,而这个 flag 表明,这些设备的 vendor name 和 product name 不需要查询,或者根本就不支持查询,她们的 vendor name 和 product name 直接就定义好了,在 unusal_devs.h 中就设好了.那么 359 行这里这个 cmnd[0]是什么?struct scsi_cmnd 里边有这么一个成员,

```
79 #define MAX_COMMAND_SIZE        16
80          unsigned char cmnd[MAX_COMMAND_SIZE];
```

这个数组 16 个元素,她包含的就是 scsi 命令,要看懂这个条件判断,得先看下边那句 fill_inquiry_response()函数调用.如果大家都没意见的话,我提议咱们下节再接着讲.

最后贴几个设了 US_FL_FIX_INQUIRY 这个 flag 的设备,这几个都是 Sony 的 PEG 记忆棒,或者叫记忆卡,可以用在 PDA 里边.drivers/usb/storage/unusual_devs.h 中:

```
377 /* Submitted by Nathan Babb <nathan@lexi.com> */
378 UNUSUAL_DEV(  0x054c, 0x006d, 0x0000, 0x9999,
379                 "Sony",
380                 "PEG Mass Storage",
381                 US_SC_DEVICE, US_PR_DEVICE, NULL,
382                 US_FL_FIX_INQUIRY ),
383
384 /* Submitted by Mike Alborn <malborn@deandra.homeip.net> */
385 UNUSUAL_DEV(  0x054c, 0x016a, 0x0000, 0x9999,
386                 "Sony",
387                 "PEG Mass Storage",
388                 US_SC_DEVICE, US_PR_DEVICE, NULL,
389                 US_FL_FIX_INQUIRY ),
390
```

```
391 /* Submitted by Frank Engel <frankie@cse.unsw.edu.au> */
392 UNUSUAL_DEV(  0x054c, 0x0099, 0x0000, 0x9999,
393                 "Sony",
394                 "PEG Mass Storage",
395                 US_SC_DEVICE, US_PR_DEVICE, NULL,
396                 US_FL_FIX_INQUIRY ),
```

常用掌上电脑的同志们不会对 Sony 的这些 PEG 产品陌生吧.

彼岸花的传说(六)

七年前,在那个千禧年里,凭借<<我的父亲母亲>>获得金鸡奖最佳女主角的章子怡姐姐说:"我长得挺漂亮,又是单身,男人不可能对我没兴趣!"是的,古人云,男人分两种,一种是好色,一种是十分好色.所以章子怡这话一点没错.不过,对于大多数 80 后来说,他们早已不再像十年前那么另类,那么出格,因为他们生活压力很重,他们很老实,很现实,一个显而易见的事实,yy 章子怡不如老老实实的学 Linux.虽然很多人对两者都有兴趣.但至少学会了后者,可以混口饭吃,谁叫我们都是知识混子呢.

此时,镜头一转,我们继续接着上一节往下看.fill_inquiry_response(),这个函数来自 drivers/usb/storage/usb.c 中,

```
240 /*
241  * fill_inquiry_response takes an unsigned char array (which must
242  * be at least 36 characters) and populates the vendor name,
243  * product name, and revision fields. Then the array is copied
244  * into the SCSI command's response buffer (oddly enough
245  * called request_buffer). data_len contains the length of the
246  * data array, which again must be at least 36.
247  */
248
249 void fill_inquiry_response(struct us_data *us, unsigned char *data,
250                 unsigned int data_len)
251 {
252         if (data_len<36) // You lose.
253                 return;
254
255         if(data[0]&0x20) { /* USB device currently not connected. Return
256                               peripheral qualifier 001b ("...however, the
257                               physical device is not currently connected
258                               to this logical unit") and leave vendor and
259                               product identification empty. ("If the target
260                               does store some of the INQUIRY data on the
261                               device, it may return zeros or ASCII spaces
262                               (20h) in those fields until the data is
263                               available from the device."). */
```

```
264                 memset(data+8,0,28);
265         } else {
266                 memcpy(data+8, us->unusual_dev->vendorName,
267                         strlen(us->unusual_dev->vendorName) > 8 ? 8 :
268                         strlen(us->unusual_dev->vendorName));
269                 memcpy(data+16, us->unusual_dev->productName,
270                         strlen(us->unusual_dev->productName) > 16 ? 16 :
271                         strlen(us->unusual_dev->productName));
272                 data[32] = 0x30 + ((us->pusb_dev->descriptor.bcdDevice>>12)
& 0x0F);
273                 data[33] = 0x30 + ((us->pusb_dev->descriptor.bcdDevice>>8) &
0x0F);
274                 data[34] = 0x30 + ((us->pusb_dev->descriptor.bcdDevice>>4) &
0x0F);
275                  data[35] = 0x30 + ((us->pusb_dev->descriptor.bcdDevice) &
0x0F);
276         }
277
278         usb_stor_set_xfer_buf(data, data_len, us->srb);
279 }
```

故事发生的太突然,会让人产生幻觉. 本来我们正儿八经用来处理 scsi 命令的函数是后面将要讲的 proto_handler(),但想不到我们在这里开始接触 scsi 命令了.理由正是因为像 Sony 这几款 PEG 产品做的不好,连最基本的 scsi 命令 INQUIRY 都不支持,完了又想在 Linux 中使用,那没办法了,Sony 毕竟是大公司,连欧洲冠军杯都是他们家和喜力给赞助的,开源社区没有必要得罪他们,所以就准备一个函数来 fix 这个问题吧,毫无疑问,这属于硬件上的一个 bug.

那么什么是 INQUIRY 命令?曾经也提过,INQUIRY 命令是最最基本的一个 SCSI 命令,比如主机第一次探测设备的时候就要用 INQUIRY 命令来了解这是一个什么设备,比如 scsi 总线上有一个插槽插了一个设备,那么 scsi 主机就问它,你是 scsi 磁盘呢,还是 scsi 磁带,又或是 scsi 的 CD ROM 呢?作为设备,它内部一定有一段固件程序,即所谓的 firmware.它就在接收到主机的 INQUIRY 命令之后作出回答.具体应该怎么回答?当然是依据 scsi 协议里规定的格式了.不仅仅 INQUIRY 命令,对于每一个命令都应该如此.只要对方问:天王盖地虎.你作为设备就该回答,宝塔镇河妖.这其实就好比我们对对联,人家问天恢弘,地恢弘,天地恢弘,就得对你妈的,他妈的,你他妈的.这都是不成文的规矩,而开发 scsi 的人把这些写成了规范,它就变成了成文的规矩了.具体来说, 设备在受到 INQUIRY 命令查询时,她的相应遵从 scsi 协议里面规定的标准格式,标准格式规定了,响应数据必须至少包含 36 个字节,所以 252 行,如果 data_len 小于 36,那就甭往下走了,返回吧.您违规了.

如果你对 scsi 协议很陌生,还是没有明白 INQUIRY 命令究竟是做什么,那么推荐一个工具给你,你可以试一试,以便有个直观的印象,其实 INQUIRY 命令就是如其字面意思一样,查询,查询设备的一些基本信息,从软件的角度来说,在主机扫描的时候,或者说枚举的时候,向每一个设备发送这个命令,并且获得回答,驱动程序从此就会保存这些信息,因为这些信息之后可能都会用到或者说其中的一部分会被用到.这里推荐的工具是 sg_utils3,这是一个软件包,Linux 中可以使用的软件包,到处都有下,下了之后安装上,然后它包含一个应用程序 sg_inq,这其实就是给设备发送 INQUIRY 命令用的,用法如下所示:

```
[root@localhost ~]# sg_inq -36 /dev/sda

standard INQUIRY:
  PQual=0  Device_type=0  RMB=1  version=0x02  [SCSI-2]
  [AERC=0]  [TrmTsk=0]  NormACA=0  HiSUP=0  Resp_data_format=2
  SCCS=0  ACC=0  TGPS=0  3PC=0  Protect=0  BQue=0
  EncServ=0  MultiP=0  [MChngr=0]  [ACKREQQ=0]  Addr16=0
  [RelAdr=0]  WBus16=0  Sync=0  Linked=0  [TranDis=0]  CmdQue=0
  length=36 (0x24)   Peripheral device type: disk
 Vendor identification: Intel
 Product identification: Flash Disk
 Product revision level: 2.00
```

这里我使用的是我们 Intel 自己生产的一块 U 盘,去年过年的时候我们大老板发的,使用 sg_inq 命令可以查询到关于我的这块 U 盘的基本信息,实际上 sg_inq 可以查询所有的 scsi 设备的信息,因为 INQUIRY 本来就是一个标准的 SCSI 命令嘛.当然以上这些信息中,我们之后用得到的大概也就是 Vendor ID,Product ID,Product revision,以及那个 length,device type--disk,还有那个中括号里的 SCSI-2,这代表遵守的 SCSI 的版本,scsi 协议也发展了这么多年,当然也有不同的版本了.

Ok,有了直观的印象了我们就继续看代码,255 行,判断 data[0]是否是 20h,20h 有什么特别的吗?当然.scsi 协议里规定了,标准的 INQUIRY data 的 data[0],总共 8 个 bit 嘛不是,其中 bit7~bi5 被称为 peripheral qualifier(三位),而 bit4~bit0 被称为 perpheral device type(五位),这俩家伙代表了不同的含义,但是 20h 就表示 peripheral qualifier 这个外围设备限定符为 001b,而 peripheral device type 这个外围设备类型则为 00h,查阅 scsi 协议可知,后者代表的是设备类型为磁盘,或者说直接访问设备,前者代表的是目标设备的当前 lun 支持这种类型,然而,实际的物理设备并没有连接在当前 lun 上.在 data[36]中,从 data[8]一直到 data[35]这 28 个字节都是保存的 vendor 和 product 的信息.scsi 协议里边写了,如果设备里有保存这些信息,那么她可以暂时先返回 0x20h,因为现在是系统 power on 时期或者是 reset 期间,要尽量减少延时,于是 fill_inquiry_response()就会把 data[8]到 data[35]都给设置成 0.等到保存在设备上的这些信息可以读了再去读.

如果不是 20h,比如我们这里传递进来的 data[0]就是 0,那么看 266 行,data[8]开始的 8 个字节可以保存 vendor 相关的信息,对于,us->unusual_dev,我们早已不陌生,struct us_data 结构体中的成员 struct us_unusual_dev *unusual_dev, 想当年,咱们在 storage_probe() 时曾经把 us_unusual_dev_list[] 数组中的对应元素赋给了她, 而 us_unusua_dev_list[] 又来自 unusual_devs.h,都是预先定义好了的.所以这里就是把其中的 vendorName 复制到 data 数组中来,但是如果 vendorName 超过 8 个字符了那可不行,只取前 8 个就 ok 了,当然像我家 Intel 就不存在这个问题了,只有 5 个字符,大多数公司也都是八个字符以内,比如长一点的名字有 Motorola,Sansung 也都没问题.同样 productName 也是一样的方法,复制到 data 数组中来,协议里规定了,从 16 开始存放 productName,不能超过 16 个字符,那么”Flash Disk”也没有问题.(注:正式版此处将插入图片,Standard INQUIRY data format)

然后可以看 272 行,us->pusb_dev->descriptor.bcdDevice,struct us_data 中有一个成员 struct usb_device *pusb_dev,而 struct usb_device 中有一个成员 struct usb_device_descriptor descriptor,而 struct usb_device_descriptor 中的成员__u16 bcdDevice,表示的是制造商指定的产品的版本号,道上的规矩是用版本号,制造商 id 和产品 id 来标志一个设备.bcdDevice 一共 16 位,是以 bcd

码的方式保存的信息,也就是说,每 4 位代表一个十进制的数,比如 0011 0110 1001 0111 就代表的 3697.而在 scsi 标准的 INQUIRY data 中,data[32]到 data[35]被定义为保存这四个数,并且要求以 ASCII 码的方式保存,ASCII 码中 48 对应咱们日常的 0,49 对应 1,50 对应 2,也就是说得在现有数字的基础上加上 48,或者说加上 0x30.这就是 272 到 275 行所表达的意思.

一切准备好了之后,我们就可以把 data 数组,这个包含 36 个字符的信息发送到 scsi 命令指定的位置了,即 srb 指定的位置.这正是 278 行中, usb_stor_set_xfer_buf 的所作所为.

在接着讲 278 行这个函数 usb_stor_set_xfer_buf 之前,先解释一下之前定义 data_ptr[36]时初始化的前 8 个元素.她们的含义都和 scsi 协议规定的对应.data_ptr[0]不用说了,data_ptr[1]被赋为 0x80,这表明这个设备是可移除的,data_ptr[2]被赋为 0x02 这说明设备遵循 SCSI-2 协议,data_ptr[3]被赋为 0x02,说明数据格式遵循国际标准化组织所规定的格式,而 data_ptr[4]被称为 additional length,附加参数的长度,即除了用这么一个标准格式的数据响应之外,可能还会返回更多的一些信息.这里设置的是 0x1F.

彼岸花的传说(七)

很显然,我们是把为 INQUIRY 命令准备的数据保存到了我们自己定义的一个结构体中,即 struct data_ptr[36],但是我们是为了回应一个 SCSI 命令,最终需要知道答案的是 scsi 核心层.正是它们传递了一个 scsi_cmnd 结构体下来,即 srb.struct scsi_cmnd 中有两个成员, unsigned request_bufflen 和 void *request_buffer,小宇宙告诉我们,应该把 data 数组中的数据传送到 request_buffer 中去,这样,scsi 核心层就知道去哪里获取结果.没错,当时就是这样!

usb_stor_set_xfer_buf()这个函数来自,drivers/usb/storage/protocol.c 中.

```
281 /* Store the contents of buffer into srb's transfer buffer and set the
282  * SCSI residue. */
283 void usb_stor_set_xfer_buf(unsigned char *buffer,
284         unsigned int buflen, struct scsi_cmnd *srb)
285 {
286         unsigned int index = 0, offset = 0;
287
288         usb_stor_access_xfer_buf(buffer, buflen, srb, &index, &offset,
289                         TO_XFER_BUF);
290         if (buflen < srb->request_bufflen)
291                 srb->resid = srb->request_bufflen - buflen;
292 }
```

主要调用的又是 usb_stor_access_xfer_buf()函数,这个函数也来自同一个文件,drivers/usb/storage/protocol.c,

```
185
/***********************************************************************
186  * Scatter-gather transfer buffer access routines
187  ***********************************************************************
```

```
******/
 188
 189 /* Copy a buffer of length buflen to/from the srb's transfer buffer.
 190  * (Note: for scatter-gather transfers (srb->use_sg > 0), srb->request_buffer
 191  * points to a list of s-g entries and we ignore srb->request_bufflen.
 192  * For non-scatter-gather transfers, srb->request_buffer points to the
 193  * transfer buffer itself and srb->request_bufflen is the buffer's length.)
 194  * Update the *index and *offset variables so that the next copy will
 195  * pick up from where this one left off. */
 196
 197 unsigned int usb_stor_access_xfer_buf(unsigned char *buffer,
 198         unsigned int buflen, struct scsi_cmnd *srb, unsigned int *index,
 199         unsigned int *offset, enum xfer_buf_dir dir)
 200 {
 201         unsigned int cnt;
 202
 203         /* If not using scatter-gather, just transfer the data directly.
 204          * Make certain it will fit in the available buffer space. */
 205         if (srb->use_sg == 0) {
 206                 if (*offset >= srb->request_bufflen)
 207                         return 0;
 208                 cnt = min(buflen, srb->request_bufflen - *offset);
 209                 if (dir == TO_XFER_BUF)
 210                         memcpy((unsigned char *) srb->request_buffer + *offset,
 211                                         buffer, cnt);
 212                 else
 213                         memcpy(buffer, (unsigned char *) srb->request_buffer +
 214                                         *offset, cnt);
 215                 *offset += cnt;
 216
 217         /* Using scatter-gather.  We have to go through the list one entry
 218          * at a time.  Each s-g entry contains some number of pages, and
 219          * each page has to be kmap()'ed separately.  If the page is already
 220          * in kernel-addressable memory then kmap() will return its address.
 221          * If the page is not directly accessible -- such as a user buffer
 222          * located in high memory -- then kmap() will map it to a temporary
 223          * position in the kernel's virtual address space. */
 224         } else {
 225                 struct scatterlist *sg =
 226                                 (struct scatterlist *) srb->request_buffer
 227                                 + *index;
 228
 229                 /* This loop handles a single s-g list entry, which may
 230                  * include multiple pages.  Find the initial page structure
```

```
231                  * and the starting offset within the page, and update
232                  * the *offset and *index values for the next loop. */
233                 cnt = 0;
234                 while (cnt < buflen && *index < srb->use_sg) {
235                         struct page *page = sg->page +
236                                         ((sg->offset + *offset) >> PAGE_SHIFT);
237                         unsigned int poff =
238                                         (sg->offset + *offset) & (PAGE_SIZE-1);
239                         unsigned int sglen = sg->length - *offset;
240
241                         if (sglen > buflen - cnt) {
242
243                                 /* Transfer ends within this s-g entry */
244                                 sglen = buflen - cnt;
245                                 *offset += sglen;
246                         } else {
247
248                                 /* Transfer continues to next s-g entry */
249                                 *offset = 0;
250                                 ++*index;
251                                 ++sg;
252                         }
253
254                         /* Transfer the data for all the pages in this
255                          * s-g entry.  For each page: call kmap(), do the
256                          * transfer, and call kunmap() immediately after. */
257                         while (sglen > 0) {
258                                 unsigned int plen = min(sglen, (unsigned int)
259                                                 PAGE_SIZE - poff);
260                                 unsigned char *ptr = kmap(page);
261
262                                 if (dir == TO_XFER_BUF)
263                                         memcpy(ptr + poff, buffer + cnt, plen);
264                                 else
265                                         memcpy(buffer + cnt, ptr + poff, plen);
266                                 kunmap(page);
267
268                                 /* Start at the beginning of the next page */
269                                 poff = 0;
270                                 ++page;
271                                 cnt += plen;
272                                 sglen -= plen;
273                         }
274                 }
```

```
275         }
276
277         /* Return the amount actually transferred */
278         return cnt;
279 }
```

风,总是在最温柔的时候醉人;

雨,总是在最纤细的时候飘逸;

花,总是在将调零的时候让人惋惜;

夜,总是在最深冷的时候令人希冀;

在编写 Linux 设备驱动时,在生与死的抉择中,我们总是无法逃避,我们总是要涉及内存管理.内存管理毫无疑问是 Linux 内核中最复杂的一部分,能不涉及我们都希望别去涉及.但生活中总是充满了无奈,该来的还是会来.而我们倒是该庆幸,毕竟它不是每个月来一次.(这里说的是水电煤气费的单子,不许误解.)更不是八点档的连续剧,每天准时来赚你的眼泪.

所以,usb_stor_access_xfer_buf()函数映入了人们的眼帘.

首先判断 srb->use_sg 是否为 0.

无聊的 it 玩家们创建了有一个词,叫做 scatter/gather,她是一种用于高性能 IO 的标准技术.她通常意味着一种 DMA 传输方式,对于一个给定的数据块,她老人家可能在内存中存在于一些离散的缓冲区,换言之,就是说一些不连续的内存缓冲区一起保存一个数据块,如果没有 scatter/gather 呢,那么当我们要建立一个从内存到磁盘的传输,那么操作系统通常会为每一个 buffer 做一次传输,或者干脆就是把这些不连续的 buffer 里边的冬冬全都移动到另一个很大的 buffer 里边,然后再开始传输.那么这两种方法显然都是效率不高的.毫无疑问,如果 操作系统/驱动程序/硬件 能够把这些来自内存中离散位置的数据收集起来(gather up)并转移她们到适当位置整个这个步骤是一个单一的操作的话,效率肯定就会更高.反之,如果要从磁盘向内存中传输,而有一个单一的操作能够把数据块直接分散开来(scatter)到达内存中需要的位置,而不再需要中间的那个块移动,或者别的方法,那么显然,效率总会更高.

在 struct scsi_cmnd 中,有一个成员 unsigned short use_sg,上头传下来的 scsi_cmnd,其 use_sg 是设好了的,咱们判断一下,如果她为 0,那么说明没有使用 scatter/gather.struct scsi_cmnd 中还有两个成员,unsigned request_bufflen 和 void *request_buffer,她们和 use_sg 是什么关系呢?

事实上,要玩 scatter/gather,就需要一个 scatterlist 数组,有人管她叫散列表数组.对于不同的硬件平台,定义了不同的 struct scatterlist 结构体,她们来自 include/asm/scatterlist.h 中,(如果是硬件平台 i386 的,那么就是 include/asm-i386/scatterlist.h,如果是 x86_64 的平台,那么就在 include/asm-x86_64/scatterlist.h 中),然后所谓的 scatter/gather 就是一次把整个 scatterlist 数组给传送掉.而 use_sg 为 0 就表示没有 scatter gather list,或者说 scatterlist,对于这种情况,数据将直接传送给 request_buffer 或者直接从 request_buffer 中取得数据.而如果 use_sg 大于 0,那么表示 scatter gather list 这么一个数组就在 request_buffer 中,而数组元素个数正是 use_sg 个.也就是说,srb->request_buffer 里边的冬冬有两种可能,一种是包含了数据本身,另一种是包含了 scatter gather list.具体是哪种情况通过判断 use_sg 来决定.而接下来即将要讲到的 srb->request_bufflen 顾

名思义,就是 buffer 的长度,但对于 use_sg 大于 0 的情况,换言之,对于使用 scatter gather list 的情况,request_bufflen 没有意义,将被忽略.

对这些原理有了基本的了解之后,我们可以从下节开始看代码了.这里先提醒一下,要注意我们这个函数虽然看似是传输数据,可它实际上并没有和 usb 真正发生关系,我们只是从软件上来 fix 一个硬件的 bug,这个 bug 就是我们已经说过了的,不能响应基本的 SCSI 命令 INQUIRY,你问她,她完全不予理睬.至于为什么不能响应我就不用说了吧?人的冷漠,不是因为天生就如此,只是暗淡的心境早已将所有通向阳光的窗户关闭了.而设备的冷漠,显然是制造商做的傻事了.(我没有讽刺 Sony 的意思,事实上这种情况并非只有 Sony 他们家才出现了,别的厂家也有这样的产品.只是,Device 有 bug,可以让程序去补,我们 80 后的梦碎了,拿什么去补?)

所以对于那些不能相应 INQUIRY 命令的设备,当上层的驱动程序去 INQUIRY 的时候,实际上是调用我们的 queuecommand,那么我们根本就不用和下面的硬件去打交道,就直接回复上层,即我们从软件上来准备这个一段 INQUIRY 数据给上层,这才是我们这个函数的目的.真正的和硬件打交道的代码在后面,我们还没走到那一步.到了那一步再说.

彼岸花的传说(八)

对于 use_sg 为 0 的情况,我们接下来再看 206 行,offset 是函数调用传递进来的参数,注释里说的很清楚,就是用来标志偏移量的,每次 copy 几个字节她就增加几,最大她也不能超过 request_bufflen,这是显然的.usb_stor_access_xfer_buf()这个函数所做的事情就是从 srb->request_buffer 往 buffer 里边 copy 数据,或者反过来从 buffer 往 srb->request_buffer,然后返回 copy 了多少个字节.对于 offset 大于等于 request_bufflen 的情况,当然就直接返回 0 了,因为 request_buffer 已经满了.

参数 enum xfer_buf_dir dir 标志的正是传输方向,这个数据类型是在某个猥琐的角落里被定义的,drivers/usb/storage/protocol.h 中:

```
68 /* struct scsi_cmnd transfer buffer access utilities */
69 enum xfer_buf_dir       {TO_XFER_BUF, FROM_XFER_BUF};
```

其实是很简单的一个枚举数据类型,含义也很简单,一个表示向 srb->request_buffer 里边 copy,TO_XFER_BUF,另一个表示从 srb->request_buffer 里边往外 copy,FROM_XFER_BUF.(题外话:XFER 就是 TRANSFER 的意思,老外喜欢这样缩写.刚进 Intel 的时候老板专门给了我一个 excel 文件,里边全是 Intel 内部广泛使用的英文缩写,不在 Intel 呆一段时间基本没法理解.)这里定义成枚举数据类型也是很有必要的,因为数据传输肯定得有且仅有一个方向,就像思念,只有一个方向,因为我思念的人未必思念我. 而此情此景,咱们传进来的是前者,所以 209 行判断之后会执行 210 行,从 buffer 里边 copy cnt 个字节到(unsigned char *) srb->request_buffer + *offset 去.cnt 在 208 行确定,min 函数不用说也知道,取最小值,认真学过谭浩强老师那本书的同志们应该不会不熟悉这样的函数,不过 linux 内核里边确实有定义这个函数,include/linux/kernel.h 中,不妨也列出来,仅供娱乐,毕竟咱们的故事太严肃了一点,娱乐性不够强:

```
203 /*
204  * min()/max() macros that also do
205  * strict type-checking.. See the
```

```
206  * "unnecessary" pointer comparison.
207  */
208 #define min(x,y) ({ \
209         typeof(x) _x = (x);     \
210         typeof(y) _y = (y);     \
211         (void) (&_x == &_y);            \
212         _x < _y ? _x : _y; })
213
214 #define max(x,y) ({ \
215         typeof(x) _x = (x);     \
216         typeof(y) _y = (y);     \
217         (void) (&_x == &_y);            \
218         _x > _y ? _x : _y; })
```

而 208 行比较的是一个 buflen,一个 srb->request_bufflen-*offset,咱们这次要传送的数据长度是 buflen,但是显然也不能够超过后者,所以就取其中小的那个,调用 memcpy copy,然后 215 行*offset 加上 copy 的字节 cnt,对于这种不采用 scatter gather 方式的传输,那么到这里就可以返回了,直接就到 278 行,返回 cnt 即可.

但是对于使用 scatter gather 方式的传输,情况当然不一样了.从 224 行开始往下看,显然如咱们所说,得定义一个 struct scatterlist 结构体的指针,由于 struct scatterlist 是和体系结构有关的,作为曾经的 Intel 人,我没有任何犹豫的应该以 i386 的为例,include/i386/scatterlist.h:

```
4 struct scatterlist {
5       struct page         *page;
6       unsigned int        offset;
7       dma_addr_t            dma_address;
8       unsigned int        length;
9 };
```

这个结构并不复杂,其中 page 指针通常指向将要进行 scatter gather 操作的 buffer.而 length 表示 buffer 的长度,offset 表示 buffer 在 page 内的偏移量.225 行,定义一个 struct scatterlist 指针 sg,然后令她指向(struct scatterlist*)srb->request_buffer+*index,搜索一下内核代码就可以知道,每次 *index 都是被初始化为 0 然后才调用 usb_stor_access_xfer_buf()的,233 行,cnt 设为 0,234 行开始进入循环,循环的条件是 cnt 小于 buflen,同时*index 小于 srb->use_sg, srb->use_sg 咱们刚才说过了,只要她不是 0,那么她里边的冬冬就代表了 scatter gather 传输时数组元素的个数.

观众朋友们请注意,225 行这里让 sg 等于 srb->request_buffer,(当然还要加上*index,如果 index 不为 0 的话),那么 request_buffer 究竟是什么?对于使用 scatter/gather 传输的情况,request_buffer 里边实际上是一个数组,每一个元素都是一个 struct scatterlist 的指针,而每一个 scatterlist 指针的 page 里边包含了一些 page(而不是一个 page),而 offset 里边包含的是每一个 DMA buffer 的总的偏移量,她由两部分组成,高位部分标志着 page 号,低位部分标志着具体某个 page 中的偏移量,高位低位由 PAGE_SHIFT 宏来划分,不同的硬件平台 PAGE_SHIFT 值不一样,因为不同的硬件平台 page 的大小也不一样,即这里的 PAGE_SIZE 不一样,目前比较前卫的硬件平台其 page size 有 4k 或者 8k 的,而 PAGE_SHIFT 也就是 12 或者 13,换言之,sg->offset 去掉低 12 位或者低 13 位就是 page 号,而低 12

位或者低 13 位恰恰是在该 page 内的偏移量.之所以可以把一个 sg->offset 起两个作用,正是因为 page size 只需要 12 位或者 13 位就足够了,或者说偏移量本身只有 12 位或者 13 位,而一个 int 型变量显然可以包含比 12 位或 13 位更多的信息.我们最终是要把 buffer(即前面说的那个 36 个 bytes 的标准的 INQUIRY data buffer)里边的冬冬 copy 至 DMA buffer 中,buffer 我们已经知道,她就是 usb_stor_access_xfer_buf()函数传递进来的参数,而 DMA buffer 在哪呢?只要我们知道她在哪个 page 中,知道她的 offset,那么有一个函数可以帮助我们获得她对应的内核虚拟地址,而这个地址正是我们需要的,有了她,我们就能调用 memcpy 函数来 copy 数据了.所以 235 行到 238 行,就是计算出究竟是哪个 page,究竟是多少 offset,后者用被赋给了 unsigned int 变量 poff,*offset 是 usb_stor_access_xfer_buf()函数传递进来的参数,她也可以控制我们要传送数据的 DMA buffer 对应的 offset,不过这里我们传递进来的是 0.所以不去 care.

239 行对 unsigned int sglen 赋值,sg->length 实际上就是 DMA buffer 的长度.所以显然我们 copy 冬冬不能超过这个长度,如果我们还指定了*offset,就表明 DMA buffer 中从*offset 开始装,那么就不能超过 sg->length-*offset.

241 行,由于我们现在还在 while 循环中,所以先看第一次执行到 241 行,这时 cnt 等于 0,buflen 就是那个 data buffer 的长度,传进来的是 36.sglen 表征了 DMA buffer 里边可以装多少,如果 sglen 比 buflen 要大,那么很好,一次就可以装满,因为这就好比 buflen 是一吨沙子,而 sglen 则表示装沙车载重两吨或者更多,比如三吨.这样 244 行 245 行,其作用就是做个标记,比如 sglen 被用来记录实际装载了多少重量的沙子,而*offset 则表征了装沙车用了多少了,如果还没卸货下次又要继续往里装那就装吧,反正没满就可以装.那么如果 sglen 比 buflen 要小或者刚好够大,那么*offset 肯定就被设为 0,因为这一车必然会被装满,要再装沙子只能调用下一辆车,所以同时 sg 和*index 也自加.

然后来看 257 行了,sglen 一开始肯定应该大于 0 吧,她表征的是实际装了多少,但是内存管理机制有一个限制,memcpy 传输不能够跨页,也就是说不能跨 page,一次最多就是把本 page 的冬冬给 copy 走,如果你的数据超过了一个 page,那你还得再 copy 一次或者多次,因为这里咱们用了循环.每一次真正 copy 的长度用 plen 来表征,所以 271 行和 272 行,cnt 是计数的,所以她要加上一个 plen,而 sglen 也是一个计数的,但是她是反着计,所以她每次要减掉一个 plen.而 poff 和 page 一个设为 0,一个自加,这个道理很简单,从下一页开头进行继续 copy.而 258 行再次调用强大的 min()函数也正是为了保证每次 copy 不能跨页.260 和 266 行这对冤家的出现,kmap()和 kunmap(),道理也很简单.这对冤家是内核中的内存管理部门提供给咱们的重要函数,其中 kmap()函数的作用就是传递给她一个 struct page 指针,她能返回这个 page 指针所指的那个 page 的内核虚拟地址,而有了这个 page 对应的虚拟地址,加上咱们前面已经知道的偏移量,咱们就可以调用 memcpy 函数来 copy 数据了.至于 kunmap,古书有云:凡是 kmap()建立起来的良好关系必须由 kunmap()来摧毁.

然后,262 到 265 行的两个 memcpy 无需再讲了.

278 行,返回实际传输的字节数.

至此,usb_stor_access_xfer_buf()这个函数的每一点每一滴都讲完了.让我们欢呼吧!这一刻,我们不是一个人在战斗.(注:应该说如果不是很有悟性的话,这段代码要看懂还是挺难的,虽说不至于让你哭得花枝乱颤,但也足以令你看得眼花缭乱.正式版中关于这个函数应该有一个比较好的图解,来描述这个双重循环,须知外循环是按 sg entry 来循环,即一个 sg entry 循环一次,而内循环是按 page 来循环,我们说过,一个 sg entry 可以有多个 page,而因为我们没有办法跨 page 映射,所以只能每个 page 映射一次,所以才需要循环.)

回到 usb_stor_set_xfer_buf()中来,也只剩下一句话了,如果我们要传递的 36 个字节还不足以填满这辆装沙车的话,那就让我们记录下这辆车还能装多少沙子吧,srb->resid 正是扮演着这个记录者的角色.

然后我们回到 fill_inquiry_response()中来,一看,嗬,这个函数也结束了.我们再一次回到了那个不死的精灵进程,usb_stor_control_thread()中来,这个函数俨然就像当年圣斗士星矢中的不死鸟一辉,总也死不了,隔一会又会出现.对于 INQUIRY 命令,咱们在 fill_inquiry_response()之后,把 srb->result 设为了 SAM_STAT_GOOD,她是 scsi 系统里边定义的一个宏,include/scsi/scsi.h 中:

```
117 /*
118  *   SCSI Architecture Model (SAM) Status codes. Taken from SAM-3 draft
119  *   T10/1561-D Revision 4 Draft dated 7th November 2002.
120  */
121 #define SAM_STAT_GOOD                0x00
```

其实就是 0,咱们完成了工作之后把 srb->result 设为 0,以后自有 scsi 那边的函数会去检测.不用咱们管了.

然后咱们进入到 381 行,更确切的是,385 行,srb->scsi_done()函数会被调用,srb->scsi_done()实际上是一个函数指针,咱们在 queuecommand 中令她等于 scsi_done,咱们被要求在完成了该做的事情之后调用这个函数,剩下的事情 scsi 核心层会去处理.

酱紫,针对 queuecommand 传递进来 INQUIRY 命令的情况,该做的就都做了.

最后单独解释一下,给那些喜欢打破沙锅问到底的无聊人士:

First of all, 好端端的传输数据为什么要分散成好多个 scatter gather list,这不是自找麻烦吗?关于这个问题,曾经我单纯的以为,也许只有支离破碎才是美.后来我意识到,SCSI 层包括 usb-storage 之所以要使用 scatter gather 是因为这一特性允许系统在一个 SCSI 命令中传输多个物理上不连续的 buffers.

关于 kmap()和 kunmap().这两个函数是干嘛的?为什么要映射?小时候妈妈没有告诉过你吗?将来你长大了,等你熟悉了 Linux 内核中的内存管理部分你就会知道,这个世界上有一个地址,叫做物理地址,还有一个地址叫做内核地址,struct page 代表的是物理地址,内核用这个结构体来代表每一个物理 page,或者说物理页,显然我们代码中不能直接操作物理地址,memcpy 这个函数根本就不认识物理地址,它只能使用内核地址.所以我们需要把物理地址映射到内核地址空间中来.kmap()从事的正是这项伟大的工作.不过写过代码的人了解 kmap()更多的和 high memory 打交道的时候认识的,此乃题外话,不表.

彼岸花的传说(The End)

解决了这个 INQUIRY 的问题,我们就可以继续往下走了,373 行,这就是真正的 Bulk 传输的地方,proto_handler()就是正儿八经的处理 SCSI 命令的函数指针.而 usb_stor_control_thread 之前的所有代码就是为了判断是不是有必要调用 proto_handler(),比如超时了,比如模块该卸载了,比如设置了断开 flag 了,比如要处理的就是这个有问题的 INQUIRY,等等这些情况都需要先排除了才有必要到达这里来执行真正的命令.实际上这就是先从宏观上来控制,保证我们走的是一条正确的道路,而不至于是沿着错误的道路走半天,毕竟,在错误的路上,就算奔跑也没有用!别说奔跑了,裸奔也没有用!

我们倒是先不急着到 proto_handler 里边去看,先把外边的代码看完.小时候,我们不都天真的以为,外面的世界很精彩么?我们先跳过 proto_handler(),把 usb_stor_control_thread()中剩下的代码看完,从而完整的了解这个守护进程究竟是如何循环的.

382 行,只要刚才的命令的结果即 srb->result 不为 DID_ABORT,那么就是成功执行了.于是我们就调用 scsi_done 函数.

387 行,SkipForAbort,就是一个行标志,对应前面的那个 goto SkipForAbort 语句. 记得大学期间,学到谭浩强大哥那本书中 goto 的时候,老师总是会告诫大家,不要乱用 goto 语句,这是不好的编程习惯,不过在 Linux 内核源代码中,goto 语句却是比比皆是,真让人为这些写 Linux 源代码的同志们的 C 语言水平担心啊,肯定在校期间没有好好念书,辜负了家长殷切的期望啊.算了,不说他们了,这些家伙少年不识愁滋味.我们继续说 Linux,396 行,前面的注释也说得很清楚了,如果是设置了 US_FLIDX_TIMED_OUT 那么就唤醒设这个 flag 的进程,其实就是唤醒 command_abort,后面我们会讲 command_abort().这里之所以判断这个 flag 而不是判断 srb->result==DID_ABORT 注释里说得也很清楚,因为有可能是在 usb 传输结束之后才收到的 abort 命令,换言之,即便你的 srb->result 不为 DID_ABORT 也可能最新又接到了 abort 的请求,所以这里就判断 abort 请求必然要设置的一个 flag 来判断.

400 行,SkipForDisconnect,这也没啥说的,和前面的 goto SkipForDisconnect 语句对应,如果要断开了,或者是一个命令执行完了,或者是 abort 了,那么最终就是把 us->srb 给置空.剩下两行的两把锁我们已经说过,会到最后统一来讲.

406 行,至此,这个守护进程就算是走了一遍了,for 循环继续,就像彼岸花,开一千年,落一千年,花叶永不相见.情不为因果,缘注定生死.不仅仅是曼珠和沙华一次次的跌入诅咒的轮回,其实世间万事万物都是轮回的,包括痛苦.

最后的最后,只剩下 422 这一行了,程序执行到这一行意味着 for 循环结束了,而从 for 循环的代码我们不难看出,结束 for 循环的只有一句话,就是那句 break,我们前面说过,它意味着模块要被卸载了.所以这里 complete_and_exit()就是唤醒别人同时结束自己,于是这一刻,usb_stor_control_thread()也就正式结束了,也许对她来说,结束就是解脱吧.此情可待成追忆,只是当时已惘然.

需要解释一下的是,complete_and_exit()这里唤醒的是谁?是 usb_stor_release_resources(),为什么要唤醒,我们稍候讲到这个函数的时候再来看.而关于这个守护进程,我们也终于讲完了,其中的 proto_handler()这两行,因为其重要性,我们单独挑出来讲.

scsi 命令之我型我秀

usb_stor_control_thread()基本讲完了,但是其中下面这几行,正是高潮中的高潮.所谓的 Bulk 传输,所谓的 Bulk-Only 协议.正是在这里体现出来的.

```
372                 /* we've got a command, let's do it! */
373                 else {
374                         US_DEBUG(usb_stor_show_command(us->srb));
375                         us->proto_handler(us->srb, us);
376                 }
```

所谓的 US_DEBUG,我们前面已经讲过,无非就是打印条是信息的.而眼下这句话就是执行 usb_stor_show_command(us->srb)这个函数,鉴于这个函数是我们自己写的,而且有点意义,所以也就列出来.这个函数定义于 drivers/usb/storage/debug.c 中,

```
55 void usb_stor_show_command(struct scsi_cmnd *srb)
56 {
57         char *what = NULL;
58         int i;
59
60         switch (srb->cmnd[0]) {
61         case TEST_UNIT_READY: what = "TEST_UNIT_READY"; break;
62         case REZERO_UNIT: what = "REZERO_UNIT"; break;
63         case REQUEST_SENSE: what = "REQUEST_SENSE"; break;
64         case FORMAT_UNIT: what = "FORMAT_UNIT"; break;
65         case READ_BLOCK_LIMITS: what = "READ_BLOCK_LIMITS"; break;
66         case REASSIGN_BLOCKS: what = "REASSIGN_BLOCKS"; break;
67         case READ_6: what = "READ_6"; break;
68         case WRITE_6: what = "WRITE_6"; break;
69         case SEEK_6: what = "SEEK_6"; break;
70         case READ_REVERSE: what = "READ_REVERSE"; break;
71         case WRITE_FILEMARKS: what = "WRITE_FILEMARKS"; break;
72         case SPACE: what = "SPACE"; break;
73         case INQUIRY: what = "INQUIRY"; break;
74         case RECOVER_BUFFERED_DATA: what = "RECOVER_BUFFERED_DATA";
break;
75         case MODE_SELECT: what = "MODE_SELECT"; break;
76         case RESERVE: what = "RESERVE"; break;
77         case RELEASE: what = "RELEASE"; break;
78         case COPY: what = "COPY"; break;
79         case ERASE: what = "ERASE"; break;
80         case MODE_SENSE: what = "MODE_SENSE"; break;
81         case START_STOP: what = "START_STOP"; break;
82         case RECEIVE_DIAGNOSTIC: what = "RECEIVE_DIAGNOSTIC"; break;
83         case SEND_DIAGNOSTIC: what = "SEND_DIAGNOSTIC"; break;
84         case ALLOW_MEDIUM_REMOVAL: what = "ALLOW_MEDIUM_REMOVAL";
break;
85         case SET_WINDOW: what = "SET_WINDOW"; break;
86         case READ_CAPACITY: what = "READ_CAPACITY"; break;
87         case READ_10: what = "READ_10"; break;
88         case WRITE_10: what = "WRITE_10"; break;
89         case SEEK_10: what = "SEEK_10"; break;
90         case WRITE_VERIFY: what = "WRITE_VERIFY"; break;
91         case VERIFY: what = "VERIFY"; break;
92         case SEARCH_HIGH: what = "SEARCH_HIGH"; break;
```

```
 93          case SEARCH_EQUAL: what = "SEARCH_EQUAL"; break;
 94          case SEARCH_LOW: what = "SEARCH_LOW"; break;
 95          case SET_LIMITS: what = "SET_LIMITS"; break;
 96          case READ_POSITION: what = "READ_POSITION"; break;
 97          case SYNCHRONIZE_CACHE: what = "SYNCHRONIZE_CACHE"; break;
 98          case LOCK_UNLOCK_CACHE: what = "LOCK_UNLOCK_CACHE"; break;
 99          case READ_DEFECT_DATA: what = "READ_DEFECT_DATA"; break;
100          case MEDIUM_SCAN: what = "MEDIUM_SCAN"; break;
101          case COMPARE: what = "COMPARE"; break;
102          case COPY_VERIFY: what = "COPY_VERIFY"; break;
103          case WRITE_BUFFER: what = "WRITE_BUFFER"; break;
104          case READ_BUFFER: what = "READ_BUFFER"; break;
105          case UPDATE_BLOCK: what = "UPDATE_BLOCK"; break;
106          case READ_LONG: what = "READ_LONG"; break;
107          case WRITE_LONG: what = "WRITE_LONG"; break;
108          case CHANGE_DEFINITION: what = "CHANGE_DEFINITION"; break;
109          case WRITE_SAME: what = "WRITE_SAME"; break;
110          case GPCMD_READ_SUBCHANNEL: what = "READ SUBCHANNEL"; break;
111          case READ_TOC: what = "READ_TOC"; break;
112          case GPCMD_READ_HEADER: what = "READ HEADER"; break;
113          case GPCMD_PLAY_AUDIO_10: what = "PLAY AUDIO (10)"; break;
114          case GPCMD_PLAY_AUDIO_MSF: what = "PLAY AUDIO MSF"; break;
115          case GPCMD_GET_EVENT_STATUS_NOTIFICATION:
116                  what = "GET EVENT/STATUS NOTIFICATION"; break;
117          case GPCMD_PAUSE_RESUME: what = "PAUSE/RESUME"; break;
118          case LOG_SELECT: what = "LOG_SELECT"; break;
119          case LOG_SENSE: what = "LOG_SENSE"; break;
120          case GPCMD_STOP_PLAY_SCAN: what = "STOP PLAY/SCAN"; break;
121          case GPCMD_READ_DISC_INFO: what = "READ DISC INFORMATION";
break;
122          case GPCMD_READ_TRACK_RZONE_INFO:
123                  what = "READ TRACK INFORMATION"; break;
124          case GPCMD_RESERVE_RZONE_TRACK: what = "RESERVE TRACK"; break;
125          case GPCMD_SEND_OPC: what = "SEND OPC"; break;
126          case MODE_SELECT_10: what = "MODE_SELECT_10"; break;
127          case GPCMD_REPAIR_RZONE_TRACK: what = "REPAIR TRACK"; break;
128          case 0x59: what = "READ MASTER CUE"; break;
129          case MODE_SENSE_10: what = "MODE_SENSE_10"; break;
130          case GPCMD_CLOSE_TRACK: what = "CLOSE TRACK/SESSION"; break;
131          case 0x5C: what = "READ BUFFER CAPACITY"; break;
132          case 0x5D: what = "SEND CUE SHEET"; break;
133          case GPCMD_BLANK: what = "BLANK"; break;
134          case MOVE_MEDIUM: what = "MOVE_MEDIUM or PLAY AUDIO (12)"; break;
135          case READ_12: what = "READ_12"; break;
```

```
136         case WRITE_12: what = "WRITE_12"; break;
137         case WRITE_VERIFY_12: what = "WRITE_VERIFY_12"; break;
138         case SEARCH_HIGH_12: what = "SEARCH_HIGH_12"; break;
139         case SEARCH_EQUAL_12: what = "SEARCH_EQUAL_12"; break;
140         case SEARCH_LOW_12: what = "SEARCH_LOW_12"; break;
141         case SEND_VOLUME_TAG: what = "SEND_VOLUME_TAG"; break;
142         case READ_ELEMENT_STATUS: what = "READ_ELEMENT_STATUS"; break;
143         case GPCMD_READ_CD_MSF: what = "READ CD MSF"; break;
144         case GPCMD_SCAN: what = "SCAN"; break;
145         case GPCMD_SET_SPEED: what = "SET CD SPEED"; break;
146         case GPCMD_MECHANISM_STATUS: what = "MECHANISM STATUS"; break;
147         case GPCMD_READ_CD: what = "READ CD"; break;
148         case 0xE1: what = "WRITE CONTINUE"; break;
149         case WRITE_LONG_2: what = "WRITE_LONG_2"; break;
150         default: what = "(unknown command)"; break;
151         }
152         US_DEBUGP("Command %s (%d bytes)\n", what, srb->cmd_len);
153         US_DEBUGP("");
154         for (i = 0; i < srb->cmd_len && i < 16; i++)
155                 US_DEBUGPX(" %02x", srb->cmnd[i]);
156         US_DEBUGPX("\n");
157 }
```

相信即使是天下无贼里边的傻根也能看懂这个函数,很简单,就是把要执行的 scsi 命令打印出来.列出这个函数没别的意思,让不熟悉 scsi 的同志们知道基本上会遇到些啥命令.显然,刚才说的那个 INQUIRY 也包含在其中的.

不过别看这个函数很 easy,你要是不熟悉 scsi 协议的话,你还真的解释不了这个函数.比如你说 srb->cmnd[]这个数组到底是什么内容?有什么格式?为啥函数一开始只判断 cmnd[0]?实不相瞒,这里边还真有学问.首先,在 scsi 的规范里边定义了一些命令,每个命令都有一定的格式,命令的字节数也有好几种,有的命令是六个字节的,有的命令是 10 个字节的,有的命令是 12 个字节的.你看,你看,下面就是摘自 scsi 规范里边的几张图,scsi 命令就该是这个样子.

Table 21 - Typical command descriptor block for six-byte commands

Bit / Byte	7	6	5	4	3	2	1	0
0	Operation code							
1	Logical unit number			(MSB)				
2	Logical block address (if required)							
3								(LSB)
4	Transfer length (if required) Parameter list length (if required) Allocation length (if required)							
5	Control							

这是 6 字节的,

Table 22 - Typical command descriptor block for ten-byte commands

Bit / Byte	7	6	5	4	3	2	1	0
0	Operation code							
1	Logical unit number			Reserved				
2	(MSB)							
3				Logical block address (if required)				
4								
5								(LSB)
6	Reserved							
7	(MSB)			Transfer length (if required) Parameter list length (if required)				
8				Allocation length (if required)				(LSB)
9	Control							

这是 10 个字节的,

Table 23 - Typical command descriptor block for twelve-byte commands

Bit / Byte	7	6	5	4	3	2	1	0
0	Operation code							
1	Logical unit number			Reserved				
2	(MSB)							
3				Logical block address (if required)				
4								
5								(LSB)
6	(MSB)							
7				Transfer length (if required) Parameter list length (if required)				
8				Allocation length (if required)				
9								(LSB)
10	Reserved							
11	Control							

这是 12 个字节的.

之所以有这好几种命令描述块,没什么特别的,也许只是想体现生物的多样性而已,又或许,印证了张爱玲的那句话,也许每一个男子全都有过这样的两个女人,至少两个.娶了红玫瑰,久而久之,红的变成了墙上的一抹蚊子血,白的还是"窗前明月光";娶了白玫瑰,白的便是衣服上的一粒饭粘子,红的却是心口上的一颗朱砂痣.

江湖中人们把这样几个字节的命令称之为 CDB, command descriptor block, 命令描述符块.而我们为 CDB 准备了一个字符数组,结构体 struct scsi_cmnd 中的 unsigned char cmnd[16],你说最大就 12 个字节,干嘛不申请一个 12 个字节的数组?给你一个建议:我记得复旦并没有一门课程叫做可持续性发

展,但是我记得我上海交大的同学有修过这样一门课程,有机会的话,去交大听一下吧.

Ok,既然这个 CDB 有 16 个字节,那么为什么我们每次都判断 cmnd[0]就够了?仔细看这三幅图,注意到那个 Operation code 了吗?没错,三幅图中的第一个字节都被称为 Operation code,换言之,不管你是什么样子的命令,你都必须在第一个字节里签上自己的名字,向世人说明你是谁.于是在 include/scsi/scsi.h 中,定义了好多好多宏,比如#define INQUIRY 0x12,又比如#define READ_6 0x08,再比如#define FORMAT_UNIT 0x04,够了,实际上操作码就相当于 scsi 命令的序列号,scsi 命令总共也就那么多,8 位的操作码已经足够表示了,因此,我们只要用一个字节就可以判断出这是哪个命令了.因为你的第一个字节就相当于你的眼睛,不管你埋藏的多深,你会发现最终总是你的眼睛背叛了你的心,这一点郑中基大概感受颇深吧.

好了,命令说完了,开始进入真正处理命令的部分了.

迷雾重重的 bulk 传输(一)

2006 年的最后一个星期,来到了北京,开始了北漂的生活.和上海不同的是,在这里待了三个月之后,发现竟然没有下过一次雨,难怪日本小孩说:"你们北京小孩真幸福,城外就是大沙漠,出了城就可以骑骆驼看日落了."不过,今天下雨了,下了大雨,好大好大,一阵阵的闪电,回家的时候下半身都湿了…(天哪,怎么写着写着又往那个方向走去了…算了,我承认我只是一个用下半身思考的男青年.)

很累,但是听着北京不眠夜,又不想入睡,听着刘杨的声音,心里感到特别温暖,这些年里,从长沙,到上海,再到北京,每每只有在夜深人静的时候,听着广播,才能忘却一些绝望.于是继续写吧,既然人生的幕布已经拉开,就一定要积极的演出;既然脚步已经跨出,风雨坎坷也不能退步;既然我已把希望播在这里,就一定要坚持到胜利的谢幕.

375 行,us->proto_handler()其实是一个函数指针,知道它指向什么吗?不要说你不知道,早年我们在 storage_probe()中,确切的说,在 get_protocol()就赋了值,当时只知道是 get protocol,却不知道究竟干什么用,现在该用上了,别以为写代码的都是傻子,一个指针要是没什么用人家才不会为它赋值呢.当初我们就讲了,对于 U 盘,proto_handler 被赋值为 usb_stor_transparent_scsi_command,所以我们来看后者吧. 后者定义于 drivers/usb/storage/protocol.c:

```
172 void usb_stor_transparent_scsi_command(struct scsi_cmnd *srb,
173                                         struct us_data *us)
174 {
175         /* send the command to the transport layer */
176         usb_stor_invoke_transport(srb, us);
177
178         if (srb->result == SAM_STAT_GOOD) {
179                 /* Fix the READ CAPACITY result if necessary */
180                 if (us->flags & US_FL_FIX_CAPACITY)
181                         fix_read_capacity(srb);
182         }
183 }
```

首先注意到的是 usb_stor_invoke_transport()函数这个函数可不简单.咱们先做好思想准备，接下来就去见识一下她的庐山真面目．她来自 drivers/usb/storage/transport.c:

```
519 /***********************************************************************
520  * Transport routines
521  ***********************************************************************/
522
523 /* Invoke the transport and basic error-handling/recovery methods
524  *
525  * This is used by the protocol layers to actually send the message to
526  * the device and receive the response.
527  */
528 void usb_stor_invoke_transport(struct scsi_cmnd *srb, struct us_data *us)
529 {
530         int need_auto_sense;
531         int result;
532
533         /* send the command to the transport layer */
534         srb->resid = 0;
535         result = us->transport(srb, us);
536
537         /* if the command gets aborted by the higher layers, we need to
538          * short-circuit all other processing
539          */
540         if (test_bit(US_FLIDX_TIMED_OUT, &us->flags)) {
541                 US_DEBUGP("-- command was aborted\n");
542                 goto Handle_Abort;
543         }
544
545         /* if there is a transport error, reset and don't auto-sense */
546         if (result == USB_STOR_TRANSPORT_ERROR) {
547                 US_DEBUGP("-- transport indicates error, resetting\n");
548                 us->transport_reset(us);
549                 srb->result = DID_ERROR << 16;
550                 return;
551         }
552
553         /* if the transport provided its own sense data, don't auto-sense */
554         if (result == USB_STOR_TRANSPORT_NO_SENSE) {
555                 srb->result = SAM_STAT_CHECK_CONDITION;
556                 return;
557         }
```

```
558
559         srb->result = SAM_STAT_GOOD;
560
561         /* Determine if we need to auto-sense
562          *
563          * I normally don't use a flag like this, but it's almost impossible
564          * to understand what's going on here if I don't.
565          */
566         need_auto_sense = 0;
567
568         /*
569          * If we're running the CB transport, which is incapable
570          * of determining status on its own, we will auto-sense
571          * unless the operation involved a data-in transfer.  Devices
572          * can signal most data-in errors by stalling the bulk-in pipe.
573          */
574         if ((us->protocol == US_PR_CB || us->protocol == US_PR_DPCM_USB) &&
575                         srb->sc_data_direction != DMA_FROM_DEVICE) {
576                 US_DEBUGP("-- CB transport device requiring auto-sense\n");
577                 need_auto_sense = 1;
578         }
579
580         /*
581          * If we have a failure, we're going to do a REQUEST_SENSE
582          * automatically.  Note that we differentiate between a command
583          * "failure" and an "error" in the transport mechanism.
584          */
585         if (result == USB_STOR_TRANSPORT_FAILED) {
586                 US_DEBUGP("-- transport indicates command failure\n");
587                 need_auto_sense = 1;
588         }
589
590         /*
591          * A short transfer on a command where we don't expect it
592          * is unusual, but it doesn't mean we need to auto-sense.
593          */
594         if ((srb->resid > 0) &&
595             !((srb->cmnd[0] == REQUEST_SENSE) ||
596               (srb->cmnd[0] == INQUIRY) ||
597               (srb->cmnd[0] == MODE_SENSE) ||
598               (srb->cmnd[0] == LOG_SENSE) ||
599               (srb->cmnd[0] == MODE_SENSE_10))) {
600                 US_DEBUGP("-- unexpectedly short transfer\n");
601         }
```

```
602
603         /* Now, if we need to do the auto-sense, let's do it */
604         if (need_auto_sense) {
605                 int temp_result;
606                 void* old_request_buffer;
607                 unsigned short old_sg;
608                 unsigned old_request_bufflen;
609                 unsigned char old_sc_data_direction;
610                 unsigned char old_cmd_len;
611                 unsigned char old_cmnd[MAX_COMMAND_SIZE];
612                 unsigned long old_serial_number;
613                 int old_resid;
614
615                 US_DEBUGP("Issuing auto-REQUEST_SENSE\n");
616
617                 /* save the old command */
618                 memcpy(old_cmnd, srb->cmnd, MAX_COMMAND_SIZE);
619                 old_cmd_len = srb->cmd_len;
620
621                 /* set the command and the LUN */
622                 memset(srb->cmnd, 0, MAX_COMMAND_SIZE);
623                 srb->cmnd[0] = REQUEST_SENSE;
624                 srb->cmnd[1] = old_cmnd[1] & 0xE0;
625                 srb->cmnd[4] = 18;
626
627                 /* FIXME: we must do the protocol translation here */
628                 if (us->subclass == US_SC_RBC || us->subclass == US_SC_SCSI)
629                         srb->cmd_len = 6;
630                 else
631                         srb->cmd_len = 12;
632
633                 /* set the transfer direction */
634                 old_sc_data_direction = srb->sc_data_direction;
635                 srb->sc_data_direction = DMA_FROM_DEVICE;
636
637                 /* use the new buffer we have */
638                 old_request_buffer = srb->request_buffer;
639                 srb->request_buffer = srb->sense_buffer;
640
641                 /* set the buffer length for transfer */
642                 old_request_bufflen = srb->request_bufflen;
643                 srb->request_bufflen = 18;
644
645                 /* set up for no scatter-gather use */
```

```
646                 old_sg = srb->use_sg;
647                 srb->use_sg = 0;
648
649                 /* change the serial number -- toggle the high bit*/
650                 old_serial_number = srb->serial_number;
651                 srb->serial_number ^= 0x80000000;
652
653                 /* issue the auto-sense command */
654                 old_resid = srb->resid;
655                 srb->resid = 0;
656                 temp_result = us->transport(us->srb, us);
657
658                 /* let's clean up right away */
659                 srb->resid = old_resid;
660                 srb->request_buffer = old_request_buffer;
661                 srb->request_bufflen = old_request_bufflen;
662                 srb->use_sg = old_sg;
663                 srb->serial_number = old_serial_number;
664                 srb->sc_data_direction = old_sc_data_direction;
665                 srb->cmd_len = old_cmd_len;
666                 memcpy(srb->cmnd, old_cmnd, MAX_COMMAND_SIZE);
667
668                 if (test_bit(US_FLIDX_TIMED_OUT, &us->flags)) {
669                         US_DEBUGP("-- auto-sense aborted\n");
670                         goto Handle_Abort;
671                 }
672                 if (temp_result != USB_STOR_TRANSPORT_GOOD) {
673                         US_DEBUGP("-- auto-sense failure\n");
674
675                         /* we skip the reset if this happens to be a
676                          * multi-target device, since failure of an
677                          * auto-sense is perfectly valid
678                          */
679                         if (!(us->flags & US_FL_SCM_MULT_TARG))
680                                 us->transport_reset(us);
681                         srb->result = DID_ERROR << 16;
682                         return;
683                 }
684
685                 US_DEBUGP("-- Result from auto-sense is %d\n", temp_result);
686                 US_DEBUGP("-- code: 0x%x, key: 0x%x, ASC: 0x%x, ASCQ:
0x%x\n",
687                           srb->sense_buffer[0],
688                           srb->sense_buffer[2] & 0xf,
```

```
689                             srb->sense_buffer[12],
690                             srb->sense_buffer[13]);
691 #ifdef CONFIG_USB_STORAGE_DEBUG
692                 usb_stor_show_sense(
693                             srb->sense_buffer[2] & 0xf,
694                             srb->sense_buffer[12],
695                             srb->sense_buffer[13]);
696 #endif
697
698                 /* set the result so the higher layers expect this data */
699                 srb->result = SAM_STAT_CHECK_CONDITION;
700
701                 /* If things are really okay, then let's show that.  Zero
702                  * out the sense buffer so the higher layers won't realize
703                  * we did an unsolicited auto-sense. */
704                 if (result == USB_STOR_TRANSPORT_GOOD &&
705                         /* Filemark 0, ignore EOM, ILI 0, no sense */
706                                 (srb->sense_buffer[2] & 0xaf) == 0 &&
707                         /* No ASC or ASCQ */
708                                 srb->sense_buffer[12] == 0 &&
709                                 srb->sense_buffer[13] == 0) {
710                         srb->result = SAM_STAT_GOOD;
711                         srb->sense_buffer[0] = 0x0;
712                 }
713         }
714
715         /* Did we transfer less than the minimum amount required? */
716         if (srb->result == SAM_STAT_GOOD &&
717                         srb->request_bufflen - srb->resid < srb->underflow)
718                 srb->result = (DID_ERROR << 16) | (SUGGEST_RETRY << 24);
719
720         return;
721
722         /* abort processing: the bulk-only transport requires a reset
723          * following an abort */
724   Handle_Abort:
725         srb->result = DID_ABORT << 16;
726         if (us->protocol == US_PR_BULK)
727                 us->transport_reset(us);
728 }
```

好家伙,洋洋洒洒两百余行的一个函数,怎一个壮观二字了得! 欧阳修大哥曾经的一首蝶恋花把这个复杂的函数可谓描绘的淋漓尽致.

庭院深深深几许?杨柳堆烟,帘幕无重数.

玉勒雕鞍游冶处,楼高不见章台路.

雨横风狂三月暮,门掩黄昏,无计留春住.

泪眼问花花不语,乱红飞过秋千去.

上片深几许,无重数,不见章台路正是写的这段代码的复杂,调用关系一层又一层,让很多新手看了感觉无可奈何,如果没有高人的指导,盲目的去阅读代码或者去看那些很垃圾的书,那么无异于对美好生命的戕害.下片狂风暴雨正是比喻这种盲目的学习的害处,词中以花被摧残喻读代码者自己青春被毁.韶华空逝,人生易老.何必呢?

迷雾重重的 Bulk 传输(二)

其实故事已经讲了很久了,但如果你觉得到这里你已经把故事都看明白了,那么你错了.不仅仅是错了,你这种想法无异于就是,手里拿着一把刀,就以为自己是刀郎,手里举着一个窝头,就以为自己是托塔李天王.不信,我们就继续看,先看 535 行,us->transport(),这个函数指针同样是在 storage_probe 的时候被赋值,对于咱们的 u 盘,她遵守的是 Bulk-Only 协议,因此 us->transport()被赋值为 usb_stor_Bulk_transport().来看 usb_stor_Bulk_transport(),她同样来自 drivers/usb/storage/transport.c:

```
948 int usb_stor_Bulk_transport(struct scsi_cmnd *srb, struct us_data *us)
949 {
950         struct bulk_cb_wrap *bcb = (struct bulk_cb_wrap *) us->iobuf;
951         struct bulk_cs_wrap *bcs = (struct bulk_cs_wrap *) us->iobuf;
952         unsigned int transfer_length = srb->request_bufflen;
953         unsigned int residue;
954         int result;
955         int fake_sense = 0;
956         unsigned int cswlen;
957
958         /* set up the command wrapper */
959         bcb->Signature = cpu_to_le32(US_BULK_CB_SIGN);
960         bcb->DataTransferLength = cpu_to_le32(transfer_length);
961         bcb->Flags = srb->sc_data_direction == DMA_FROM_DEVICE ? 1 << 7 : 0;
962         bcb->Tag = srb->serial_number;
963         bcb->Lun = srb->device->lun;
964         if (us->flags & US_FL_SCM_MULT_TARG)
965                 bcb->Lun |= srb->device->id << 4;
966         bcb->Length = srb->cmd_len;
967
```

```
968         /* copy the command payload */
969         memset(bcb->CDB, 0, sizeof(bcb->CDB));
970         memcpy(bcb->CDB, srb->cmnd, bcb->Length);
971
972         /* send it to out endpoint */
973         US_DEBUGP("Bulk Command S 0x%x T 0x%x L %d F %d Trg %d LUN %d
CL %d\n",
974                         le32_to_cpu(bcb->Signature), bcb->Tag,
975                         le32_to_cpu(bcb->DataTransferLength), bcb->Flags,
976                         (bcb->Lun >> 4), (bcb->Lun & 0x0F),
977                         bcb->Length);
978         result = usb_stor_bulk_transfer_buf(us, us->send_bulk_pipe,
979                                 bcb, US_BULK_CB_WRAP_LEN, NULL);
980         US_DEBUGP("Bulk command transfer result=%d\n", result);
981         if (result != USB_STOR_XFER_GOOD)
982                 return USB_STOR_TRANSPORT_ERROR;
983
984         /* DATA STAGE */
985         /* send/receive data payload, if there is any */
986
987         /* Genesys Logic interface chips need a 100us delay between the
988          * command phase and the data phase */
989         if (us->pusb_dev->descriptor.idVendor == USB_VENDOR_ID_GENESYS)
990                 udelay(100);
991
992         if (transfer_length) {
993                 unsigned int pipe = srb->sc_data_direction ==
DMA_FROM_DEVICE ?
994                                 us->recv_bulk_pipe : us->send_bulk_pipe;
995                 result = usb_stor_bulk_transfer_sg(us, pipe,
996                                         srb->request_buffer, transfer_length,
997                                         srb->use_sg, &srb->resid);
998                 US_DEBUGP("Bulk data transfer result 0x%x\n", result);
999                 if (result == USB_STOR_XFER_ERROR)
1000                         return USB_STOR_TRANSPORT_ERROR;
1001
1002                 /* If the device tried to send back more data than the
1003                  * amount requested, the spec requires us to transfer
1004                  * the CSW anyway.  Since there's no point retrying the
1005                  * the command, we'll return fake sense data indicating
1006                  * Illegal Request, Invalid Field in CDB.
1007                  */
1008                 if (result == USB_STOR_XFER_LONG)
1009                         fake_sense = 1;
```

```
1010        }
1011
1012        /* See flow chart on pg 15 of the Bulk Only Transport spec for
1013         * an explanation of how this code works.
1014         */
1015
1016        /* get CSW for device status */
1017        US_DEBUGP("Attempting to get CSW...\n");
1018        result = usb_stor_bulk_transfer_buf(us, us->recv_bulk_pipe,
1019                                bcs, US_BULK_CS_WRAP_LEN, &cswlen);
1020
1021        /* Some broken devices add unnecessary zero-length packets to the
1022         * end of their data transfers.  Such packets show up as 0-length
1023         * CSWs.  If we encounter such a thing, try to read the CSW again.
1024         */
1025        if (result == USB_STOR_XFER_SHORT && cswlen == 0) {
1026                US_DEBUGP("Received 0-length CSW; retrying...\n");
1027                result = usb_stor_bulk_transfer_buf(us, us->recv_bulk_pipe,
1028                                bcs, US_BULK_CS_WRAP_LEN, &cswlen);
1029        }
1030
1031        /* did the attempt to read the CSW fail? */
1032        if (result == USB_STOR_XFER_STALLED) {
1033
1034                /* get the status again */
1035                US_DEBUGP("Attempting to get CSW (2nd try)...\n");
1036                result = usb_stor_bulk_transfer_buf(us, us->recv_bulk_pipe,
1037                                bcs, US_BULK_CS_WRAP_LEN, NULL);
1038        }
1039
1040        /* if we still have a failure at this point, we're in trouble */
1041        US_DEBUGP("Bulk status result = %d\n", result);
1042        if (result != USB_STOR_XFER_GOOD)
1043                return USB_STOR_TRANSPORT_ERROR;
1044
1045        /* check bulk status */
1046        residue = le32_to_cpu(bcs->Residue);
1047        US_DEBUGP("Bulk Status S 0x%x T 0x%x R %u Stat 0x%x\n",
1048                        le32_to_cpu(bcs->Signature), bcs->Tag,
1049                        residue, bcs->Status);
1050        if ((bcs->Signature != cpu_to_le32(US_BULK_CS_SIGN) &&
1051                    bcs->Signature !=
cpu_to_le32(US_BULK_CS_OLYMPUS_SIGN)) ||
1052                        bcs->Tag != srb->serial_number ||
```

```
1053                    bcs->Status > US_BULK_STAT_PHASE) {
1054            US_DEBUGP("Bulk logical error\n");
1055            return USB_STOR_TRANSPORT_ERROR;
1056    }
1057
1058    /* try to compute the actual residue, based on how much data
1059     * was really transferred and what the device tells us */
1060    if (residue) {
1061            if (!(us->flags & US_FL_IGNORE_RESIDUE) ||
1062                            srb->sc_data_direction == DMA_TO_DEVICE) {
1063                    residue = min(residue, transfer_length);
1064                    srb->resid = max(srb->resid, (int) residue);
1065            }
1066    }
1067
1068    /* based on the status code, we report good or bad */
1069    switch (bcs->Status) {
1070            case US_BULK_STAT_OK:
1071                    /* device babbled -- return fake sense data */
1072                    if (fake_sense) {
1073                            memcpy(srb->sense_buffer,
1074                                    usb_stor_sense_invalidCDB,
1075                                    sizeof(usb_stor_sense_invalidCDB));
1076                            return USB_STOR_TRANSPORT_NO_SENSE;
1077                    }
1078
1079                    /* command good -- note that data could be short */
1080                    return USB_STOR_TRANSPORT_GOOD;
1081
1082            case US_BULK_STAT_FAIL:
1083                    /* command failed */
1084                    return USB_STOR_TRANSPORT_FAILED;
1085
1086            case US_BULK_STAT_PHASE:
1087                    /* phase error -- note that a transport reset will be
1088                     * invoked by the invoke_transport() function
1089                     */
1090                    return USB_STOR_TRANSPORT_ERROR;
1091    }
1092
1093    /* we should never get here, but if we do, we're in trouble */
1094    return USB_STOR_TRANSPORT_ERROR;
1095 }
```

看傻了吧,这个函数也不是好惹的.但正是这个函数掀开了我们 bulk 传输的新篇章.

迷雾重重的 Bulk 传输(三)

在 usb_stor_Bulk_transport()中，古人一针见血的为我们指出了这个函数中调用的第一个最重要的函数,那就是 usb_stor_bulk_transfer_buf().仍然是来自 drivers/usb/stroage/transport.c.

```
 409 /*
 410  * Transfer one buffer via bulk pipe, without timeouts, but allowing early
 411  * termination.  Return codes are USB_STOR_XFER_xxx.  If the bulk pipe
 412  * stalls during the transfer, the halt is automatically cleared.
 413  */
 414 int usb_stor_bulk_transfer_buf(struct us_data *us, unsigned int pipe,
 415         void *buf, unsigned int length, unsigned int *act_len)
 416 {
 417         int result;
 418
 419         US_DEBUGP("%s: xfer %u bytes\n", __FUNCTION__, length);
 420
 421         /* fill and submit the URB */
 422         usb_fill_bulk_urb(us->current_urb, us->pusb_dev, pipe, buf, length,
 423                       usb_stor_blocking_completion, NULL);
 424         result = usb_stor_msg_common(us, 0);
 425
 426         /* store the actual length of the data transferred */
 427         if (act_len)
 428                 *act_len = us->current_urb->actual_length;
 429         return interpret_urb_result(us, pipe, length, result,
 430                         us->current_urb->actual_length);
 431 }
```

一路走来的同志们不会对这里这个 usb_fill_bulk_urb()完全陌生.我们的确是第一次见这个函数,但是此前我们有见过 usb_fill_control_urb(),除此之外还有一个叫做 usb_fill_int_urb()的函数,不用说,这几个函数是差不多的,只不过她们分别对应 usb 传输模式中的 bulk,control,interrupt.唯一一处和 usb_fill_control_urb 不同的便是 bulk 传输不需要有一个 setup_packet.具体来看,usb_fill_bulk_urb()定义于 include/linux/usb.h:

```
845 /**
 846  * usb_fill_bulk_urb - macro to help initialize a bulk urb
 847  * @urb: pointer to the urb to initialize.
 848  * @dev: pointer to the struct usb_device for this urb.
 849  * @pipe: the endpoint pipe
 850  * @transfer_buffer: pointer to the transfer buffer
 851  * @buffer_length: length of the transfer buffer
 852  * @complete: pointer to the usb_complete_t function
```

```
853  * @context: what to set the urb context to.
854  *
855  * Initializes a bulk urb with the proper information needed to submit it
856  * to a device.
857  */
858 static inline void usb_fill_bulk_urb (struct urb *urb,
859                                       struct usb_device *dev,
860                                       unsigned int pipe,
861                                       void *transfer_buffer,
862                                       int buffer_length,
863                                       usb_complete_t complete,
864                                       void *context)
865 {
866         spin_lock_init(&urb->lock);
867         urb->dev = dev;
868         urb->pipe = pipe;
869         urb->transfer_buffer = transfer_buffer;
870         urb->transfer_buffer_length = buffer_length;
871         urb->complete = complete;
872         urb->context = context;
873 }
```

看过了那个 usb_fill_control_urb 之后看这个函数应该是很简单的了.结合上面调用这个函数的代码,可知,urb->complete 被赋值为 usb_stor_blocking_completion,不用说,这个函数之后肯定会被调用.正如上次控制传输中所讲的那样.

424 行,usb_stor_msg_common()这个函数再一次被调用,年年岁岁花相似,岁岁年年人不同,urb 还像上次那样被被提交,然后核心层去调度,去执行她.如果结果是提交成功了,那么返回值 result 将是 0.而 act_len 将记录实际传输的长度.不过光看这两个函数其实看不出什么,我们必须结合上下文来看.换句话说,我们需要结合 usb_stor_Bulk_transport()中 usb_stor_bulk_transfer_buf 被调用的上下文,对比形参和实参来看,才能真的明白,才能拨开这浓浓的迷雾.

usb_stor_Bulk_transport()函数中,978 行,usb_stor_bulk_transfer_buf()函数得到调用.第一个参数,us,无需多说,第二个参数,us->send_bulk_pipe,作为 u 盘来说,她除了有一个控制管道以外,还会有两个 bulk 管道,一个是 In,一个是 Out,经历过此前的风风雨雨,咱们已经对 usb 中那些名词不再有神秘感,所谓管道无非就是一个 unsigned int 类型的数.us->send_bulk_pipe 和接下来我们立刻会邂逅的 us->recv_bulk_pipe 都是在曾经那个令人回味的 storage_probe()中调用 get_pipes()函数获得的.然后第三个参数 bcb,这是什么玩艺?嘿嘿,看仔细了.

950 行,定义了这么一个指针 bcb,是 struct bulk_cb_wrap 结构体的指针,这是一个专门为 bulk only 协议特别准备的数据结构,来自 drivers/usb/storage/transport.h:

```
80 /*
81  * Bulk only data structures
82  */
```

```
83
84 /* command block wrapper */
85 struct bulk_cb_wrap {
86         __le32   Signature;              /* contains 'USBC' */
87         __u32    Tag;                    /* unique per command id */
88         __le32   DataTransferLength;     /* size of data */
89         __u8     Flags;                  /* direction in bit 0 */
90         __u8     Lun;                    /* LUN normally 0 */
91         __u8     Length;                 /* of of the CDB */
92         __u8     CDB[16];                /* max command */
93 };
```

眼疾手快的同志们一定已经看到,同一文件中还定义了另一个数据结构,struct bulk_cs_wrap,

```
100 /* command status wrapper */
101 struct bulk_cs_wrap {
102         __le32   Signature;              /* should = 'USBS' */
103         __u32    Tag;                    /* same as original command */
104         __le32   Residue;                /* amount not transferred */
105         __u8     Status;                 /* see below */
106         __u8     Filler[18];
107 };
```

这两个数据结构对应于江湖中传说的 CBW 和 CSW,即 command block wrapper 和 command status wrapper.事到如今,咱们需要关注一下 usb mass storage bulk only transport 协议了,因为 u 盘是按照这个协议规定的方式去传输数据的,Bulk only 传输方式是这样进行的,首先由 host 给设备发送一个 CBW,然后 device 接收到了 CBW,她会进行解释,然后按照 CBW 里定义的那样去执行她该做的事情,然后她会给 host 返回一个 CSW.CBW 实际上是命令的封装包,而 CSW 实际上是状态的封装包.(命令执行后的状态,成功,失败,浪里看不出有未有...所以需要使用这么一个状态包). 至于你说为啥要把命令以及反映命令执行成功与否的状态包装起来,那很简单, 包装是房子富丽堂皇的外壳,包装是丑妇手上绚丽的太阳伞,包装是模特在舞台上走出的一字猫步.爱美之心人皆有之,设计 spec 的人也不例外.

这时候我们就可以看看 usb_stor_Bulk_transport()函数中,调用 usb_stor_bulk_transfer_buf()之前的那几行究竟在干嘛了.很明显,这些行都是在为 usb_stor_bulk_transfer_buf()这个函数调用做准备,对应于那些三级片中的前戏,真正精彩的部分还是在 usb_stor_bulk_transfer_buf()中,但前戏的存在必然是合理的,毕竟,子曾经曰过:没有激情的拥吻,何来床上的翻滚.所以我们来具体看看这部分前戏.(唉,沦为今天这样一个优秀的大学生,不能怪复旦,主要是自己没有坚强的意志品质啊!)

950 行,struct bulk_cb_wrap *bcb,赋值为(struct bulk_cb_wrap *) us->iobuf,951 行,struct bulk_cs_wrap *bcs,也赋值为(struct bulk_cb_wrap *) us->iobuf,然后定义一个 unsigned int 的变量 transfer_length,赋值为 srb->request_bufflen.然后接下来就开始为 bcb 的各成员赋值了.我们不妨看一下 usb mass storage spec 中的两张图片,一张是 CBW 的格式,一张是 CSW 的格式,

Table 5.1 - Command Block Wrapper

Byte \ bit	7	6	5	4	3	2	1	0
0-3	*dCBWSignature*							
4-7	*dCBWTag*							
8-11 (08h-0Bh)	*dCBWDataTransferLength*							
12 (0Ch)	*bmCBWFlags*							
13 (0Dh)	Reserved (0)					*bCBWLUN*		
14 (0Eh)	Reserved (0)					*bCBWCBLength*		
15-30 (0Fh-1Eh)	*CBWCB*							

这是 CBW,

Table 5.2 - Command Status Wrapper

Byte \ bit	7	6	5	4	3	2	1	0
0-3	*dCSWSignature*							
4-7	*dCSWTag*							
8-11 (8-Bh)	*dCSWDataResidue*							
12 (Ch)	*bCSWStatus*							

而这,就是 CSW.

959 行, bcb->Signature=cpu_to_le32(US_BULK_CB_SIGN),Signature 对应 usb mass storage spec 中 CBW 的前四个 bytes,即 dCBWSignature,US_BULK_CB_SIGN 这个宏定义于 drivers/usb/storage/transport.h 中,

```
96 #define US_BULK_CB_SIGN         0x43425355      /*spells out USBC */
```

也不知道是哪个傻 X 规定的,只有把 dCBWSignature 里边写上 43425355h 才能标志着个数据包是一个 CBW.另外,CBW 的传输全是遵守 little endian 的,所以 cpu_to_le32()这个宏需要使用,来转换数据格式.

然后 bcb->DataTransferLength 对应 CBW 中的 dCBWDataTransferLength.这个就是标志 host 希望这个 endpoint 传输多少个 bytes 的数据.这里把 cpu_to_le32(transfer_length)赋给了她.而 transfer_length 刚才已经说了,就是 srb->request_bufflen. 其实这几个变量名换来换去最重要记录的还是同一样东西.

bcb->Flags,对应于 CBW 中的 bmCBWFlags,bcb->Flags = srb->sc_data_direction == DMA_FROM_DEVICE ? 1 << 7 : 0;这个表明的是数据传输的方向,DMA_FROM_DEVICE 咱们前面讲过,表示数据是从设备传向主存.而 bmCBWFlags 是 8 位的,其中 bit7 表示方向,0 表示 Data-Out,即 from host to the device,1 表示 Data-in,即 from the device to the host.所以这里如果是 1 的话要左移 7 位.

bcb->Tag = srb->serial_number,这个 Tag 对应 CBW 中的 dCBWTag,这个 dCBWTag 的意义在于,host 会 send 出去,而 device 将会把这个 Tag 的内容给打印出来,确切的说,device 会回送一个 CSW 回来,而在 CSW 中会有一个 dCSWTag,她的内容和这个 dCBWTag 是一样的,所以实际上这就跟接头暗号似的.每一个 scsi 命令都会被赋上一个 serial_number,这里把她用在了 Tag 上.

bcb->Lun = srb->device->lun,很简单,对应 CBW 中的 bCBWLUN,就是表征这个命令是发给哪个 LUN 的,我们知道一个设备如果支持多个 LUN,那么显然每个 LUN 会有一个编号.比如咱们要读写 u 盘上的某个分区,那么当然得指明是哪个分区了.如果设备不支持多个 lun,那么这儿会被设置为 0.不过需要注意,这里 bcb->Lun 和 CBW 中的 bCBWLUN 并不完全对应,bCBWLUN 只有 4 个 bit,而咱们这里定义的时候,Lun 是有 8 位的,低四位用来对应 bCBWLUN,而高四位实际上是用来表征 target id 的.所以接下来判断 us->flags 里边设了 US_FL_SCM_MULT_TARG 这个标志没有,如果有,说明是支持多个 target 的,于是就要记录下是哪个 target.

bcb->Length = srb->cmd_len,这个对应于 CBW 中的 bCBWCBLength,即命令的有效长度,单位是 bytes. scsi 命令的有效长度只能是 1 到 16 之间.接下来有个 CDB 数组,数组共 16 个元素,理由咱们刚才讲 struct scsi_cmnd 中的 cmnd 就已经说过了.而 969 行,970 行正是把命令 srb->cmnd 数组的内容 copy 至 bcb->CDB 中.

这时候,前戏结束了,usb_stor_bulk_transfer_buf 正式被调用了.传递给她的第三个参数正是 bcb,而第四个参数是 US_BULK_CB_WRAP_LEN,她也是定义于 drivers/usb/storage/transport.h 中,

```
95 #define US_BULK_CB_WRAP_LEN     31
```

31 就是 CBW 的长度,CBW 正是 31 个 bytes.而 usb_stor_bulk_transfer_buf 的所作所为咱们是非常清楚地,无非就是提交这么一个 urb,然后就不用管事了,就等结果呗.而最终的 result 是由 interpret_urb_result()返回的,传输正确那么会返回 USB_STOR_XFER_GOOD,而如果不正确,那么 usb_stor_Bulk_transport()中就直接返回了,返回值是 USB_STOR_TRANSPORT_ERROR.如果正确,那么继续往下走,这才到真正的数据传输阶段. 在真正开始将数据传输阶段之前,我们先来看看 interpret_urb_result()函数.

迷雾重重的 Bulk 传输(四)

在讲数据传输阶段之前,先解决刚才的历史遗留问题. usb_stor_bulk_transfer_buf()中,429 行,有一个很有趣的函数 interpret_urb_result()被调用.这个函数同样来自 drivers/usb/storage/transport.c:

```
277 /*
278  * Interpret the results of a URB transfer
279  *
280  * This function prints appropriate debugging messages, clears halts on
```

```
281  * non-control endpoints, and translates the status to the corresponding
282  * USB_STOR_XFER_xxx return code.
283  */
284 static int interpret_urb_result(struct us_data *us, unsigned int pipe,
285                 unsigned int length, int result, unsigned int partial)
286 {
287         US_DEBUGP("Status code %d; transferred %u/%u\n",
288                         result, partial, length);
289         switch (result) {
290
291         /* no error code; did we send all the data? */
292         case 0:
293                 if (partial != length) {
294                         US_DEBUGP("-- short transfer\n");
295                         return USB_STOR_XFER_SHORT;
296                 }
297
298                 US_DEBUGP("-- transfer complete\n");
299                 return USB_STOR_XFER_GOOD;
300
301         /* stalled */
302         case -EPIPE:
303                 /* for control endpoints, (used by CB[I]) a stall indicates
304                  * a failed command */
305                 if (usb_pipecontrol(pipe)) {
306                         US_DEBUGP("-- stall on control pipe\n");
307                         return USB_STOR_XFER_STALLED;
308                 }
309
310                 /* for other sorts of endpoint, clear the stall */
311                 US_DEBUGP("clearing endpoint halt for pipe 0x%x\n", pipe);
312                 if (usb_stor_clear_halt(us, pipe) < 0)
313                         return USB_STOR_XFER_ERROR;
314                 return USB_STOR_XFER_STALLED;
315
316         /* timeout or excessively long NAK */
317         case -ETIMEDOUT:
318                 US_DEBUGP("-- timeout or NAK\n");
319                 return USB_STOR_XFER_ERROR;
320
321         /* babble - the device tried to send more than we wanted to read */
322         case -EOVERFLOW:
323                 US_DEBUGP("-- babble\n");
324                 return USB_STOR_XFER_LONG;
```

```
325
326         /* the transfer was cancelled by abort, disconnect, or timeout */
327         case -ECONNRESET:
328                 US_DEBUGP("-- transfer cancelled\n");
329                 return USB_STOR_XFER_ERROR;
330
331         /* short scatter-gather read transfer */
332         case -EREMOTEIO:
333                 US_DEBUGP("-- short read transfer\n");
334                 return USB_STOR_XFER_SHORT;
335
336         /* abort or disconnect in progress */
337         case -EIO:
338                 US_DEBUGP("-- abort or disconnect in progress\n");
339                 return USB_STOR_XFER_ERROR;
340
341         /* the catch-all error case */
342         default:
343                 US_DEBUGP("-- unknown error\n");
344                 return USB_STOR_XFER_ERROR;
345         }
346 }
```

应该说这个函数的作用是一目了然的.就是根据传进来的参数 result 进行判断,从而采取相应的行动.partial 是实际传输的长度,而 length 是期望传输的长度,传输结束了当然要比较这两者,因为有所期待,才会失望.result 是 usb_stor_msg_common()函数的返回值,其实就是状态代码,如果为 0 说明一切都很顺利,结果也是成功的,287 这行,打印出 result 来,同时打印出 partial 和 length 的比,注意两个%u 中间那个"/",就是除号,或者说分割分子和分母的符号.然后通过一个 switch 语句判断 result,为 0,说明至少数据有传输,然后有两种情况,于是返回不同的值,一个是 USB_STOR_XFER_SHORT,另一个是 USB_STOR_XFER_GOOD.至于返回这些值之后会得到什么反应,让我们边走边看.目前只需要知道的是,对于真正传输完全令人满意的情况,返回值只能是 USB_STOR_XFER_GOOD.返回其它值都说明有问题.而这里作为传递给 switch 的 result,实际上是 usb core 那一层传过来的值.而我们注意到,interpret_urb_result 这个函数整个是被作为一个返回值出现在 usb_stor_bulk_transfer_buf()中的,换言之,前者返回之后,后者也马上就返回了,即再次返回到了 usb_stor_Bulk_transport()中来了.因此,我们把视线拉回 usb_stor_Bulk_transport(),981 行,如果 result 不为 USB_STOR_XFER_GOOD,就说明多少有些问题,于是索性 usb_stor_Bulk_transport()也返回,没必要再进行下一阶段的传输了.否则,才可以进行下一阶段.

什么下一阶段?所谓的 Bulk Only 传输,总共就是三个阶段,命令传输阶段,数据传输阶段,状态传输阶段.很显然,真正最有意义的阶段就是数据传输阶段,而在此之前,我们已经讲了第一阶段,即命令传输阶段.下面我们可以来看数据阶段.

989 行,990 行,实在没话可说,USB_VENDOR_ID_GENESYS 代表某公司,这公司的产品在命令阶段和数据阶段居然还得延时 100 微秒.没办法,谁要我们生活在一个宣扬个性的时代呢.体谅他们吧,没有哪一

种胭脂能涂抹时间,没有哪一件服装能掩饰灵魂,没有哪一套古籍能装潢空虚,没有哪一家公司能说自己的产品是完美的,是没有缺陷的.

992 行,transfer_length 可能为 0,因为有的命令她并不需要您传输数据,所以她没有数据阶段.而对于那些有数据阶段的情况,咱们进入 if 这一段.

993 行,没什么可说的,就是根据数据传输方向确定用接收 pipe 还是发送 pipe.

然后,995 行,usb_stor_bulk_transfer_sg()这个函数是真正的执行 bulk 数据传输了.这个函数来自 drivers/usb/storage/transport.c 中:

```
484 /*
485  * Transfer an entire SCSI command's worth of data payload over the bulk
486  * pipe.
487  *
488  * Note that this uses usb_stor_bulk_transfer_buf() and
489  * usb_stor_bulk_transfer_sglist() to achieve its goals --
490  * this function simply determines whether we're going to use
491  * scatter-gather or not, and acts appropriately.
492  */
493 int usb_stor_bulk_transfer_sg(struct us_data* us, unsigned int pipe,
494                 void *buf, unsigned int length_left, int use_sg, int *residual)
495 {
496         int result;
497         unsigned int partial;
498
499         /* are we scatter-gathering? */
500         if (use_sg) {
501                 /* use the usb core scatter-gather primitives */
502                 result = usb_stor_bulk_transfer_sglist(us, pipe,
503                                 (struct scatterlist *) buf, use_sg,
504                                 length_left, &partial);
505                 length_left -= partial;
506         } else {
507                 /* no scatter-gather, just make the request */
508                 result = usb_stor_bulk_transfer_buf(us, pipe, buf,
509                                 length_left, &partial);
510                 length_left -= partial;
511         }
512
513         /* store the residual and return the error code */
514         if (residual)
515                 *residual = length_left;
516         return result;
517 }
```

注释说得很清楚,这个函数是一个壳,真正干活的是她所调用或者说利用的那两个函数.usb_stor_bulk_transfer_sglist()和 usb_stor_bulk_transfer_buf().后者咱们刚才已经遇到过了,而前者是专门为 scatter-gather 传输准备的函数,她也来自 drivers/usb/storage/transport.c 中:

```
433 /*
434  * Transfer a scatter-gather list via bulk transfer
435  *
436  * This function does basically the same thing as usb_stor_bulk_transfer_buf()
437  * above, but it uses the usbcore scatter-gather library.
438  */
439 int usb_stor_bulk_transfer_sglist(struct us_data *us, unsigned int pipe,
440                 struct scatterlist *sg, int num_sg, unsigned int length,
441                 unsigned int *act_len)
442 {
443         int result;
444
445         /* don't submit s-g requests during abort/disconnect processing */
446         if (us->flags & ABORTING_OR_DISCONNECTING)
447                 return USB_STOR_XFER_ERROR;
448
449         /* initialize the scatter-gather request block */
450         US_DEBUGP("%s: xfer %u bytes, %d entries\n", __FUNCTION__,
451                         length, num_sg);
452         result = usb_sg_init(&us->current_sg, us->pusb_dev, pipe, 0,
453                         sg, num_sg, length, SLAB_NOIO);
454         if (result) {
455                 US_DEBUGP("usb_sg_init returned %d\n", result);
456                 return USB_STOR_XFER_ERROR;
457         }
458
459         /* since the block has been initialized successfully, it's now
460          * okay to cancel it */
461         set_bit(US_FLIDX_SG_ACTIVE, &us->flags);
462
463         /* did an abort/disconnect occur during the submission? */
464         if (us->flags & ABORTING_OR_DISCONNECTING) {
465
466                 /* cancel the request, if it hasn't been cancelled already */
467                 if (test_and_clear_bit(US_FLIDX_SG_ACTIVE, &us->flags)) {
468                         US_DEBUGP("-- cancelling sg request\n");
469                         usb_sg_cancel(&us->current_sg);
470                 }
471         }
472
```

```
473         /* wait for the completion of the transfer */
474         usb_sg_wait(&us->current_sg);
475         clear_bit(US_FLIDX_SG_ACTIVE, &us->flags);
476
477         result = us->current_sg.status;
478         if (act_len)
479                 *act_len = us->current_sg.bytes;
480         return interpret_urb_result(us, pipe, length, result,
481                         us->current_sg.bytes);
482 }
```

usb_stor_bulk_transfer_sg()函数中,判断 use_sg 是否为 0,从而确定是否用 scatter-gather.对于 use_sg 等于 0 的情况,表示不用 scatter-gather,那么调用 usb_stor_bulk_transfer_buf()发送 scsi 命令.实际传递的数据长度用 partial 记录,然后 length_left 就记录还剩下多少没传递,初值当然就是期望传递的那个长度.每次减去实际传递的长度即可.对于 use_sg 不等于 0 的情况,usb_stor_bulk_transfer_sglist()函数被调用.我们来看这个函数.

迷雾重重的 Bulk 传输(五)

usb_stor_bulk_transfer_sglist()这个函数有一定的蛊惑性,我们前面说过,之所以采用 sglist,就是为了提高传输效率.我们更知道,sg 的目的就是让一堆不连续的 buffers 一次 DMA 操作就都传输出去.其实在 usb 的故事中,事情并非如此.不过如果你对 usb core 里边的行为不关心的话,那就无所谓了.有些事情,你不知道也好.

446 行,447 行,aborting 了或者 disconnecting 了,就不要传递数据了.

然后 452 行,usb_sg_init()函数被调用,这个函数来自 drivers/usb/core/message.c,也就是说,她是 usb 核心层提供的函数,干嘛用的?初始化 sg 请求.其第一个参数是 struct usb_sg_request 结构体的指针.这里咱们传递了 us->current_sg 的地址给她,这里 us->current_sg 第一次派上用场,所以咱们需要隆重的介绍一下.在 struct us_data 中,定义了这么一个成员,struct usb_sg_request current_sg.曾几何时咱们见到过current_urb,这里又来了一个current_sg.也许你感觉很困惑,这很正常,色彩容易让人炫目,文字容易让人迷惑,只有张爱玲对现实的认识是清醒的.其实可以这样理解,之前我们知道 struct urb 表征的是一个 usb request,而这里 struct usb_sg_request 实际上表示的是一个 scatter gather request,从我们非 usb 核心层的人来看,这两个结构体的用法是一样的.对于每次 urb 请求,我们所作的只是申请一个结构体变量或者说申请指针然后申请内存,第二步就是提交 urb,即调用 usb_submit_urb(),剩下的事情 usb core 就会去帮我们处理了,Linux 中的模块机制酷就酷在这里,每个模块都给别人服务,也同时享受着别人提供的服务.就像我们常说的,你站在桥上看风景,看风景的人在楼上看你.明月装饰了你的窗子,你装饰了别人的梦.你要想跟别人协同工作,你只要按照人家提供的函数去调用,把你的指针你的变量传递给别人,其它的你根本不用管,事成之后你人家自然会通知你.同样对于 sg request,usb core 也实现了这些,我们只需要申请并初始化一个 struct usb_sg_request 的结构体,然后提交,然后 usb core 那边自然就知道该怎么处理了.闲话少说,先来看 struct usb_sg_request 结构体.她来自 include/linux/usb.h:

```
988 /**
989  * struct usb_sg_request - support for scatter/gather I/O
990  * @status: zero indicates success, else negative errno
```

```
 991  * @bytes: counts bytes transferred.
 992  *
 993  * These requests are initialized using usb_sg_init(), and then are used
 994  * as request handles passed to usb_sg_wait() or usb_sg_cancel().  Most
 995  * members of the request object aren't for driver access.
 996  *
 997  * The status and bytecount values are valid only after usb_sg_wait()
 998  * returns.  If the status is zero, then the bytecount matches the total
 999  * from the request.
1000  *
1001  * After an error completion, drivers may need to clear a halt condition
1002  * on the endpoint.
1003  */
1004 struct usb_sg_request {
1005         int                     status;
1006         size_t                  bytes;
1007
1008         /*
1009          * members below are private to usbcore,
1010          * and are not provided for driver access!
1011          */
1012         spinlock_t              lock;
1013
1014         struct usb_device       *dev;
1015         int                     pipe;
1016         struct scatterlist      *sg;
1017         int                     nents;
1018
1019         int                     entries;
1020         struct urb              **urbs;
1021
1022         int                     count;
1023         struct completion       complete;
1024 };
```

整个 usb 系统都会使用这个数据结构,如果我们希望使用 scatter gather 方式的话.usb core 已经为我们准备好了数据结构和相应的函数,我们只需要调用即可.一共有三个函数,她们是 usb_sg_init,usb_sg_wait,usb_sg_cancel.我们要提交一个 sg 请求,需要做的是,先用 usb_sg_init 来初始化请求,然后 usb_sg_wait()正式提交,然后我们该做的就都做了.如果想撤销一个 sg 请求,那么调用 usb_sg_cancel 即可.

咱们虽说不用仔细去看着三个函数内部是如何实现的,但至少得知道该传递什么参数吧.不妨来仔细看一下 usb_sg_init()被调用时传递给她的参数.头一个刚才已经说了,就是 sg request,第二个,需要告诉她是哪个 usb 设备要发送或接收数据,咱们给她传递的是 us->pusb_dev,第三个,是哪个 pipe,这个没什么好说的,pipe 是上面一路传下来的.第四个参数,这是专门适用于中断传输的,被传输中断端点的轮询率,对于

bulk 传输,直接忽略,所以咱们传递了 0.第五个和第六个参数就分别是 sg 数组和 sg 数组中元素的个数.然后第七个参数,length,传递的就是咱们希望传输的数据长度,最后一个是 SLAB flag,内存申请相关的一个 flag.如果驱动程序处于 block I/O 路径中应该使用 GFP_NOIO,咱们这里 SLAB_NOIO 实际上是一个宏,实际上就是 GFP_NOIO.不要问我为什么用 SLAB_NOIO 或者说 GFP_NOIO,无可奉告.(如果你真的想知道为什么的话,回去看当初我们是如何调用 usb_submit_urb()的,理由当时就已经讲过了.)这个函数成功返回值为 0,否则返回负的错误码.初始化好了之后就可以为 us->flags 设置 US_FLIDX_SG_ACTIVE 了,对这个 flag 陌生吗?还是回去看 usb_submit_urb(),当时我们也为 urb 设置了这么一个 flag,US_FLDX_URB_ACTIVE,其实历史总是惊人的相似.当初我们对待 urb 的方式和如今对待 sg request 的方式几乎一样.所以其实是很好理解的.

对比一下当初调用 usb_submit_urb()的代码,就会发现 464 到 471 这一段我们不会陌生,当年咱们提交 urb 之前就有这么一段,usb_stor_msg_common()函数中,只不过那时候是 urb 而不是 sg,这两段代码之间何其的相似!只是年年岁岁花相似,岁岁年年人不同啊!然后 474 行,usb_sg_wait()函数得到调用.她所需要的参数就是 sg request 的地址,咱们传递了 us->current_sg 的地址给她.这个函数结束,US_FLIDX_SG_ACTIVE 这个 flag 就可以 clear 掉了.返回值被保存在 us->current_sg.status 中,然后把她赋给了 result.而 us->current_sg.bytes 保存了实际传输的长度,把她赋给*act_len,然后返回之前,once more,调用 interpret_urb_result()转换一下结果.

最后,usb_stor_bulk_transfer_sg()函数返回之前还做了一件事,将剩下的长度赋值给了 *residual.*residual 是形参,实参是&srb->resid.而最终 usb_stor_bulk_transfer_sg()返回的值就是 interpret_urb_result()翻译过来的值.但是需要明白的一点是,这个函数的返回就意味着 Bulk 传输中的关键阶段,即数据阶段的结束.剩下一个阶段就是状态阶段了,要传递的是 CSW,就像当初传递 CBW 一样.

回到 usb_stor_Bulk_transport()函数中来,判断结果是否为 USB_STOR_XFER_ERROR 或者 USB_STOR_XFER_LONG,前者表示出错,这没啥好说的.而后者表示设备试图发送的数据比咱们需要的数据要多,这种情况咱们使用一个 fake sense data 来向上层汇报,出错了,但是和一般的出错不一样的是,告诉上层,这个命令别再重发了.fake_sense 刚开始初始化为 0,这里设置为 1,后面将会用到.到时候再看.目前只需要知道的是,这种情况并不是不存在,实际上 usb mass storage bulk-only spec 里边就定义了这种情况,spec 说了对这种情况,下一个阶段还是要照样进行.至于设备干嘛要这样做,那就只有天知道了,就是说你明明只是对他说,"给我十块钱",他却硬塞给你一百块钱.(我只是打个比方,别做梦了.)文雅一点说,这叫,原想采撷一枚红叶,你却给了我整个的枫林.

最后,解释一点,USB_STOR_XFER_LONG 只是我们自己定义的一个宏,实际上是由 interpret_urb_result()翻译过来的,真正的从 usb core 那一层传递过来的结果是叫做-EOVERFLOW,这一点在 interpret_urb_result 函数中能找到对应关系.-EOVERFLOW 我们就常见了,顾名思义,就是溢出.

最后的最后,再解释一点,实际上 usb core 这一层做的最人性化的一点就是对 urb 和对 sg 的处理了.写代码的人喜欢把数据传输具体化为 request,urb 和 sg 都被化作 request,即请求.而 usb core 的能耐就是让你写设备驱动的人能够只要申请一个请求,调用 usb core 提供的函数进行初始化,然后调用 usb core 提供的函数进行提交,这些步骤都是固定的,完全就像使用傻瓜照相机一样,然后进程可以睡眠,或者可以干别的事情,完事之后 usb core 会通知你.然后你就可以接下来干别的事情了.我做一个比方,就好比你考四六级,找了一个枪手,让他去给你考,你只要告诉他你的基本信息,把你的准考证给他,然后你就不用管别的什么了,剩下的事情他会去处理,然后你也不用担心完事之后他不会通知你,这简直是不容置疑的,因为你还没给钱呢.明白了不,小朋友?

迷雾重重的 Bulk 传输(六)

接下来咱们该看看如何处理 CSW 了.1018 行,usb_stor_bulk_transfer_buf()函数再一次被调用,这次是获得 CSW,期望长度是 US_BULK_CS_WRAP_LEN,这个宏来自 drivers/usb/storage/transport.h 中:

```
109 #define US_BULK_CS_WRAP_LEN     13
```

13 对应 CSW 的长度,13 个 bytes.而 cswlen 记录了实际传输的长度.1025 行,如果返回值是 USB_STOR_XFER_SHORT,表明数据传少了,没有达到我们期望的那么多,而假如 cswlen 又等于 0,那么说明没有获得真正的 CSW,正如注释所说,有些变态的设备会在数据阶段末尾多加一些 0 长度的包进来,这就意味着咱们并没有获得 CSW,于是重新执行一次 usb_stor_bulk_transfer_buf(),再获得一次.(旁白:数据传输失败了可以重来一次,你我失去的青春能重来一次么?唉,人生没有彩排,天天都是现场直播.)

1032 行,如果 result 等于 USB_STOR_XFER_STALLED,在 interpret_urb_result 中查找一下,USB_STOR_XFER_STALLED 对应于 usb core 传回来的是-EPIPE,这种情况说明管道不通,就相当于您家里的下水管道堵塞,当然这也说明 get CSW 再次失败了...,这种情况很简单,直接 retry,为什么要 retry?我们看一下 interpret_urb_result()函数,最重要的就是 310 到 314 行,这里判断了,因为我们曾经讲过,bulk 端点可能会设置了 halt 条件,设置了这种条件的端点必然会堵塞管道,所以这里就不管如何,试一试看,看清掉这个 flag 是否会有好转.所以对于这种情况,我们可以重试一次.我们抱着试一试的心态去 retry,应该说这种心态是正确的,我再重申一次,这里实际上反映的就是 Linux 代码背后的哲学,反映的是一种勇敢面对挫折的人生态度,一枚贝壳要用一生的时间才能将无数的沙粒转化成一粒并不规则的珍珠,雨后的彩虹绽放刹那的美丽却要积聚无数的水汽.如果把这些都看成是一次又一次挫折,那么是挫折成就了光彩夺目的珍珠和美丽的彩虹.我们要相信,失败并不可怕,失败是通往成功的道路.

如果您不是像李白一样近视的话,(床前明月光都能看成地上霜的人,还不是近视吗?)您应该会看见这次传递给 usb_stor_bulk_transfer_buf()函数的最后一个参数不是像之前那样,这次是 NULL,这是因为实际上 cswlen 作为一个临时变量,表征的是状态阶段的实际传输长度,但是在眼下这种情况我们已经不需要使用这个临时变量了.

1042 行,好家伙,如果都这么重新获取了还不成功的话,不用再瞎耽误工夫了,直接返回吧,向领导汇报这设备无药可救了.没办法,返回 USB_STOR_TRANSPORT_ERROR,到这里还不成功那真的就是让人绝望了.什么?你说失败是成功之母?对于这句话我没有异议,问题是失败在遇上我之前已经结扎了...

而从 1046 行开始,正式分析 CSW 了.结合我们从 usb mass storage bulk only 协议中抓出来的那幅图,那幅介绍 CSW 的格式的图,bcs->Residue 对应于 CSW 的 dCSWDataResidue,她表示的是实际传输的数据和期望传输的数据的差值.bcs->Signature 对应于 CSW 中的 dCSWSignature,bcs->Tag 对应于 CSW 中的 dCSWTag,而 bcs->Status 对应于 bCSWStatus.我们有些事情没道理的,有人很抢手,有人没资格,路是人走的.在 bcs 中成员的存储格式居然还有区别,有的是 little endian 的,有些却不是.对于那些 little endian 的,咱们需要调用像 cpu_to_le32 这样的宏来转换,而其她的却不需要转换,对于 bcs 来说,其成员 Residue 和 Signature 就需要这样转换.这些规矩仿佛是没有道理的.

1050 行,和之前 bcb 中使用 US_BULK_CB_SIGN 一样,US_BULK_CS_SIGN 这个宏用来标志这个数据包是一个 CSW 包.而 US_BULK_CS_OLYMPUS_SIGN 也是一个宏,不过她是专为某种变态设备专门准备的.这两个宏和接下来将提到的一些宏依然来自 drivers/usb/storage/transport.h 中,

```
110 #define US_BULK_CS_SIGN         0x53425355      /* spells out 'USBS' */
111 /* This is for Olympus Camedia digital cameras */
112 #define US_BULK_CS_OLYMPUS_SIGN         0x55425355      /* spells out
'USBU' */
113 #define US_BULK_STAT_OK         0
114 #define US_BULK_STAT_FAIL       1
115 #define US_BULK_STAT_PHASE      2
```

对大多数普通的设备来说,如果要标志一个 CSW 包,其 Signature 会是 53425355h.但是 Olympus Camedia 这种数码相机偏偏要标新立异,她愣是跟您换个数字,那咱也没办法.敢情人家设计者是穿美特斯邦威长大的,就是不走寻常路.

Tag 就是和 CBW 相对应的,两个 Tag 应该相同.要不然也就不叫接头暗号了.前面为 Tag 赋值为 srb->serial_number,这回自然也应该等于这个值.

而 bcs->Status,标志命令执行是成功还是失败,当她是 0 表明命令是成功的,当她是非 0,嘿嘿,肯定有问题.目前的 spec 规定,她只能是 00h,01h,02h,而 03h 到 FFh 都是保留的,不能用,所以这里会判断她是否是大于 US_BULK_STAT_PHASE,也就是说是否会大于 02h,大于了当然就不行.好,这样子,就是说这些条件如果不满足的话,那么一定是有问题的.返回错误值吧.

1060 行至 1066 行,如果 residue 不为 0,那么说明数据没传完,或者说和预期的不一样,那么来细看一下,首先该设备应该没有设置 US_FL_IGNORE_RESIDUE 这个 flag,老规矩,让我们看一下什么样的设备设置了这个 flag,

```
269 /* Yakumo Mega Image 37
270  * Submitted by Stephan Fuhrmann <atomenergie@t-online.de> */
271 UNUSUAL_DEV(  0x052b, 0x1801, 0x0100, 0x0100,
272                 "Tekom Technologies, Inc",
273                 "300_CAMERA",
274                 US_SC_DEVICE, US_PR_DEVICE, NULL,
275                 US_FL_IGNORE_RESIDUE ),
276
277 /* Another Yakumo camera.
278  * Reported by Michele Alzetta <michele.alzetta@aliceposta.it> */
279 UNUSUAL_DEV(  0x052b, 0x1804, 0x0100, 0x0100,
280                 "Tekom Technologies, Inc",
281                 "300_CAMERA",
282                 US_SC_DEVICE, US_PR_DEVICE, NULL,
283                 US_FL_IGNORE_RESIDUE ),
284
285 /* Reported by Iacopo Spalletti <avvisi@spalletti.it> */
```

```
286 UNUSUAL_DEV(  0x052b, 0x1807, 0x0100, 0x0100,
287                 "Tekom Technologies, Inc",
288                 "300_CAMERA",
289                 US_SC_DEVICE, US_PR_DEVICE, NULL,
290                 US_FL_IGNORE_RESIDUE ),
291
292 /* Yakumo Mega Image 47
293  * Reported by Bjoern Paetzel <kolrabi@kolrabi.de> */
294 UNUSUAL_DEV(  0x052b, 0x1905, 0x0100, 0x0100,
295                 "Tekom Technologies, Inc",
296                 "400_CAMERA",
297                 US_SC_DEVICE, US_PR_DEVICE, NULL,
298                 US_FL_IGNORE_RESIDUE ),
299
300 /* Reported by Paul Ortyl <ortylp@3miasto.net>
301  * Note that it's similar to the device above, only different prodID */
302 UNUSUAL_DEV(  0x052b, 0x1911, 0x0100, 0x0100,
303                 "Tekom Technologies, Inc",
304                 "400_CAMERA",
305                 US_SC_DEVICE, US_PR_DEVICE, NULL,
306                 US_FL_IGNORE_RESIDUE ),
```

一般的设备是不会设置这个 flag 的,但是确实有那么一些设备是设了这个 flag 的,查一查 drivers/usb/storage/unusual_devs.h,发现 Tekom 公司的数码相机全都有这么一个问题.这个 flag 的意思很明确,对于这类设备不需要管在乎 CSW 中的那个 dCSWDataResidue,因为十有八九这个字节汇报的东西是错的,是不准的,当然这也就是有一个硬件 bug 的例子了.所以这里判断的就是这个 flag 没有设,或者 srb->sc_data_direction 等于 DMA_TO_DEVICE,这种情况下,发送给设备的数据长度不应该超过 transfer_length.而 1064 行 srb->resid 本来是我们传递给 usb_stor_bulk_transfer_sg 的参数,记录的就是剩下的数据长度,(比如期待值是 10,传递了 8,那么剩余就是 2.白痴都知道.),而 residue 刚刚被再次赋值了,不是原来的 residue 就是 transfer_length,要知道原来的 residue 等于 dCSWDataResidue,这是设备传递过来的,换言之,是硬件传来的数据,它未必就和我们软件得来的相同.所以 srb->resid 这时候就等于 residue 了,以硬件的为准呗.不过我想说的是,这几行代码实际上涉及到一个鲜为人知的花絮,如今这个世界每天都有人爆料,每天都有人炮轰别人,以至于我们常说生活就像宋祖德的嘴,你永远都不知道下一个倒霉的会是谁.不过这里我只爆料不炮轰.首先你看这段代码的时候,一定不明白为什么要判断传输方向是不是 DMA_TO_DEVICE 对吧,其实这里又是一个硬件的 bug,开发设备驱动这东西,最讲究的就是实战,尤其是像 usb-storage 这么一个通用的模块,它要支持各种各样的设备,不管你是三星家的还是索尼家的,只要你生产的是 usb mass storage 设备,而且你又不准备自己专门写一个设备驱动,那么我们这个 usb-storage 就应该支持你这个设备.而,在实战中,我们发现,有些设备,他们在执行读操作的时候,经常会在状态阶段汇报一个错误的 dCSWDataResidue,就是说比如本来没有传输完全顺利,该传几个字节就传了几个字节,按理说这种情况,dCSWDataResidue 应该是记录着 0,可是实际上这些设备却把这个值设成了某个正数,你说这不是胡来吗?而如果我们发现这个值是正数,那么当我们向 scsi 核心层反映我们我们这个命令的结果的时候就会说这个命令执行失败了.但事实上这些设备执行读操作并没有问题,这种情况属于谎报军情,罪该论斩.所以我们这里就加入这么一个标志,对于读操作,即方向为 DMA_FROM_DEVICE 的情况,咱们就忽略这个 dCSWDataResidue,换言之,也就是忽略 residue,我们直接返回给 scsi 那边

srb->resid 就可以了.反之,对于写操作,即方向为 DMA_TO_DEVICE 的操作,我们当然不愿意无缘无故的抛弃有效的 dCSWDataResidue 啦.所以对于 DMA_TO_DEVICE 的情况,我们最终返回给 scsi 核心层的 srb->resid 是以 srb->resid 和 residue 中那个大一点的为准.这就是为什么我们这里要判断或者我们设置了 US_FL_IGNORE_RESIDUE 这么一个 flag,或者我们执行的是读操作,对于这两种情况我们要忽略掉 residue,反之我们就不忽略.不过有趣的是,三年前,某个老外去开源社区抱怨,说他买了一个中国厂商生产的 MP3,该设备在写操作的时候老是莫名其妙的报错,但是在 Windows 下却用得好好的.最后大家一分析,发现问题就是在这里,即这个设备在写的时候会误报 dCSWDataResidue,用社区里面那些伙计的话说就是,这种设备在执行写命令的时候,会往 dCSWDataResidue 填写垃圾信息.本来写操作是正确的执行了,可是偏偏要让 scsi 那边以为操作没有执行成功.所以,某位帅哥就提交了一个 patch,把这个判断方向的代码去掉了,因为反正读写都有可能出问题,那么干脆甭判断了,都给忽略掉得了,也因此,对于这种有问题的设备,就必须设置 US_FL_IGNORE_RESIDUE 这个 flag 了.当时那个 patch 是这样的:

```
===== drivers/usb/storage/transport.c 1.151 vs edited =====
--- 1.151/drivers/usb/storage/transport.c          2004-10-20 12:38:15 -04:00
+++ edited/drivers/usb/storage/transport.c         2004-10-28 10:50:42 -04:00
@@ -1058,8 +1058,7 @@
          /* try to compute the actual residue, based on how much data
           * was really transferred and what the device tells us */
          if (residue) {
-                 if (!(us->flags & US_FL_IGNORE_RESIDUE) ||
-                                 srb->sc_data_direction == DMA_TO_DEVICE) {
+                 if (!(us->flags & US_FL_IGNORE_RESIDUE)) {
                          residue = min(residue, transfer_length);
                          srb->resid = max(srb->resid, (int) residue);
                  }
```

同时我们也把当时去开源社区抱怨的那位哥们的调试信息贴出来:

```
usb-storage: Command WRITE_10 (10 bytes)
usb-storage:   2a 00 00 00 01 37 00 00 08 00
usb-storage: Bulk Command S 0x43425355 T 0x82 L 4096 F 0 Trg 0 LUN 0 CL 10
usb-storage: usb_stor_bulk_transfer_buf: xfer 31 bytes
usb-storage: Status code 0; transferred 31/31
usb-storage: -- transfer complete
usb-storage: Bulk command transfer result=0
usb-storage: usb_stor_bulk_transfer_sglist: xfer 4096 bytes, 2 entries
usb-storage: Status code 0; transferred 4096/4096
usb-storage: -- transfer complete
usb-storage: Bulk data transfer result 0x0
usb-storage: Attempting to get CSW...
usb-storage: usb_stor_bulk_transfer_buf: xfer 13 bytes
usb-storage: Status code 0; transferred 13/13
usb-storage: -- transfer complete
usb-storage: Bulk status result = 0
usb-storage: Bulk Status S 0x53425355 T 0x82 R 3072 Stat 0x0
```

usb-storage: -- unexpectedly short transfer
usb-storage: scsi cmd done, result=0x10070000
SCSI error : <0 0 0 0> return code = 0x10070000
end_request: I/O error, dev sda, sector 311

应该说这段信息清晰的打印出来整个Bulk传输是怎么进行的.一共三个阶段,Command/Data/Status,这里执行的命令就是 WRITE_10,本来这是一次成功的传输,但是最后返回值 result 却不为 0,而是 0x10070000,关于这个 0x10070000 如何出来的,我们稍候会知道.最后两行是 scsi core 那边的代码打印出来的,我们不用管,只是需要知道我们最终返回给 scsi 核心层的一个有用信息就是 srb->result.所以我们看到 scsi 那边打印了一个 reture code,和我们这里的 result 是一样的.其实打印的都是 srb->result.很显然,srb 这个东西相当于 usb-storage 和 scsi 那边的桥梁,连接了两个模块.

Ok,继续往下走.1068 行开始基于 CSW 返回的状态,来判断结果了.判断的值就是 bcs->Status.如果是 0,那么表明命令执行成功了.关于 bcs->Status 的取值,usb mass storage bulk only spec 里面规定的很清楚,参考下面这幅图

Table 5.3 - Command Block Status Values

Value	Description
00h	Command Passed ("good status")
01h	Command Failed
02h	Phase Error
03h and 04h	Reserved (Obsolete)
05h to FFh	Reserved

我们前面看到,我们定义了三个宏 US_BULK_STAT_OK / US_BULK_STAT_FAIL / US_BULK_STAT_PHASE ,分别对应 00h,01h,02h.所以这里我们就用这三个宏来进行判断.先来看后两个,如果是 US_BULK_STAT_FAIL,那么返回 USB_STOR_TRANSPORT_FAILED,如果是 US_BULK_STAT_PHASE,那么返回 USB_STOR_TRANSPORT_ERROR.这俩没啥好说的,我们在 drivers/usb/storage/transport.h 中一共定义了四个这样的宏:

```
131 /*
132  * Transport return codes
133  */
134
135 #define USB_STOR_TRANSPORT_GOOD     0   /* Transport good, command good      */
136 #define USB_STOR_TRANSPORT_FAILED   1   /* Transport good, command failed    */
137 #define USB_STOR_TRANSPORT_NO_SENSE 2   /* Command failed, no auto-sense     */
138 #define USB_STOR_TRANSPORT_ERROR    3   /* Transport bad (i.e. device dead) */
```

其意思都很明显,从字面上就能看出来.相信受过九年制义务教育的你不会看不明白.至于这里返回这些值以后上面如何处理那我们稍后从这个函数返回了就知道了.另一个问题,FAILED 和 ERROR 的区别也很明显,一个是说传输没问题,但是命令执行的时候有错误,另一个是传输本身就有错.

现在我们来看看 US_BULK_STAT_OK 的情况了,这种情况说明,传输成功了.但是这里需要判断 fake_sense.什么是 fake_sense?我们下一节再来专门讨论这个问题,需要知道的是,这里这三个 return 就意味着 usb_stor_Bulk_transport()函数将结束了(当然,还有一个 return,1094 行,就是说如果以上情况都不属于,那当然更加是出错了,所以直接返回 USB_STOR_TRANSPORT_ERROR).我们将返回 usb_stor_invoke_transport(),而这更加意味着一次 Bulk 传输的结束.至此我们就算是把这个迷雾重重的 Bulk 传输给从头到尾讲了一遍.而回到 usb_stor_invoke_transport()之后所需要做的就是一些错误处理了,或者说秋后算账.

跟着感觉走(一)

接下来的时间里我们会接触两个变量,fake_sense 和 need_auto_sense,sense 顾名思义,感觉.所以就让我们跟着感觉走.我们前面提到过,如果设备想发送比期望值更多的数据,那么我们前面就设了 fake_sense 为 1.这里就来看看设为 1 之后怎么办.这里咱们看到了这个一个冬冬,usb_stor_sense_invalidCDB,她是谁?

让我们把镜头对准 drivers/usb/storage/scsiglue.c,

```
479 /* To Report "Illegal Request: Invalid Field in CDB */
480 unsigned char usb_stor_sense_invalidCDB[18] = {
481         [0]     = 0x70,                    /* current error */
482         [2]     = ILLEGAL_REQUEST,           /* Illegal Request = 0x05 */
483         [7]     = 0x0a,                    /* additional length */
484         [12]    = 0x24                      /* Invalid Field in CDB */
485 };
486
```

这是一个字符数组,共 18 个元素,初始化的时候其中 4 个元素被赋了值,为了说明这个数组,下面不得不插播一段 scsi 广告,广告过后立刻回来.

我们知道 SCSI 通过命令通信,有一个命令是 Request Sense.她是用来获取错误信息的,不知道为什么,那些有文化的人把错误信息唤作 sense data.可能老外取名字都喜欢取得很优雅吧,相比之下,我们国内很多东西取名字就有些土,比如某所高校,中文名是沈阳理工,而英文名居然就是 Shenyang Ligong University.这样没文化的名字实在让人笑死了.如果一个设备接收到了一个 Request Sense 命令,那么她将按游戏规则返回一个 sense data,我们可以参考 scsi 协议,找到 sense data 的格式规定,如下图所示:

Table 65 - Error codes 70h and 71h sense data format

Bit / Byte	7	6	5	4	3	2	1	0
0	Valid	Error code (70h or 71h)						
1	Segment number							
2	Filemark	EOM	ILI	Reserved	Sense key			
3	(MSB)	Information						
6								(LSB)
7	Additional sense length (n-7)							
8	(MSB)	Command-specific information						
11								(LSB)
12	Additional sense code							
13	Additional sense code qualifier							
14	Field replaceable unit code							
15	SKSV	Sense-key specific						
17								
18	Additional sense bytes							
n								

标准的 sense data 是 18 个 bytes 的.所以这里准备了一个 18 个元素的数组,第 0 个 byte 的低七位称为 error code,0x70 表明是出问题的是当前这个命令,第二个 byte 的低四位成为 sense key,0x5h 称为 Illegal Request,表明命令本身有问题,比如命令的参数不合法.而第七个 byte 称为 additional sense length 表明在这个 18 个元素之后还会有 additional sense bytes,而她的长度就在这里被标注了,这些 additional sense bytes 通常指的是一些命令特有的数据,或者是一些外围设备特有的数据,这里为她赋值为 0x0a.而第十二个 byte,称为 additional sense code,这部分针对 sense key 提供一些信息,也就是说比如 sense key 如果是 Illegal Request,那么我们知道了是命令有问题,那么究竟有什么问题呢?additional sense code 提供更详细的一些信息,scsi 规范中对 24h 的描述是 Invalid Field in CDB,正是我们这里注释所说.

所以,这样我们明白了,1073 行,就是将 usb_stor_sense_invalidCDB 数组里边的冬冬 copy 至 srb->sense_buffer 里边,然后返回 USB_STOR_TRANSPORT_NO_SENSE.struct scsi_cmnd 结构体里面是这样定义 sense_buffer 的,

```
115 #define SCSI_SENSE_BUFFERSIZE    96
116         unsigned char sense_buffer[SCSI_SENSE_BUFFERSIZE];              /*
obtained by REQUEST SENSE
117                                                 * when CHECK CONDITION is
118                                                 * received on original command
119                                                 * (auto-sense) */
```

关于 sense_buffer 就得从 scsi 协议以及 Linux 中的 scsi 核心层来讲了.scsi 协议里边有这么一码子事,当一个 scsi 命令执行出了错,你可以发送一个 REQUEST_SENSE 命令给目标设备,然后它会返回给你一些信息,即 sense data.不过呢,scsi 核心层偷懒,把这一艰巨的任务抛给了底层驱动,即我们作为底层驱动不得不自己发送 REQUEST SENSE 命令给目标设备.当然了,所谓的 scsi core 偷懒并不是没有它的道理,因为有些 scsi host 卡会自动发送这个命令,就是说当设备汇报说命令执行有误,那这时 scsi host 卡会自动

发送 REQUEST SENSE 命令去了解详情.所以 scsi core 就干脆把权力下放,让底层驱动自己去处理吧.因此稍后我们会看到一个变量名字叫做 need_auto_sense.就是说,REQUEST SENSE 这个命令要么就是硬件你自动发出去,要么就让软件自动发出去,总之 scsi core 这一层是不管你了.只要你最终返回 scsi 核心层的时候把相关的 sense data 保存在 srb->sense_buffer 里,scsi 核心层自然就知道该如何处理了.

再回到我们具体的问题中来,我们说了,有些设备就是贱,你明明只期望它返回 n 个字节,它偏偏要给你捣乱,它想返回 n+m 个字节,对于这种情况我们怎么处理?老实说,它想多返回的几个字节我们完全可以抛弃,因为我们只关心我们提供的 buffer 是否装满了,是否达到了我们要求的 length 个字节,如果达到了,那么剩下的不管也罢,不过写代码的同志们在这个问题上考虑得比我们要周到,他们对这个细节也是体贴入微的.或者说他们对这种傻 X 的设备也是很关心的.对于这种情况,写代码的同志们考虑,还是应该向上层汇报一下,说明这个命令对这个设备来说,执行起来总有些问题.因为这种情况完全可能就是,比如说,一个命令可以带有一些参数,而可能某个设备并不支持其中的某个参数,而你执行命令的时候去设了这个参数,那么设备的返回值可能就不正常,所谓的比预期的值要多就是一种不正常的表现,所以呢,对于这种情况,我们干脆就告诉上层这个命令有问题,在 Linux 中,我们就可以通过 sense data 来向上层汇报.而之所以这里称作 fake sense,说的是这个 sense data 里面的东西是我们自己设置好的,因为我们已经很清楚我们应该在设备里放置什么,不需要向设备发送一个 REQUEST SENSE 的命令,而更确切的说,我们这个命令的返回结果是 US_BULK_STAT_OK,也就是说,从设备那方返回的状态来看,设备认为命令没有问题,但是你说设备的话你能相信吗?老实说,从孙志刚案件开始,我不相信警察!从刘涌案件开始,我不相信法律!从苏秀文案件开始,我不相信政府!从打工受户籍歧视开始,我不相信有公平!从 CCTV 每天报喜不报忧开始,我不相信有真话!而从我进入复旦微电子系开始学习计算机硬件,我他妈的就没相信过硬件设备.不是这有毛病就是那有毛病,硬件 bug 到处都是.算了,别跑题了,继续说,因为设备认为传输是成功的,所以你发送 REQUEST SENSE 根本就没用,因为设备根本就不会为你准备 sense data,因为 sense data 本来就是为了提供错误信息的.因此我们需要自己设一个 sense data,放进 sense_buffer 里去.从而让 scsi core 那一层知道有这么一回事,别被设备瞒天过海给忽悠了.

讲到这里,usb_stor_Bulk_transport()这个函数就算结束了.返回值一共就是四种情况,USB_STOR_TRANSPORT_GOOD,USB_STOR_TRANSPORT_FAILED,USB_STOR_TRANSPORT_ERROR,以及 USB_STOR_TRANSPORT_NO_SENSE.然后上层会去分析这些返回值.让我们结束这个函数,回到调用她的函数中来,即 usb_stor_invoke_transport().在回去之前,我们需要记住的就是,对于刚才说的这种情况,即 fake_sense 为 1 的情况,我们返回的就是 USB_STOR_TRANSPORT_NO_SENSE.一会我们会看到 usb_stor_invoke_transport()中是如何应付这种情况的.

跟着感觉走(二)

回到 usb_stor_invoke_transport()中来,540 行,还是老套路,又问是不是命令被放弃了,放弃了当然下面的就别执行了.goto Handle_Abort 去.

546 行,如果有错误,注意正如前面所说,USB_STOR_TRANSPORT_ERROR 表示传输本身就是有问题的,比如管道堵塞.而 USB_STOR_TRANSPORT_FAILED 则只是说明命令传输是没有问题的,就比如你作为场外观众给非常六加一发短信了,然后李咏随机抽到你,给你打电话,电话通了,让你砸金蛋,砸出了金花你就能获得自己想要的奖品,但是问题是你没有砸中,这样你就失去了机会,这属于 FAILED,但是另一种更惨的情况是,咏哥给你打电话还赶上你小子把手机关了,那你就只好认倒霉了,这就属于 ERROR.对于这种疑似管道堵塞的问题,我们会调用自己写的一个函数 us->transport_reset(us),us->transport_reset()其

实也是一个指针,我们也是很早以前和 us->transport()一起赋的值,对于 U 盘,我们赋的值是 usb_stor_Bulk_reset().所谓 reset,就相当于我们重起计算机,每次遇到些什么乱七八糟的问题,我们二话不说,重起机器通常就会发现一切都好了.关于设备 reset 的冬冬,我们讲完命令的执行这一块之后再来专门讲.暂且不表.对于这种情况,当然我们会设置 srb->result 为 DID_ERROR.然后返回.

554 行,看到了吧,这里就判断是不是 USB_STOR_TRANSPORT_NO_SENSE 了.如果是的,那么返回给 scsi 的结果是 SAM_STAT_CHECK_CONDITION.返回这个值,scsi 核心层那边就知道会去读 srb->sense_buffer 里边的东西.SAM_STAT_CHECK_CONDITION 是 scsi 那边定义的宏,scsi 协议规定,scsi 总线上有若干个阶段,比如命令阶段,比如数据阶段,比如状态阶段,这三个阶段其实咱们 Bulk-Only spec 里边也有.不过 scsi 协议里还规定了更多的一些阶段,在 scsi 协议里边称一个阶段为一个 phase.除了这三个 phase 以外,还可以有 bus free phase,有 selection phase,有 message phase 等等.而状态阶段就是要求目标设备返回给主机一个状态码(status code).关于这些状态码,在 scsi 的规范里边定义得很清楚.在 include/scsi/scsi.h 中也有相关的宏定义.

```
117 /*
118  *  SCSI Architecture Model (SAM) Status codes. Taken from SAM-3 draft
119  *  T10/1561-D Revision 4 Draft dated 7th November 2002.
120  */
121 #define SAM_STAT_GOOD            0x00
122 #define SAM_STAT_CHECK_CONDITION 0x02
123 #define SAM_STAT_CONDITION_MET   0x04
124 #define SAM_STAT_BUSY            0x08
125 #define SAM_STAT_INTERMEDIATE    0x10
126 #define SAM_STAT_INTERMEDIATE_CONDITION_MET 0x14
127 #define SAM_STAT_RESERVATION_CONFLICT 0x18
128 #define SAM_STAT_COMMAND_TERMINATED 0x22        /* obsolete in SAM-3
*/
129 #define SAM_STAT_TASK_SET_FULL   0x28
130 #define SAM_STAT_ACA_ACTIVE      0x30
131 #define SAM_STAT_TASK_ABORTED    0x40
```

其中,SAM_STAT_CHECK_CONDITION 就是对应 scsi 协议中的 CHECK CONDITION,这一状态表明有 sense data 被放置在相应的 buffer 里,于是 scsi core 那边就会去读 sense buffer.而我们这里遇到这种情况,当然就可以返回了.

559 行,要是没别的啥意外的话,到了这里我们就可以设置 srb->result 为 SAM_STAT_GOOD 了,说明一切都是 ok 的.当然,对于之后会出现的 REQUEST SENSE 的执行失败,我们会再次修改 srb->result 的.

下面 566 行,出现了一个叫做 need_auto_sense 的变量,这是我们定义的临时变量,这里赋初值为 0.574 到 588 行,两个 if 语句,为 need_auot_sense 赋了值,赋值为 1.我们首先很容易理解第二个 if,正如我们前面介绍的那样,REQUEST SENSE 这个艰巨的任务被下放给了我们底层驱动,那么我们就勇敢的去承担它,在 USB_STOR_TRANSPORT_FAILED 的情况下,我们就去发送一个 REQUEST SENSE 命令.设了这个 flag 稍后我们就会看到我们会因此而执行 REQUEST SENSE.那么第一个 if 呢?相信那些注释已经说得很清楚了吧,如果你对自己的智商还有一丁点儿信心的话,应该就不需要我解释了.我只说一句,对于那些遵守 CB 或者 DPCM 协议的设备它们没有自己没有办法决定状态,所以 scsi 核心层当然就不知道去读它

们的 sense buffer 了,但是不读 sense buffer 我们连这个命令执行成功与否都不知道,那怎么行?而设备对于大多数读操作的错误也不需要使用 sense buffer,因为它们对于读操作的错误通常会停止掉 bulk-in pipe,这已经是一个很明显的信号了,不需要再检测 sense buffer,因为检测 sense buffer 或者说检测 sense data 的目的无非就是出错了以后想知道出错的原因,而这种情况下原因已经清楚了嘛不是.至于你问为什么读操作会有这种特点,那我只能说两个字,经验.写设备驱动靠的就是经验.

594 行,srb->resid 大于 0,说明有问题,希望传输 n 个字节,结果汇报上来说只传递了 n-m 个字节.对于这里列出的这五个命令,少传几个字节倒是无所谓,比如 INQUIRY,我就想知道设备的基本信息,那你说你姓甚名谁,生辰八字,学历如何,婚否,等等这些信息,你多说两句就多说两句,少说两句就少说两句,无所谓,没什么影响,但是有些命令就不能少传输了,比如我要传一个 pdf 文档,你传到一半就不传了,那肯定不行,直接导致我可能打不开这个 pdf 文档.驱动程序要是写成这样,那人家不跟你急就怪了.这里咱们就是调用 US_DEBUGP 打印一句调试语句,也就不退出了,咱们忍了.

接下来的一些行,604 行开始,一直到 713 行,我们就是为 need_auto_sense 的情况发送 REQUEST SENSE 命令了.其实和之前一样,我们还是等于再进行一次 Bulk 传输,还是三个阶段,不过我们有了之前的经验,现在再看 Bulk 传输就简单多了,无非就是准备一个 struct scsi_cmnd,调用 us->transport(us->srb,us),然后结束了之后检查结果.这正是这一百多行所做的事情.不过我们偷了点懒,没有另外申请一个 struct scsi_cmnd,而是直接利用之前的那个 srb,只是调用 us->transport 之前先把一些关键的东西备份起来,然后执行完 us->transport 之后再恢复过来.所以接下来我们看到如下事件:

用 old_cmnd 保存了 srb->cmnd;

用 old_cmd_len 保存了 srb->cmd_len;

先把 srb->cmnd 清零,然后对它重新赋值,按 REQUEST SENSE 的意思来赋值.

不同的命令集里边 REQUEST SENSE 的长度也不同,对于 RBC 或者咱们的 SCSI,长度为 6,而对于别的命令集,其长度为 12.

然后用 old_sc_data_direction 保存了 srb->sc_data_direction,而把 srb->sc_data_direction 设置为 REQUEST SENSE 的要求,DMA_FROM_DEVICE,很显然,REQUEST SENSE 是向设备要 sense data,那么当然数据传输方向是从设备到主机.

然后用 old_request_buffer 来保存 srb->request_buffer,而将 srb->request_buffer 设置为 srb->sense_buffer,同时用 old_request_bufflen 来备份 srb->request_bufflen,同时把 srb->request_bufflen 设置为 18.

用 old_sg 来备份 srb->use_sg,而把 srb->use_sg 设置为 0,传这么点数据也就别用那麻烦的 scatter-gather 机制了.

然后用 old_serial_number 来备份 srb->serial_number,并把 srb->serial_number 的最高位取反.

最后用 old_resid 来备份 srb->resid,而把 srb->resid 再次初始化为 0.

这时候就可以调用 us->transport(us->srb,us)了.并且用一个临时变量 temp_result 来保存这个结果.

这次命令完了之后,我们 659 到 666 行,就把刚才备份的那些变量给恢复原来的值.

668 行,再一次判断是不是被放弃了,如果是又 goto Handle_Abort.

672 行,然后判断 temp_result,如果这个 result 说明这次传输还有问题,那就说明连 REQUEST SENSE 都 fail 了.于是我们会设置 srb->result=DID_ERROR<<16.在这之前我们还会调用 us->transport_reset(us)把设备 reset,因为 REQUEST SENSE 都出错本身就说明很不正常,这种情况下我们不得不来点狠的了,男人嘛,就是要对自己狠一点!当然这里有一个条件,我们判断的是 US_FL_SCM_MULT_TARG 这个 flag 没有设置,因为设了这个 flag 的设备有多个 target,这种情况下咱们就不好胡乱给人全 reset 了,因为 REQUEST SENSE 这个命令虽然是一个基本的命令,但是毕竟执行成功与否无所谓,我们只是出于好奇才想知道一个命令执行出错的原因.即使不知道也没有太大的关系.没必要非得大张旗鼓的把一个多个 target 的设备给人家 reset 了,不该管的事情不要管.

然后 685 到 696 行无非就是把 temp_result 的值打印出来,把 sense_buffer 里的值打印出来.这些都是调试信息.对调试设备驱动非常管用.

继续讲之前,我们先看一下,692 行,usb_stor_show_sense().这个函数,是第一次出现,曾经我们见过一个类似的函数,名叫 usb_stor_show_command().她们是老乡,都来自 drivers/usb/storage/debug.c:

```
159 void usb_stor_show_sense(
160                 unsigned char key,
161                 unsigned char asc,
162                 unsigned char ascq) {
163
164         const char *what, *keystr;
165
166         keystr = scsi_sense_key_string(key);
167         what = scsi_extd_sense_format(asc, ascq);
168
169         if (keystr == NULL)
170                 keystr = "(Unknown Key)";
171         if (what == NULL)
172                 what = "(unknown ASC/ASCQ)";
173
174         US_DEBUGP("%s: ", keystr);
175         US_DEBUGPX(what, ascq);
176         US_DEBUGPX("\n");
177 }
```

这里面又调用了其她函数,scsi_sense_key_string 和 scsi_extd_sense_format,这两个函数来自 driver/scsi/constants.c,暂且不表.先来看对 usb_stor_show_sense 这个函数的调用.传递给她的实参是 srb->sense_buffer 中的几个元素,对比咱们前面贴出来的那个 sense data 的格式,可知 sense_buffer[2]的低四位被称为 Sense Key,而 sense_buffer[12]是 Additional sense code,也称 ASC,sense_buffer[13]是 Additional sense code qulifier,也称 ASCQ.这三个冬冬联手为 mid level

提供了需要的信息,主要也就是错误信息或者异常信息.为什么要三个冬冬呢?实际上就是一个分层的描述方法,比如要描述某个房间就要说某城市某街道某门牌号.这三个冬冬也是起着这么一个作用,Sense Key 是第一层,ASC 则是对她的补充,而 ASCQ 则又是对 ASC 的补充,或者说解释.这样我们再来看看 usb_stor_show_sense 就很清楚了,咱们传递进来的是三个 char 变量,而实际的信息就像某种编码一样被融入在了这些 char 变量中,而调用的两个来自 scsi 核心层的函数 scsi_sense_key_string 和 scsi_extd_sense_format 就是起着翻译的作用,也叫解码.解码了就可以打印出来了.Yeah!

699 行,srb->result 设置为 SAM_STAT_CHECK_CONDITION.为什么?不为什么,Request Sense 执行完之后,scsi 规范告诉我们应该把 srb->result 设为 SAM_STAT_CHECK_CONDITION,酱紫 mid level 就知道去检查 sense data.这也是为什么在 554 行,555 行会令 srb->result 也为这个值,只不过那次 sense data 是咱们自己手工准备的,不是通过命令获得的.

704 这个 if 这一小段,首先咱们需要明白,need_auot_sense 这个 flag 被设为 1 实际上是有两种可能的,它本身是在 usb_stor_invoke_transport()中第一行所定义的一个局部变量,并且在这个函数中特意把它初始化为 0. 第一处设置为 1 的位置是 574 行当时 check us->protocol 为 US_PR_CB 或者 US_PR_DPCM_USB,对于这种设备,(如果您只关心 u 盘,那么就甭理这种设备了.)第二处设置这个 flag 的就是我们确实遇到了 failure,585 行,result 如果等于 USB_STOR_TRANSPORT_FAILED,这种情况当然要设置 need_auto_sense 了.而 704 行这里判断 result 是否等于 USB_STOR_TRANSPORT_GOOD,那么很显然,如果 result 等于 USB_STOR_TRANSPORT_FAILED,那么它就不可能等于 USB_STOR_TRANSPORT_GOOD,因此,这里这个判断一定是针对第一种 need_auto_sense 的情况,正如我们曾经说过的,遵守 US_PR_CB/US_PR_DPCM_USB 协议的设备是不会自己返回命令执行之后的 Status,所以我们不管它执行到底成功与否,我们都会对它来一次 REQUEST SENSE,就是为了尽可能多的获取一些信息,这样一旦出了问题,我们至少能多一些辅助信息来帮我们判断问题出在哪.那么对于 USB_STOR_TRANSPORT_GOOD 的情况,首先这说明命令执行是没有问题的了,我们仔细看一下这个 if 语句,除了这个条件以外还判断了另外三个条件,(srb->sense_buffer[2]&0xaf)结果为 0,那么说明 srb->sense_buffer[2]的 bit0~bit3 都为 0,bit5 为 0,bit7 也为 0,而 bit4 和 bit6 是什么我们无所谓.(如果这个你还要问为什么那么我只能说你没救了.没办法,这个世界上只有 10 种人,一种是懂二进制的,一种是不懂二进制的.)虽然我们没有兴趣熟悉每一个 SCSI 命令的细节,但我们毕竟是共产主义接班人,应该对社会主义建设的方方面面都有所了解,所以让我们来仔细看看这个 sense_buffer[2].对照 sense data 的格式那张图,sense data 的第二个字节,bit0~bit3 是 sense key,bit4 是 Reserved,即保留的,不使用的.bit5 是 ILI,全称 incorrect length indicator,bit6 是 EOM,全称 End of Medium,bit7 是 Filemark,伟大的不朽的金山词霸告诉我们这个词叫做卷标.关于 sense key,Scsi 协议是这么规定的,如果 sense key 为 0h,那么这种情况表示 NO SENSE.这种情况通常对应于命令的成功执行或者就是 Filemark/EOM/ILI bits 中的任一个被设置为了 1.需要注意的是,scsi 协议里边定义了四样东西,Filemark/EOM/ILI/Sense Key,它们都是为了提供错误信息的,只是前三者只要一个 bit 就能表达明确的意思了,而最后一个包含很多信息,所以需要用 4 个 bits,并且还在后面附有很多额外信息,即 sense_buffer[12]和 sense_buffer[13],这里也要求它们为 0,即所谓的 ASC 和 ASCQ 都为 0,在 scsi 协议里面,这种情况称之为 NO ADDITIONAL SENSE INFORMATION.关于这一点 scsi 协议是这么说的:"The REQUEST SENSE command requests that the target transfer sense data to the initiator. If the target has no other sense data available to return, it shall return a sense key of NO SENSE and an additional sense code of NO ADDITIONAL SENSE INFORMATION."而这正是我们这里的代码所表达的意思.(什么?你要我翻译这段话?有没有搞错啊,难道你没上过新东方,没听过老罗的课?那么我代表人民代表党义正严辞的告诉你,同志,你真的落伍了耶!)

(filemark 和 eom 都是针对磁带设备的,跟磁盘设备无关.也就是说跟咱们无关.)

最后,满足了这四个条件的情况就表示刚才这次 scsi 命令的传输确实是圆满完成了.应该说这次检测还是蛮严格的,毕竟开源社区的同志们觉得写代码不像我们开会,每次看新闻,发现凡是会议必然是圆满成功的.这里人家检查了这么多条件都满足然后就设置 srb->result 为 SAM_STAT_GOOD,并且把 srb->sense_buffer[0]也置为 0.sense data 的 byte 0 由两部分组成,Valid 和 Error code,如果置为 0,首先就说明这整个 sense data 是无效的, 用 scsi 标准的说法叫 invalid, 所以 scsi core 自然没法识别这么一个 sense data,而我们既然认定这个命令是成功执行的,当然就没有必然让 scsi core 再去理睬这么一个 sense data 了.

以上花了大量笔墨就讲了 704 到 712 这个 if 语句段.需要重新强调一点,正如我们已经说过的,对于 U 盘,这段代码根本就不可能执行,理由我们已经说过了.但是既然它出现在我们眼前了,我们又有什么理由去逃避呢?写代码,尤其是写这种通用的设备驱动程序,必然要考虑各种情况,不是完全跟着感觉走,也不是纯粹的追求华丽的算法和数据结构,更应该接近实际,华丽的代码堆砌的东西缺乏骨质感.

这样,关于 need_auto_sense 设置了的这一段就结束了.最后还想重复一点,说起来,REQUEST SENSE 这种命令应该由 mid level 来发,不应该由底层驱动来发,不过通常 mid-level 并不愿意发这个命令,因为实际上很多 SCSI 主机适配卡(SCSI host adapter)会自动 request the sense.所以为了让事情变得简单,设计上要求底层驱动去对付这个问题.所以要么 SCSi host adapters 自动获得 sense data,要么就是咱们 LLD(底层驱动程序)去发送这个命令,对于咱们这个模拟的 scsi 系统,当然只能是用软件去实现,即咱们必须在 LLD 中用代码来发送 request sense.

再然后,716 行,如果经过了这么一番折腾,srb->result 仍然等于 SAM_STAT_GOOD,(我们在 559 行,即进行 autosense 之前把 srb->result 设置成了 SAM_STAT_GOOD.)那么说明真金不怕火炼,我们再判断最后一个条件,即我们要求传输的数据长度是 srb->request_bufflen,而实际上还剩下 srb->resid 个字节没有被传送,这种情况本身没什么,但是 struct scsi_cmnd 中有一个成员叫做 underflow,其意思是如果传输的数据连这个值都没有达到的话,不管其它条件如何,必须向上层反映,出错了.换句话说,有些 scsi 命令有一个底线,你至少得达到我这个底线,否则我跟你急!所以这里就是判断这么一个条件是否满足,如果传输的长度小于 srb->underflow,那么不用废话,即便你其它条件判断下来都觉得这个命令是成功的,我还是要汇报说你这个命令执行有误.而关于这种情况,我们反馈给 scsi core 的 result 是 DID_ERROR<<16 或上 SUGGEST_RETRY<<24.DID_ERROR 被定义为 0x07,SUGGEST_RETRY 为 0x10.其定义都在 include/scsi/scsi.h 中.所以这里 srb->result 就最终被设置为 0x10070000 了.还记不记得当初我们贴出来的那个关于 US_FL_IGNORE_RESIDUE 关于 MP3 的调试信息了?回过去看一下,没错,当时的 result 就是 0x10070000,也就是这里赋的值.而当时之所以导致执行了这段代码,原因正是设备报虚警,明明读写正常,它偏要瞎写一个 residue 到状态字节里去.导致我们的代码在这里判断出读写出了错.这个疑案只有到了这里我们才能真正明白,哈哈!

至此,我们的这个故事也快接近尾声了.故事总有结束的时候,但 Linux 所反映的那种人们对自由的追求却是永无止境的.天长地久有时尽,此恨绵绵无绝期!--开源社区追求自由的战士白居易.

720 行, usb_stor_invoke_transport()函数终于返回了.如果之前有执行 goto Handle_Abort,那么 724 行会被执行,实际上就是设置 srb->result 为 DID_ABORT<<16,并且执行 reset.关于 reset,您别急我们马上就会讲.

usb_stor_invoke_transport()返回就回到了 usb_stor_transparent_scsi_command(),这个函数实际上不干什么正经事,就是调用 usb_stor_invoke_transport().然后它又判断了一个 flag, US_FL_FIX_CAPACITY,又是对了对付某些硬件 bug.我们可以看到,如果这个 flag 没有设置,那么

usb_stor_transparent_scsi_command()函数就会就此返回了,返回到 usb_stor_control_thread()中,别忘了我们曾经是从 375 行 us->proto_handler(us->srb,us)进入 usb_stor_transparent_scsi_command()的,返回了 375 行之后怎么走我们之前就已经分析过了,usb_stor_control_thread()这个守护进程,它还将永垂不朽的循环下去,它会进入睡眠等待着下一个命令的到来,如果你不强行 kill 它,或者卸载模块,那么它将守护到天长地久,到海枯石烂,到山峰无棱到河水不再流...

然后我们就可以去看US_FL_FIX_CAPACITY,这里又是一个硬件bug,前几年这种硬件bug出现得还不是很多,在 2.6.10 的内核中只有松下等几家公司的产品有这么一个问题,不过随着改革开放的发展,很多事情都变了,猪肉涨价了,方便面涨价了,Apple 公司的 iPod,诺基亚 3250,E70,E60,N91,N80,E61,NIKON 的 DSC D70,DSC D70s,DSC D80,索尼爱立信的 P990i,M600i,摩托罗拉的 RAZR V3x,RAZR V3i,等一大批产品都出现了这个问题.虽然偶自己的 Nokia 6108 没有这种问题,但是考虑到 Apple 的 iPod 最近被评为最伟大的 20 件 IT 产品之一,而 Nokia 作为我们 Intel 多年来最重要的 customer 之一,我觉得这里我们还是有必要来看看这个 bug 的.再说了,了解了这个,将来去 Nokia 面试,去 Apple 面试,或者次一点,去摩托罗拉或者索尼爱立信面试,跟人家谈一下,喂,伙计,我听说你家的产品有这么这么一个 bug,怎么回事啊?兄弟们是不是待遇不佳压力太大啊?人家一看你这么拽,连这种内幕都知道,能不要你么?

光荣属于苹果,属于诺基亚,属于摩托罗拉,属于索尼爱立信!

这一节我们来分析一个在很多企业的产品中都存在的 bug.写设备驱动是一件很实在的事情,你得根据实实在在的硬件来编写你的代码,如果你的硬件存在某种 bug,那么你就要去 fix 它.如果你希望成为通用的驱动程序,那么你就要兼顾各家企业,兼顾各种可能存在的bug.也许一百家企业的产品都可以很好的被你的程序所支持,但是如果地一百零一家的产品有问题,你就得尽量解决.usb-storage 正是这样一个模块.所以它的代码里会涉及到很多不同的企业.当然我相信,有一个更重要的原因,那就是,没有企业的支持,Linux 不可能像今天这样火.所以 Linux 内核代码中支持诸多企业的设备也是必然的,就像厉娜在给把票投给许飞而不是给我们复旦的尚雯婕的时候说的那句:于情于理于公于私,都应该这样.

很多年前,《商业周刊》断言:“自由软件业的开发者大部分水平不高,不可能制造高端的企业级产品”. 这是事实, 出身卑微的 Linux 真正有出人头地的机会, 的确是在各大名企大规模介入之后. 而任何一家企业支持 Linux 的目的都只是为了赚钱.IBM 干嘛支持 Linux?老板思想境界高?想为全人类服务?我告诉你,IBM 自从 2000 年开始展开 Linux 战略,2002 年他们家就从 Linux 市场上赚取了 10 亿美元.所有的公司支持 Linux 的目的无非就是想瓜分那些曾经属于微软的财富. 正如洪波所说的那样,大企业只不过是花钱为这次抢劫置办一件迷人的外衣,让所有人认同这样一个观点,那就是,张君拿着武器抢劫运钞车,是死罪,有钱人拿着 Linux 瓜分微软,是正义. (注:张君,我的老乡,也是我当年的偶像)更滑稽的是,时间长了,每一个学习 Linux 的人都有这样一种感觉,觉得自己正做着一件伟大的正义的,追求自由的大事.(郑重声明一下,我是例外,从未觉得 Linux 跟伟大有什么关系,学习 Linux 只是混口饭而已.)

Ok,下面让我们来仔细看看这段属于苹果,属于诺基亚,属于摩托罗拉,属于索尼爱立信的代码吧.US_FL_FIX_CAPACITY 这个 flag 的设置,意味着这种设备存在这么一个 bug.在 scsi 众多命令中,有一个命令叫做 READ CAPACITY.它的作用很简单,就是读取磁盘的总容量.先来点直观的印象吧,还记得当初哥们给你推荐的那个工具 sg_utils3 么,当初用它执行了 INQUIRY 命令.现在咱们用它执行 READ CAPACITY,具体命令名字叫做 sg_readcap.你要是装了 SUSE Linux 的话,里边的 rpm 包可能不叫 sg_utils,而是叫 scsi-xx-xx,比如偶的就是 scsi-1.7_2.36_1.19_0.17_0.97-12.4.这个包包含很多执行 scsi 命令的工具.而且都有 man 文档,不会用看看 man 就知道了.

先来个硬盘的,比如偶的 scsi 硬盘:

```
myhost: # sg_readcap -b /dev/sda

0x11040000 0x200

myhost: # fdisk -l /dev/sda


Disk /dev/sda: 146.1 GB, 146163105792 bytes

255 heads, 63 sectors/track, 17769 cylinders

Units = cylinders of 16065 * 512 = 8225280 bytes


   Device Boot      Start         End      Blocks   Id  System

/dev/sda1               1         266     2136613+  83  Linux

/dev/sda2             2879       17769   119611957+  83  Linux

/dev/sda3   *          267        1572    10490445   83  Linux

/dev/sda4              1573        2878    10490445   82  Linux swap / Solaris


myhost: #
```

看出 sg_readcap 读出来的信息了吗?我们可以传递不同的参数,如果像我们这里-b 参数,那么将获得 block 数以及每个 block 的字节数.我们来计算一下,我们这里返回的两个值分别是 0x11040000 和 0x200,0x11040000 对应于十进制 285474816,0x200 就是十进制的 512,N 多设备的 block 大小都是 512.这两个相乘就是我们的容量,相乘的结果是 146163105792,看到了吗?和 fdisk 命令显示出来的一模一样,呵呵,其实 fdisk 就是这么干的.不一样就见鬼了.Ok,有了直观的印象可以继续往下看了.刚才我们知道对于有设置 US_FL_FIX_CAPACITY 这个 flag 的设备,就会执行 fix_read_capacity(),这个函数定义于 drivers/usb/storage/protocol.c 中,所谓的这个 bug 在这个注释里说得很清楚.明明容量是 N,偏偏要汇报说自己是 N+1,你说这不是找抽么?当然这也很简单,我们处理起来不难.

```
60 /*
61  * Fix-up the return data from a READ CAPACITY command. My Feiya reader
62  * returns a value that is 1 too large.
63  */
64 static void fix_read_capacity(struct scsi_cmnd *srb)
```

```
65 {
66          unsigned int index, offset;
67          __be32 c;
68          unsigned long capacity;
69
70          /* verify that it's a READ CAPACITY command */
71          if (srb->cmnd[0] != READ_CAPACITY)
72                  return;
73
74          index = offset = 0;
75          if (usb_stor_access_xfer_buf((unsigned char *) &c, 4, srb,
76                          &index, &offset, FROM_XFER_BUF) != 4)
77                  return;
78
79          capacity = be32_to_cpu(c);
80          US_DEBUGP("US: Fixing capacity: from %ld to %ld\n",
81                  capacity+1, capacity);
82          c = cpu_to_be32(capacity - 1);
83
84          index = offset = 0;
85          usb_stor_access_xfer_buf((unsigned char *) &c, 4, srb,
86                          &index, &offset, TO_XFER_BUF);
87 }
```

应该说看过前面我们如果处理 INQRUIY 命令那个 bug 的代码的同志们应该能够很容易看懂眼前这段代码.其中调用的最关键的函数就是 usb_stor_access_xfer_buf(),对 usb_stor_access_xfer_buf()函数陌生的同志们可以回过去看看当时咱们是如何分析的.这里 75 行的作用就是从 request_buffer 里边把数据 copy 到 c 里边去,而 85 行的作用就是反过来把 c 里边的数据 copy 到 request_buffer 里边去.而在这之间,最重要的一步自然是 79 行和 82 行,capacity 被赋为 c,而 c 再被赋为 capacity-1.这样简简单单的几行代码就 fix 了这么一个 bug.

当然,虽然我不喜欢 Linux,但是我还是有必要为你解释一些事情.

你是不是看见好些次这样一系列函数了:
be32_to_cpu/cpu_to_be32(),cpu_to_le16(),cpu_to_le32(),此前我们一直没有讲,所以让我们现在一并来讲吧.反正整个故事的大致走向你已经很清楚了,从现在开始的故事基本上就属于一些小打小闹小修小补式的细枝末节了.le 叫做 Little Endian,be 叫做 Big Endian,这是两种字节序.le 就表示地址地位存储值的低位,地址高位存储值的高位.be 就表示地址低位存储值的高位,地址高位存储值的低位.我们就以这里这个临时变量 c 为例.假设 c 是这样被存储在内存地址 0x0000 开始的地方:

```
0x0000    0x12
0x0001    0x34
0x0002    0xab
0x0003    0xcd
```

如果你是采用 le 的字节序,那么读出来的值就是 0xcdab3412,反之,如果你采用的是 be 的字节序,那么读出来的值就是 0x1234abcd.同样的,如果你把 0x1234abcd 写入 0x0000 开始的内存中,那么结果就是:

	big-endian	little-endian
0x0000	0x12	0xcd
0x0001	0x23	0xab
0x0002	0xab	0x34
0x0003	0xcd	0x12

为什么这几个函数名字里面都一个"cpu"?谈到字节序不谈 cpu 那就好比神采飞扬的谈起超级女声却对张靓颖是何许人也茫然不知.不同的 cpu 采用不同的字节序.看生产商自己喜欢了.其中,big endian 以 Motolora 的 PowerPC 系列 cpu 为代表,而 little endian 则以我家 Intel 的 x86 系列 cpu 为代表.所以这几个函数名字里边都会有 cpu 的字样,那么毫无疑问对于不同的 cpu,这几个函数执行的代码是不一样的.但是,凡是 xx_to_cpu 就说明函数的结果是给 cpu 使用的,反之如果是 cpu_to_xx 就说明是从 cpu 的字节序转换成目标字节序.

那么目标字节序应该是什么样子?我们先来看usb的情况.usb spec 2.0第八章,白纸黑字的规定了这么一句,usb总线上使用的是little-endian的字节序.所以,当初我们在处理bcs/bcb的时候一直在调用 lexx_to_cpu/cpu_to_lexx()这样的函数,或者准确地说,这样的宏.(usb spec 2.0, Chapter 8, 8.1 Byte/Bit Ordering:
Bits are sent out onto the bus least-significant bit (LSb) first, followed by the next LSb, through to the most significant bit (MSb) last. In the following diagrams, packets are displayed such that both individual bits and fields are represented (in a left to right reading order) as they would move across the bus. Multiple byte fields in standard descriptors, requests, and responses are interpreted as and moved over the bus in little-endian order, i.e., LSB to MSB.)

而与此同时,在灯火阑珊处,我们也依稀记得,在那份名为 SCSI Primary Commands-4 的规范,即那份在江湖上有着 SCSI 葵花宝典之美誉的 SPC-4 中,第三章,3.5, Bit and byte ordering 那一段,是这般描述的: If a field consists of more than one byte and contains a single value, the byte containing the MSB is stored at the lowest address and the byte containing the LSB is stored at the highest address (i.e., big-endian byte ordering).所以 request_buffer 回来的数据是采用 be 的字节序,因此我们这里的 c 要通过 be32_to_cpu()转换才能变成 cpu 使用的结果.反过来,当我们再次 copy 回 request_buffer 中的时候,要再使用 cpu_to_be32()给转回去.

最后如果你还要问,为什么要采用两种字节序?多麻烦啊?那我没什么可说的,你问上帝问真主问释迦牟尼去吧,也许他们能告诉你,不管白老鼠黑老鼠,只要不给猫逮住的就是好老鼠.

有多少爱可以胡来?(一)

上帝给了每个人一支书写人生的铅笔,却未曾给我们橡皮擦.但计算机的世界却并非如此,电脑用着用着觉得不正常了,按一下 reset 键就一切 ok 了. (当然你要是中了熊猫烧香啊中了冲击波啥的病毒那就另当别论

了,喂,别打岔行不行,我们讲正事呢.)如果人生也可以这样,那么星爷的那段经典的妇孺皆知的"人世间最悲哀的……假如……"的对白恐怕就没有意义了.

在驱动程序中,一个非常非常重要的概念就是错误处理.生活不是林黛玉,不会因为忧伤而风情万种,写代码不是写小说,不会因为作者的构思完美而天衣无缝.所以我们来看看在 usb-storage 中,我们是如何来进行错误处理的.

一切都得从那个结构体变量 struct scsi_host_template usb_stor_host_template 开始说起.其实这个结构体正是我们和 scsi 核心层最最关键的接口.我们知道这个结构体有很多内容,我们为 usb_stor_host_template 的许多成员赋了值,但是显然很多我们都没有讲.那么现在是时候去讲了.这其中,我们不妨把与错误处理相关的三个成员给揪出来.这些函数都是我们自己定义的,提供给 scsi core 那边去调用,就好像 queuecommand()一样.

```
430         /* error and abort handlers */
431         .eh_abort_handler =             command_abort,
432         .eh_device_reset_handler =      device_reset,
433         .eh_bus_reset_handler =         bus_reset,
```

好,让我们一个一个来看.先看两个与 reset 相关的函数,网友"要挑熟女"问我为什么有两个 reset 函数,很简单,当你的电脑有点小毛病了之后,你可以有两种选择,一种是注销就可以了,一种是重起才可以.device_reset 在这里对应的就是注销,bus_reset 对应的就是重起.当然这样说并不严谨,只能说,一般来说轻微一点,device_reset 就够了,如果严重一点,眼看着设备病入膏肓了,那么可能就要 bus_reset()了.

Ok,我们让代码来告诉你.

首先看到的是 device_reset().来自 drivers/usb/storage/scsiglue.c,

```
239 /* This invokes the transport reset mechanism to reset the state of the
240  * device */
241 /* This is always called with scsi_lock(srb->host) held */
242 static int device_reset(struct scsi_cmnd *srb)
243 {
244         struct us_data *us = (struct us_data *)srb->device->host->hostdata[0];
245         int result;
246
247         US_DEBUGP("%s called\n", __FUNCTION__);
248
249         scsi_unlock(srb->device->host);
250
251         /* lock the device pointers and do the reset */
252         down(&(us->dev_semaphore));
253         if (test_bit(US_FLIDX_DISCONNECTING, &us->flags)) {
254                 result = FAILED;
255                 US_DEBUGP("No reset during disconnect\n");
```

```
256         } else
257                 result = us->transport_reset(us);
258         up(&(us->dev_semaphore));
259
260         /* lock the host for the return */
261         scsi_lock(srb->device->host);
262         return result;
263 }
```

244 行,没啥好说的,星星还是那颗星星哟,月亮还是那个月亮,山也还是那座山哟,梁也还是那道梁,碾子是碾子,缸是缸哟,爹是爹来娘是娘,麻油灯啊,还吱吱地响,点的还那么丁点亮,喔哦...喔哦...只有那篱笆墙影子咋那么长,只有那篱笆墙影子咋那么长,还有那看家的狗叫的叫的叫的叫的咋就这么狂...80 后不可能没有听过毛阿敏的歌,不可能没有听过这首毛阿敏姐姐的成名作.20 多年过去,星星不再像那颗星星,月亮也不象那个月亮,河也不是那条河哟,房也不是那座房.然而,us 还是那个 us.就像那篱笆墙,影子还那么长.

253 行,首先看 US_FLIDX_DISCONNECTING flag 设了没有.显然,disconnecting 了就没有必要再 reset 了.关于 disconnect,你别急,讲完 reset 部分就该讲它了.

否则,会调用 us->transport_reset,我们前面已经说过了,很久很久以前,我们曾经为 us->transport_reset 赋值为 usb_stor_Bulk_reset,所以这里也就是函数 usb_stor_Bulk_reset()会被调用,usb_stor_Bulk_reset 定义于 drivers/usb/storage/transport.c 中,

```
1184 /* This issues a Bulk-only Reset to the device in question, including
1185  * clearing the subsequent endpoint halts that may occur.
1186  */
1187 int usb_stor_Bulk_reset(struct us_data *us)
1188 {
1189         US_DEBUGP("%s called\n", __FUNCTION__);
1190
1191         return usb_stor_reset_common(us, US_BULK_RESET_REQUEST,
1192                                 USB_TYPE_CLASS | USB_RECIP_INTERFACE,
1193                                 0, us->ifnum, NULL, 0);
1194 }
```

进入这个函数一看,很简单,也不干别的,就是调用 usb_stor_reset_common().于是,咱们接着来到了这个来自 driver/usb/storage/transport.c 中的 usb_stor_reset_common()函数.

```
1101 /* This is the common part of the device reset code.
1102  *
1103  * It's handy that every transport mechanism uses the control endpoint for
1104  * resets.
1105  *
1106  * Basically, we send a reset with a 20-second timeout, so we don't get
1107  * jammed attempting to do the reset.
1108  */
```

```
1109 static int usb_stor_reset_common(struct us_data *us,
1110               u8 request, u8 requesttype,
1111               u16 value, u16 index, void *data, u16 size)
1112 {
1113       int result;
1114       int result2;
1115       int rc = FAILED;
1116
1117       /* Let the SCSI layer know we are doing a reset, set the
1118        * RESETTING bit, and clear the ABORTING bit so that the reset
1119        * may proceed.
1120        */
1121       scsi_lock(us->host);
1122       usb_stor_report_device_reset(us);
1123       set_bit(US_FLIDX_RESETTING, &us->flags);
1124       clear_bit(US_FLIDX_ABORTING, &us->flags);
1125       scsi_unlock(us->host);
1126
1127       /* A 20-second timeout may seem rather long, but a LaCie
1128        * StudioDrive USB2 device takes 16+ seconds to get going
1129        * following a powerup or USB attach event.
1130        */
1131       result = usb_stor_control_msg(us, us->send_ctrl_pipe,
1132                   request, requesttype, value, index, data, size,
1133                   20*HZ);
1134       if (result < 0) {
1135             US_DEBUGP("Soft reset failed: %d\n", result);
1136             goto Done;
1137       }
1138
1139       /* Give the device some time to recover from the reset,
1140        * but don't delay disconnect processing. */
1141       wait_event_interruptible_timeout(us->dev_reset_wait,
1142                   test_bit(US_FLIDX_DISCONNECTING, &us->flags),
1143                   HZ*6);
1144       if (test_bit(US_FLIDX_DISCONNECTING, &us->flags)) {
1145             US_DEBUGP("Reset interrupted by disconnect\n");
1146             goto Done;
1147       }
1148
1149       US_DEBUGP("Soft reset: clearing bulk-in endpoint halt\n");
1150       result = usb_stor_clear_halt(us, us->recv_bulk_pipe);
1151
1152       US_DEBUGP("Soft reset: clearing bulk-out endpoint halt\n");
```

```
1153        result2 = usb_stor_clear_halt(us, us->send_bulk_pipe);
1154
1155        /* return a result code based on the result of the control message */
1156        if (result < 0 || result2 < 0) {
1157                US_DEBUGP("Soft reset failed\n");
1158                goto Done;
1159        }
1160        US_DEBUGP("Soft reset done\n");
1161        rc = SUCCESS;
1162
1163  Done:
1164        clear_bit(US_FLIDX_RESETTING, &us->flags);
1165        return rc;
1166 }
```

前面几行是赋值,然后usb_stor_report_device_reset()被调用. usb_stor_report_device_reset()定义于 drivers/usb/storage/scsiglue.c 中,

```
308 /* Report a driver-initiated device reset to the SCSI layer.
309  * Calling this for a SCSI-initiated reset is unnecessary but harmless.
310  * The caller must own the SCSI host lock. */
311 void usb_stor_report_device_reset(struct us_data *us)
312 {
313        int i;
314
315        scsi_report_device_reset(us->host, 0, 0);
316        if (us->flags & US_FL_SCM_MULT_TARG) {
317                for (i = 1; i < us->host->max_id; ++i)
318                        scsi_report_device_reset(us->host, 0, i);
319        }
320 }
```

315 行,scsi_report_device_reset(),drivers/scsi/scsi_error.c 中定义的.这个函数 scsi core 那边要求我们调用的,我们身不由己.然而关于这个函数的细节,只能说,世界太大,我们只在乎我们需要在乎的冬冬,其她的我们无暇顾及.我们只想在 960 万平方千米的一个角落里,静静的为自己的理想打拼,为自己寻找一份荣耀.usb 是我们 care 的,而 scsi 的核心,我们是不想去深究的,只有写 scsi 核心的同志们会在乎.江湖中流传这么一句话:女孩在乎的是下半生的幸福,男孩关注的是下半身的幸福.同样,Linux 世界里,每一个人在乎的冬冬是不一样的……言归正传,咱们是不需要关系这个函数怎么定义的,但是咱们需要知道什么时候会调用她,调用她干嘛?需要传递什么参数?首先,要传递三个参数,第一个,Scsi_Host 指针,一块 u 盘就有一个 Scsi_Host,然后第二个参数,channel,然后第三个参数 target,描述 scsi 设备位置的四个参数就三缺一了,缺的就是 LUN,因为一个 device 都 reset 那么就不会管她上面有几个 LUN 了,有几个都一起给她 reset.那么调用这个函数的目的是什么?告诉 scsi 核心,俺观察到某个设备 reset 了.至于 scsi 核心会如何处理呢,那咱管不着,也懒得去管.总而言之,言而总之,统而言之,言而统之,咱们的职责是在发现了一个设备 reset 之后立刻向上级汇报.

US_FL_SCM_MULT_TARG 这个 flag,咱们也提过好几次,她代表的是支持多个 target,这是设备本身的属性,不是咱们的代码愣给设备设的.对于这种设备,scsi_report_device_reset()就会被多调用几次,针对每一个 target 要 report 一次.

结束了 scsi_report_device_reset(),自然又回到了 usb_stor_reset_common(),1123 行,1124 行设置一个 flag,清除一个 flag,设置的是 US_FLIDX_RESETTING,清除的是 US_FLIDX_ABORTING,关于这两个 flag,一会咱们结合 command_abort()来讲.

1131 行,usb_stor_control_msg()被调用,再一次看到这个函数想必大家已经不再陌生了吧.她就是发送一个控制命令,其实我们已经很久没有讲控制传输了.这里结合参数来看看传送的什么命令.首先,us 还是那个 us,不再多说.然后,pipe 是 us->send_ctrl_pipe,就是发送控制管道.然后 request,requesttype 这些都是在调用 usb_stor_reset_common()的时候传递进来的参数,在 usb_stor_Bulk_reset()中可以看到,request 是 US_BULK_RESET_REQUEST,requesttype 是 USB_TYPE_CLASS | USB_RECIP_INTERFACE.

US_BULK_RESET_REQUEST 在 drivers/usb/storage/transport.h 中被设置为 0xff,这是和 usb mass storage class-Bulk Only transport 协议相对应的.该协议专门为 Bulk-Only Mass Storage 设备定义了一个请求,即 Reset.协议里说,this request is used to reset the mass storage device and its associated interface.协议中规定了,当 usb host 要发送命令 reset usb 设备的时候,需要通过发送控制管道发送一个请求,即前面提过的 ctrlrequest,其格式如下图所示:

Table 3.1 – Bulk-Only Mass Storage Reset

bmRequestType	*bRequest*	*wValue*	*wIndex*	*wLength*	Data
00100001b	11111111b	0000h	Interface	0000h	none

其中 bReques 这一位须设置为 255(FFh),wValue 设置为 0,wIndex 设置为 interface number,wLength 设置为 0.(而我们这里也确实这样做了,wIndex 被赋值为 us->ifnum,和上次咱们调用 usb_stor_control_msg 的时候传递的一样,显然 interface 还是那个 interface.江山会变,四季会变,咱们心中的 interface 始终不变.)

至于 requesttype,和咱们在 usb_stor_Bulk_max_lun()中讲的差不多,唯一的区别是控制数据传输方向,当时是 device to host,现在是 host to device,所以当时多了一个 USB_DIR_IN,而现在没有写 USB_DIR_OUT,原因很简单,USB_DIR_OUT 被定义为 0,所以或不或她无所谓.

嗯,酱紫,就完成了向设备发送 reset 命令的任务.返回值小于 0 就是出错了.

没出错那么就 1141 行,wait_event_interruptible_timeout()被调用.us->dev_reset_wait 咱们前面讲过,她是一个等待队列头,在 storage_probe()中被初始化,而以后在讲 storage_disconnect()时会讲到,有这么一句,wake_up(&us->dev_reset_wait),她唤醒的正是这里进入睡眠的进程,这里 1141 行,会进入睡眠,进入睡眠之前先判断 US_FLIDX_DISCONNECTING 这个 flag 有没有设置,要是设了就没有必要睡眠了,直接退出吧.1144 行,再判断一次,是真的设了这个 flag 那么直接 goto Done,返回 rc,rc 就是初值 FAILED.返回之前先清除 US_FLIDX_RESETTING flag. 关于 wait_event_interruptible_timeout()这个函数,我们当初在分析 usb_stor_scan_thread()的时候已经详细的讲过了,所以这里不需要再浪费你我的青春去多讲了.

当然,如果 US_FLIDX_DISCONNECTING 并没有设置,那么 6s 钟时间到了,睡到自然醒,1150 行 1153 行,usb_stor_clear_halt(),又是一个很亲切的函数,表忘了当前我们在讲 GET MAX LUN 的时候就专门介绍了这个函数,而且那时候我们就已经说过,有一种情况下我们要调用这个函数,说的正是我们这里的情况,即当设备 reset 之后,需要清楚 halt 这个 feature,然后端点才能正常工作.对于我们的两个 bulk 端点,只要有一个清 halt feature 失败了,那么整个这个负责 reset 的函数 usb_stor_reset_common 就算失败了,并且因此会返回 FAILED,而且在返回之前先把 US_FLIDX_RESETTING 这个 flag 给去掉.

然后到了这里,usb_stor_reset_commond 该返回了,然后我们惊讶的发现, usb_stor_Bulk_reset() 也该返回了,再然后我们又惊讶的发现,device_reset()也该返回了.就这样,我们走完了 device_reset()这么一个函数.之所以简单是因为它的使命本身就很简单,其实就是给设备发送一个 reset 的 request,然后 clear 掉 halt feature,保证设备的端点没有停止.就这些,这就够了.

有多少爱可以胡来?(二)

device_reset()完了之后我们来看 bus_reset().同样来自 drivers/usb/storage/scsiglue.c 中.

```
265 /* This resets the device's USB port. */
266 /* It refuses to work if there's more than one interface in
267  * the device, so that other users are not affected. */
268 /* This is always called with scsi_lock(srb->host) held */
269 static int bus_reset(struct scsi_cmnd *srb)
270 {
271         struct us_data *us = (struct us_data *)srb->device->host->hostdata[0];
272         int result, rc;
273
274         US_DEBUGP("%s called\n", __FUNCTION__);
275
276         scsi_unlock(srb->device->host);
277
278         /* The USB subsystem doesn't handle synchronisation between
279          * a device's several drivers. Therefore we reset only devices
280          * with just one interface, which we of course own. */
281
282         down(&(us->dev_semaphore));
283         if (test_bit(US_FLIDX_DISCONNECTING, &us->flags)) {
284                 result = -EIO;
285                 US_DEBUGP("No reset during disconnect\n");
286         } else if (us->pusb_dev->actconfig->desc.bNumInterfaces != 1) {
287                 result = -EBUSY;
288                 US_DEBUGP("Refusing to reset a multi-interface device\n");
289         } else {
290                 rc = usb_lock_device_for_reset(us->pusb_dev, us->pusb_intf);
291                 if (rc < 0) {
292                         US_DEBUGP("unable to lock device for reset: %d\n", rc);
```

```
293                     result = rc;
294             } else {
295                     result = usb_reset_device(us->pusb_dev);
296                     if (rc)
297                             usb_unlock_device(us->pusb_dev);
298                     US_DEBUGP("usb_reset_device returns %d\n", result);
299             }
300     }
301     up(&(us->dev_semaphore));
302
303     /* lock the host for the return */
304     scsi_lock(srb->device->host);
305     return result < 0 ? FAILED : SUCCESS;
306 }
```

看完了 device_reset 再来看 bus_reset 应该就不难了.这个函数本身写的也特别清楚,注释很详细加上又有一些调试信息,基本上,中关村那些抱着小孩卖毛片的妇女同志们应该都能看懂这个函数了.咱们再简单介绍一下,和 device_reset 不同的地方在于,device_reset 实际上只是设备本身的复位,而 bus_reset 的意义就更加广泛了,它涉及到从 usb core 这一层面来初始化这个设备,千万不要混淆了,bus reset 不是说 usb bus 的 reset,而是 scsi bus,即 scsi 总线.哪来的 scsi 总线?我们模拟的,假的.难道你忘记了我们模拟了一个 scsi host,一个 scsi device 么,实际上也就是模拟了一条 scsi 总线.而这个冬冬的 reset 就意味着整个 driver 都得重新初始化,即从 usb core 那边开始,确切的说是 usb hub 那边,先给咱们在 usb 总线上来重新分配一个地址,重新建立起之前的配置,重新选择 altsetting,也就是说对于咱们这个设备,相当于重新经历了一次 usb 的总线枚举.而之后,咱们的 storage_probe()会重新被调用,整个模拟的 scsi 总线将从头再来,所以说 bus_reset 是一场很大规模的运动,而不像 device reset 那样属于小打小闹式的.

283 行,还是老套路,检查是不是 disconnecting 了,要是的话就甭 reset 了,省省事吧,别瞎折腾了.

286 行,还记得咱们说过一个 interface 对应一个 driver 吗,一个 device 可能有多个 interface,也因此可能对应多个 driver,那么咱们这里对一个 device reset 就只能针对那种一个 device 只有一个 interface 的情况,因为要不然您就影响别人了,就好比,假如你们家是在一个森林里边,前不挨村后不着店的房子, 或者说你们家的周边地形和那个史上最牛的重庆钉子户差不多,那您如果想把自己建好的房子给拆了,那可能没人管,但假如您住的是楼房,您的卧室跟邻居家的卧室一墙之隔,或者浴室跟邻居家的浴室一墙之隔,那您要是敢把您那堵墙拆了,毋庸置疑,邻居非跟您急不可.

然后 289 行到 300 行,总共三个来自 usb core 那边的关键函数,usb_lock_device_for_reset/usb_reset_device/usb_unlock_device.usb core 都为您准备好了,您要从总线上来 reset 设备,首先调用的是 usb_lock_device_for_reset,这是 usb 核心层的函数,来自 drivers/usb/core/usb.c 中,成功执行就返回 1 或者 0,失败了就返回一个负的错误代码.295 行的 usb_reset_device(),同样来自 usb 核心层,drivers/usb/core/hub.c 中,是 usb 核心层提供给咱们的用来重新初始化一个设备的函数.成功返回 0,否则就返回负的错误代码.出错了就得调用 usb_unlock_device 来释放锁.

最后 305 行,result 为 0 才是返回成功,否则返回 FAILED.

能看懂 296 行,297 行吗?我们说了,usb_lock_device_for_reset 可以返回负数,可以返回 1,可以返回 0.返回负数的情况就是 291 到 293 行所做的事情,而对于 usb_lock_device_for_reset 来说,它返回 1 表示我们在执行了 usb_reset_device 之后要调用 usb_unlock_device 来释放锁,而返回 0 表示我们在执行了 usb_reset_device 之后不需要调用 usb_unlock_device 来释放锁.如果你对这些很好奇,那么你可以看一下 usb core 的代码.特别是看一下关于两个宏的代码,USB_INTERFACE_BINDING 和 USB_INTERFACE_BOUND.

于是,我们就这样从这个 bus_reset()函数打马而过了.

最后我们来看 command_abort().很显然这是一个错误处理函数, 她的职责很明确, 试图去中止当前的命令.同样来自 drivers/usb/storage/scsiglue.c:

```
202 /* Command timeout and abort */
203 /* This is always called with scsi_lock(srb->host) held */
204 static int command_abort(struct scsi_cmnd *srb )
205 {
206         struct Scsi_Host *host = srb->device->host;
207         struct us_data *us = (struct us_data *) host->hostdata[0];
208
209         US_DEBUGP("%s called\n", __FUNCTION__);
210
211         /* Is this command still active? */
212         if (us->srb != srb) {
213                 US_DEBUGP ("-- nothing to abort\n");
214                 return FAILED;
215         }
216
217         /* Set the TIMED_OUT bit.  Also set the ABORTING bit, but only if
218          * a device reset isn't already in progress (to avoid interfering
219          * with the reset).  To prevent races with auto-reset, we must
220          * stop any ongoing USB transfers while still holding the host
221          * lock. */
222         set_bit(US_FLIDX_TIMED_OUT, &us->flags);
223         if (!test_bit(US_FLIDX_RESETTING, &us->flags)) {
224                 set_bit(US_FLIDX_ABORTING, &us->flags);
225                 usb_stor_stop_transport(us);
226         }
227         scsi_unlock(host);
228
229         /* Wait for the aborted command to finish */
230         wait_for_completion(&us->notify);
231
232         /* Reacquire the lock and allow USB transfers to resume */
233         scsi_lock(host);
234         clear_bit(US_FLIDX_ABORTING, &us->flags);
```

```
235         clear_bit(US_FLIDX_TIMED_OUT, &us->flags);
236         return SUCCESS;
237 }
```

既然是阻止某个命令,那么传递进来的参数当然是 struct scsi_cmnd 的指针了.前两行的赋值无须多说.

212 行,us->srb 表示的是当前的 srb,而 srb 是这个函数传进来的参数,command_abort()希望的是中止当前的命令,而假如传进来的参数根本就不是当前的命令,那么肯定有问题.啥也不说了,直接返回 FAILED.

然后,在 abort 的时刻,设置 US_FLIDX_TIMED_OUT 的 flag,然后如果设备没有 reset,那么继续再设置另一个 flag,US_FLIDX_ABORTING,然后调用 usb_stor_stop_transport(),这个函数的目的很简单,阻止任何进行中的传输,这个函数定义于 drivers/usb/storage/transport.c 中:

```
730 /* Stop the current URB transfer */
731 void usb_stor_stop_transport(struct us_data *us)
732 {
733         US_DEBUGP("%s called\n", __FUNCTION__);
734
735         /* If the state machine is blocked waiting for an URB,
736          * let's wake it up.  The test_and_clear_bit() call
737          * guarantees that if a URB has just been submitted,
738          * it won't be cancelled more than once. */
739         if (test_and_clear_bit(US_FLIDX_URB_ACTIVE, &us->flags)) {
740                 US_DEBUGP("-- cancelling URB\n");
741                 usb_unlink_urb(us->current_urb);
742         }
743
744         /* If we are waiting for a scatter-gather operation, cancel it. */
745         if (test_and_clear_bit(US_FLIDX_SG_ACTIVE, &us->flags)) {
746                 US_DEBUGP("-- cancelling sg request\n");
747                 usb_sg_cancel(&us->current_sg);
748         }
749 }
```

这里有两个 flag,US_FLIDX_URB_ACTIVE 和 US_FLIDX_SG_ACTIVE,前一个 flag 咱们此前在讲 storage_probe()的时候遇见过,当时咱们为了获得 max lun,曾经提交过 urb,而在 usb_stor_msg_common()函数中,在 urb 被成功提交了之后,调用 set_bit()宏为 us->flags 设置了这一位,于是这里就可以先判断,如果确实设置了,那么此时就可以调用 usb_unlink_urb 来从 urb 队列中取下来,因为没有必要再连接在上面了.

而 US_FLIDX_SG_ACTIVE 这个 flag 咱们也不陌生,在 bulk 传输里会用到.设置这个 flag 的是 bulk 传输中的函数 usb_stor_bulk_transfer_sglist(),她会调用 set_bit 函数来设置这个 flag. 于是这里也判断,如果确实设置了,那么此时就可以调用 usb_sg_cancel 来把 sg 取消.

看得出,usb_stor_stop_transport()的决心很坚定,目标很简单,就是要让你的传输进行不下去,有 urb 就取消 urb,有 sg 就取消 sg,大有人挡杀人佛当杀佛的魄力和勇气.

这时,在 230 行,咱们看到了等待函数,wait_for_completion(&us->notify),command_abort()函数进入等待.谁来唤醒它呢?usb_stor_control_thread()中的 complete(&(us->notify))唤醒她.

一种情况,您这里睡眠中,usb 设备断开了,于是 usb_stor_control_thread 要退出来,于是它会调用 complete_and_exit()来唤醒 command_abort()并且退出 usb_stor_control_thread(),这样一来的话那么世界从此就清静了.

第二种情况,一个 scsi 命令还没有开始执行,或者说即将要开始执行的时候,我们调用了 command_abort(),那么对应于 usb_stor_control_thread()中的 322 到 325 那几行的 if 语句,发现 US_FLIDX_TIMED_OUT flag 被设置了,于是,命令也甭执行了,直接 goto SkipForAbort,然后调用 complete(&(us->notify))唤醒咱们这里进入睡眠的进程.同时,对于 usb_stor_control_thread()这边来说,将 us->srb 设置为 NULL 就一切 ok 了,可以开始下一轮循环了.

第三种情况, 一个 scsi 命令执行完了之后,我们才执行的 command_abort,那么这种情况下,由于我们设置了 US_FLIDX_TIMED_OUT 标志,对应于 usb_stor_control_thread()那边的 396,397 行的 if 语句,也没什么特别的,同样的,执行 complete(&(us->notify))唤醒咱们这个进入睡眠的进程吧.并且将 us->srb 置为 NULL.

最后,234,235 行,清除这两个 US_FLIDX_ABORTING 和 US_FLIDX_TIMED_OUT 这两个 flag,接下来的传输还是外甥打灯笼-照旧.而 command_abort()函数本身当然返回 SUCCESS.阻止了别人,她就成功了.所以它返回 SUCCESS.

至此我们就算讲完了这三个负责错误处理的函数了.那么关于这两个 flag 的故事呢.让我们来看看.实际上你搜索一下整个内核目录,只有 command_abort()函数中 set 了这两个 flag,而如果 command_abort()执行成功了,会在返回之前,把两个 flag 给清除掉.而需要测试 US_FLIDX_TIMED_OUT 这个 flag 的地方,仅仅只有 usb_stor_control_thread()中和 usb_stor_invoke_transport()中.关于前者正是我们刚才所讲的那样,除了唤醒 command_abort()以外,就是设置 us->srb 为 NULL,但总的来说,基本上对 scsi 那边没有什么影响.而在 usb_stor_invoke_transport()中,如果我们遇到了 US_FLIDX_TIMED_OUT 被设置,那么情况会有所不同,因为这时候我们是在与 scsi 层打交道,我们是因为要执行一个 scsi 命令才进入到 usb_stor_invoke_transport()的,所以在这种情况下如果遇到了 abort,那么我们至少得给 scsi 一个交待,正如我们在 usb_stor_invoke_transport()中第 724 行那一小段看到的那样,我们回复给 scsi 层的是 srb->result,我们把它设成了 DID_ABORT<<16.同时,对于 Bulk-only 的设备,我们还需要把我们的设备进行 reset.这是 bulk-only 的传输协议里边规定的,即,在一个 abort 之后,必须要进行一次 reset,要不设备下次没法工作.(参见 Bulk-only transport 5.3.3.1 Phase Error/5.3.4 Reset Recovery)

再来看另一个 flag,US_FLIDX_ABORTING.实际上没有人专门去测试这个 flag 有没有设置,真正被测试的是另一个 flag,ABORTING_OR_DISCONNECTING,关于这个 flag 的定义我们早年曾经列出来过,实际上意思很明显,或者是设置了 Aborting 的 flag,或者就是设置了 Disconnecting 的 flag.很多情况下我们实际上要判断的就是这两者之中任何一个是否被设置了,因为很显然,很多情况下,如果这两个中的任何一个被设置了,那么我们的某些事情就没有必要继续了,因为当你明明知道你们之间没有未来,你有何必浪费双方的感情呢?(当然,一夜情不在我们讨论的范畴之内.)关于 ABORTING_OR_DISCONNECTING,一共有

两个函数中需要判断它,一个是 usb_stor_msg_common(),一个是 usb_stor_bulk_transfer_sglist().其实,仔细回顾一下这两个函数,你就不难发现,在 usb_stor_msg_common()中,其实我们就是在提交一个 urb 之前先检查这个 flag 有没有设置,提交了之后再次检查一下,实际上就相当于是给你两次机会,因为提交了之后就相当于表白了之后,但是双方可能还没有发生什么实质性的故事,在这一刻如果你意识到这是一段没有结果的恋情,你还可以选择放弃.但是过了这第二次判断,那么结果就将是生米煮成熟饭了.而在 usb_stor_bulk_transfer_sglist()中,其实道理是一样的,就是把 urb 的概念换成 sg,仅此而已.

Ok,到现在我们可以来看最后一个重量级的 flag 了.那就是 US_FLIDX_DISCONNECTING.而将带我们走入我们整个故事的最后一个重量级函数, storage_disconnect().而这个函数也将宣告我们整个故事的结束,毕竟天下无不散的宴席,从我开始写这个故事的时候,我就知道一切都已注定,注定了开始,注定了结束.

当梦醒了天晴了

多情自古伤离别,更那堪,冷落清秋节!
今宵酒醒何处?
杨柳岸,晓风残月.
此去经年,应是良辰好景虚设.
便纵有,千种风情,更与何人说?

伴随着婉约派才子,词坛浪子柳永的这首令人肝肠寸断的<<雨霖铃>>,我们来到了最后一个重要的函数,storage_disconnect.

usb 设备的热插拔特性注定了我们应该在设备插入的时候做一些事情,在设备拔出的时候做一些事情.主机和 usb 设备的暧昧关系体现在,需要她的时候,要多缠绵有多缠绵,如胶似漆,如鱼得水.但是,有爱就有痛,有一天 usb 设备必定要离开主机.对于主机来说,人生没有 usb 设备并不会不同.而且,事实上,usb 的即插即用特性也让主机知道,usb 设备并不曾真的离去,他们还会再相逢.前面见面的时候调用了 storage_probe 来让彼此接受对方,现在就该调用 storage_disconnect 函数来分手.分手也许需要分财产,而 usb 设备离开主机也需要处理一些后事.

相比 probe,disconnect 函数就简单多了,造人需要辛苦 30 分钟,人流只要 3 分钟,看过杨千嬅的那部<<饺子>>之后就知道人流并不是一件麻烦事,难怪北京的街边电线杆上墙壁上到处写着无痛人流轻松搞定的广告词. storage_disconnect()函数定义在 drivers/usb/storage/usb.c 中,这个函数不长,

```
1027 /* Handle a disconnect event from the USB core */
1028 static void storage_disconnect(struct usb_interface *intf)
1029 {
1030         struct us_data *us = usb_get_intfdata(intf);
1031
1032         US_DEBUGP("storage_disconnect() called\n");
1033
1034         /* Prevent new USB transfers, stop the current command, and
1035          * interrupt a device-reset delay */
```

```
1036        set_bit(US_FLIDX_DISCONNECTING, &us->flags);
1037        usb_stor_stop_transport(us);
1038        wake_up(&us->dev_reset_wait);
1039
1040        /* Interrupt the SCSI-device-scanning thread's time delay, and
1041         * wait for the thread to finish */
1042        wake_up(&us->scsi_scan_wait);
1043        wait_for_completion(&us->scsi_scan_done);
1044
1045        /* Wait for the current command to finish, then remove the host */
1046        down(&us->dev_semaphore);
1047        up(&us->dev_semaphore);
1048        scsi_remove_host(us->host);
1049
1050        /* Wait for everything to become idle and release all our resources */
1051        usb_stor_release_resources(us);
1052        dissociate_dev(us);
1053 }
```

如果直到现在你还不知道 1030 行在干嘛,那我想问一下你他妈的是不是在耍我?虽然 usb_get_intfdata()这个函数的确是第一次露面,但是这里的含义已然是司马昭之心路人皆知.usb_get_infdata()我们是没有讲过,但是我们讲过 usb_set_intfdata().想当年,associate_dev()中,我们调用 usb_set_intfdata(intf,us),当时我们分析了,这样做的结果就是使得 %intf->dev->driver_data=us,而现在我们调用 usb_get_intfdata(intf)的作用就是把 us 从中取出来,赋给我们这里的临时指针 us.

1036 行,全文中唯一一处设置 US_FLIDX_DISCONNECTING 这个 flag 的地方就在这里.

1037 行, usb_stor_stop_transport(us),这个函数我们可是刚刚才讲过,你别说你就忘记了,就在 command_abort()里调用的.目的就是停掉当前的 urb 和 sg,如果有的话.

1038 行, wake_up(&us->dev_reset_wait),我们也已经讲过了,就是在讲 device_reset()讲到的,当时在 usb_stor_reset_common()中,会使用 wait_event_interruptible_timeout()来进入睡眠,睡眠的目的是给 6 秒钟来让设备从 reset 状态恢复过来,但是如果在这期间我们要断开设备了,那么当然就没有必要再让那边继续睡眠了,设备都要断开了,还有什么恢复的意义呢?所以对于这种情况,我们回过头来看 usb_stor_reset_common(),会发现之后该函数立马从睡眠中醒来,然后清除掉为 reset 而设置的 flag,US_FLIDX_RESETTING,然后就返回了,返回值是 FAILED.

1042 行, wake_up(&us->scsi_scan_wait),和上面这种情况几乎相同,不同的是这次唤醒的是 usb_stor_scan_thread,这个函数里边也会因为 delay_use 的设置而调用 wait_event_interrruptible_timeout 去等待去睡眠,所以这里机理是一样的.而与此同时,1043 行, wait_for_completion(&us->scsi_scan_done),恰恰是是等待对方的结束,我们注意到,在 usb_stor_scan_thread()中最后一句话, complete_and_exit(&us->scsi_scan_done, 0),即唤醒咱们这里这个 storage_disconnect()同时结束它自己.应该说这样就实现了一个同步机制.就是说因为我们

之后马上要做的就是清理门户了,把一些不要的资源都释放掉,所以我们首先必须保证我们的进程都退出来,资源都不再被人使用,这样我们才可以放心的去做我们的清理工作.

1048 行,scsi_remove_host()被调用,这是和最早的 scsi_add_host 相对应的.都是调用 scsi core 提供的函数.

1051 行,usb_stor_release_resources(us),这个则是和我们当初那个 usb_stor_acquire_resources(us)相对应.而 1052 行的 dissociate_dev(us)则是和当初那个 associate_dev()相对应.我们来看一下具体代码,来自 drivers/usb/storage/usb.c,把这两个函数的代码都一并贴出来:

```
815 /* Release all our dynamic resources */
816 void usb_stor_release_resources(struct us_data *us)
817 {
818         US_DEBUGP("-- %s\n", __FUNCTION__);
819
820         /* Kill the control thread.  The SCSI host must already have been
821          * removed so it won't try to queue any more commands.
822          */
823         if (us->pid) {
824
825                 /* Wait for the thread to be idle */
826                 down(&us->dev_semaphore);
827                 US_DEBUGP("-- sending exit command to thread\n");
828
829                 /* If the SCSI midlayer queued a final command just before
830                  * scsi_remove_host() was called, us->srb might not be
831                  * NULL.  We can overwrite it safely, because the midlayer
832                  * will not wait for the command to finish.  Also the
833                  * control thread will already have been awakened.
834                  * That's okay, an extra up() on us->sema won't hurt.
835                  *
836                  * Enqueue the command, wake up the thread, and wait for
837                  * notification that it has exited.
838                  */
839                 scsi_lock(us->host);
840                 us->srb = NULL;
841                 scsi_unlock(us->host);
842                 up(&us->dev_semaphore);
843
844                 up(&us->sema);
845                 wait_for_completion(&us->notify);
846         }
847
848         /* Call the destructor routine, if it exists */
```

```
849         if (us->extra_destructor) {
850                 US_DEBUGP("-- calling extra_destructor()\n");
851                 us->extra_destructor(us->extra);
852         }
853
854         /* Finish the host removal sequence */
855         if (us->host)
856                 scsi_host_put(us->host);
857
858         /* Free the extra data and the URB */
859         if (us->extra)
860                 kfree(us->extra);
861         if (us->current_urb)
862                 usb_free_urb(us->current_urb);
863
864 }
865
866 /* Dissociate from the USB device */
867 static void dissociate_dev(struct us_data *us)
868 {
869         US_DEBUGP("-- %s\n", __FUNCTION__);
870
871         /* Free the device-related DMA-mapped buffers */
872         if (us->cr)
873                 usb_buffer_free(us->pusb_dev, sizeof(*us->cr), us->cr,
874                                 us->cr_dma);
875         if (us->iobuf)
876                 usb_buffer_free(us->pusb_dev, US_IOBUF_SIZE, us->iobuf,
877                                 us->iobuf_dma);
878
879         /* Remove our private data from the interface */
880         usb_set_intfdata(us->pusb_intf, NULL);
881
882         /* Free the structure itself */
883         kfree(us);
884 }
```

823 行,判断 us 的 pid,这个 pid 是哪来的?很显然,usb_stor_release_resources 和咱们前面说过的 usb_stor_acquire_resources 函数是一对,us->pid 也正是来自 usb_stor_acquire_resources()函数,当时在调用 kernel_thread 启动 usb_stor_control_thread 的时候,记下了 kernel_thread()的返回值,并把她赋给了 us->pid,实际上 kernel_thread()对于父进程来说,返回值就是子进程的 pid,也就是说当年创建的精灵进程的 pid 是被记录下来了的.写代码的人老辣的编程功底可见一斑.

840 行,设置 us->srb 为 NULL.

844 行和 845 行,up(&us->sema),以及 wait_for_completion(&us->notify),知道这两句干嘛的吗?还记得 usb_stor_control_thread(),当初讲到 down_interruptible(&us->sema),咱们就说该守护进程进入了睡眠,那么谁能把她唤醒,除了前面讲的 queuecommand 之外,这里同样也是唤醒她的代码,并且这个函数在唤醒别人之后,自己执行 wait_for_completion 函数来进入等待.好,再次回到那个函数吧,usb_stor_control_thread(),303 行,被 up()唤醒的 down_interruptible 返回 0.然后 312 行到 316 行的这个 if 小段当然就会执行了,因为 us->srb 这时候毫无疑问,等于 NULL,所以 break 会被执行,从而结束 for 死循环,从而 usb_stor_control_thread 函数到了最后一行,最终调用 complete_and_exit(&(us->notify),0)来结束精灵进程自己并且唤醒我们这里的进入等待的 usb_stor_release_resources.很显然,这里两个进程通过四个函数,或者说两个组合,来实现了进程的同步,这正是 Linux 中内核同步机制的典型应用.(两大组合指的是:up 和 down,wait_for_completion 和 complete.)

继续往下,849 行,struct us_data 还有一个元素,extra_data_destructor extra_destructor,这是一个函数指针,实际上对某些设备来说,她们自己定义了一些额外的函数,在退出之前需要执行,比如 Datafab USB 紧凑读卡器,不过像这样的变态的设备很少,对于大多数设备来说,这个函数指针都为空.如果定义了,那么就去执行她.

然后 856 行,scsi_host_put,只是对这个 host 的引用计数减 1,如果该 host 的引用计数达到 0 了,那么将释放其对应的 Scsi_Host 数据结构所占的空间.一个 scsi 卡在其调用 scsi_host_alloc 的时候会被设置引用计数为 1,引用计数就是表征有多少进程使用了这个资源,我记得外企面试的时候也常常会问到这个,2005 年初,大四下的时候,去张江软件科技园,参加 SAP 的面试,当时谈到一个无盘工作站的话题,面试官就问我,既然客户端都没有硬盘,那么如果多个人同时读写服务器上的同一个文件,那岂不是乱套了,当时这个问题一问我就傻了.后来想想,也许这个和引用计数用点关系吧,再用上那个所谓的写拷贝(copy-on-write)技术.不过我还是挺喜欢 SAP,可惜,人家看不上我.

接下来 859 行,判断 us->extra,us->extra 就是为前面那个 extra_data_destructor 函数准备的参数,一般不会申请,如果申请过,那就释放她.

再接下来,current_urb 也没用了,释放吧.释放 urb 需要调用专门的函数 usb_free_urb,她来自 drivers/usb/core/urb.c 中.

到这里咱们前面占有的资源基本上就释放掉了.

再看下一个函数,dissociate_dev(),很显然,这个函数和咱们前面讲的 associate_dev()函数是一对.在 associate_dev 函数中,咱们调用了 usb_buffer_alloc()函数,先后为 us->cr 和 us->iobuf 分配了空间,所以这里首先调用相对应的函数 usb_buffer_free()来释放这两段空间,然后,然后咱们曾经调用 usb_set_intfdata 来令 us->pusb_intf 的所对应的设备的 driver_data 指向了 us,所以这里 usb_set_intfdata()再次调用从而让 us->pusb_intf 的 driver_data 指向空.最后,调用 kfree 释放 us.好了,终于,世界清静了.

最后的最后,我们还有一点点内容要讲,那就是分析一下我们故事中的锁机制,至少我们曾经承诺过,要到最后才能讲锁机制,理由很简单,只有我们把整个故事都弄明白了,我们才可能弄清楚为什么在某个地方要加锁,因为锁机制永远都是牵连着多处的代码的,它不是一个人在战斗!

其实世上本有路,走的人多了,也便没了路

其实信号量这东西,就像北京户口,你占了一个名额,我就可能没有了名额.但是有些事情,没有北京户口你又办不成.比如我满怀壮志的走到医院向医生表达说我希望死了以后可以捐献遗体,可得到的只是医生冷冷的回复,对不起,你没有资格,因为你没有户口.

我们前面说过,Linux 中,有信号量,有自旋锁,有互斥锁,自旋锁或者互斥锁从某种意义上来说就只是一种特殊的信号量,即信号量意味着资源数量有限,但这个有限也许可能比如像每年的北京户口的名额,有若干个,而锁反映的就是更加有限,限制到了数量为一,即类似于所谓的一夫一妻制.她只属于你一个人,一旦你占有了她,如果别人还要想得到她,除非你释放.或者说除非你抛弃了她.

具体到 usb storage,我们不管信号量和锁在 Linux 中是怎么实现的,它们之间是否有区别对我们来说也无所谓,事实上,usb-storage 中使用的都是锁,即便是信号量,也是把它当成锁来使用,可曾记得我们当初把信号量的初始化为 1 了?当信号量初始化为 1,那么对我们来说,它就相当于退化为一把锁了.而锁,只有两种状态,上锁和解锁.

此前我们见过很多次两组函数,但我们一直绝口不提.

她们就是 scsi_lock()和 scsi_unlock(),以及 down(&us->dev_semaphore)和 up(&us->dev_semaphore).

先来看第一组.她们是我们自己定义的宏.来自 drivers/usb/storage/usb.h.

```
174 /* The scsi_lock() and scsi_unlock() macros protect the sm_state and the
175  * single queue element srb for write access */
176 #define scsi_unlock(host)       spin_unlock_irq(host->host_lock)
177 #define scsi_lock(host)         spin_lock_irq(host->host_lock)
```

显然,这两个函数就像牛郎织女一样,是一对.而她们的作用,就是利用自旋锁来保护资源,这把锁就是 struct Scsi_Host 的一个成员 spinlock_t *host_lock.那么在什么情况下需要使用这两个自旋锁函数来保护资源呢?

当你要写 us->srb 的时候,(不是写 us->srb 的元素,而是写 us->srb,比如令 us->srb=NULL),这种时候,你需要使用这把自旋锁.另一种情况是,当你调用 scsi mid layer(scsi 中层)的函数时,有时候这些函数要求调用者拥有这把锁,即 host_lock.

搜索一下,发现一共有 7 处使用了 scsi_lock.

我们列举一些来.

第一处,drivers/usb/storage/usb.c:usb_stor_release_resources()函数中,

```
839             scsi_lock(us->host);
840             us->srb = NULL;
841             scsi_unlock(us->host);
```

这个不用解释了吧,赤裸裸的写 us->srb,自然要用 scsi_lock/scsi_unlock.

第二处,drivers/usb/storage/transport.c:usb_stor_reset_common()函数中,

```
1121        scsi_lock(us->host);
1122        usb_stor_report_device_reset(us);
1123        set_bit(US_FLIDX_RESETTING, &us->flags);
1124        clear_bit(US_FLIDX_ABORTING, &us->flags);
1125        scsi_unlock(us->host);
```

这就是第二种情况,因为这里调用了 usb_stor_report_device_reset(),这个函数来自 drivers/usb/storage/scsiglue.c,也是我们定义的,

```
308 /* Report a driver-initiated device reset to the SCSI layer.
309  * Calling this for a SCSI-initiated reset is unnecessary but harmless.
310  * The caller must own the SCSI host lock. */
311 void usb_stor_report_device_reset(struct us_data *us)
312 {
313         int i;
314
315         scsi_report_device_reset(us->host, 0, 0);
316         if (us->flags & US_FL_SCM_MULT_TARG) {
317                 for (i = 1; i < us->host->max_id; ++i)
318                         scsi_report_device_reset(us->host, 0, i);
319         }
320 }
```

注意到,她里边调用了scsi_report_device_reset(),后者来自drivers/scsi/scsi_error.c,这正是scsi mid layer 定义的函数,这个函数的注释说得很清楚,调用她时必须要拥有 host lock.

有人说还有第三种情况,drivers/usb/storage/usb.c,usb_stor_control_thread()函数中,

```
318             /* lock access to the state */
319             scsi_lock(host);
320
321             /* has the command timed out *already* ? */
322             if (test_bit(US_FLIDX_TIMED_OUT, &us->flags)) {
323                     us->srb->result = DID_ABORT << 16;
324                     goto SkipForAbort;
325             }
326
327             /* don't do anything if we are disconnecting */
328             if (test_bit(US_FLIDX_DISCONNECTING, &us->flags)) {
329                     US_DEBUGP("No command during disconnect\n");
330                     goto SkipForDisconnect;
331             }
```

```
332
333             scsi_unlock(host);
```

您真是太有才了,这段代码都被您注意到了.的确看起来,这里无非是调用了 test_bit,但是却使用了 scsi_lock/scsi_unlock 这对冤家,这是什么原因?其实是这样的,我们注意到这里有两个 goto 语句,

```
387 SkipForAbort:
388                 US_DEBUGP("scsi command aborted\n");
389             }
390
391             /* If an abort request was received we need to signal that
392              * the abort has finished.  The proper test for this is
393              * the TIMED_OUT flag, not srb->result == DID_ABORT, because
394              * a timeout/abort request might be received after all the
395              * USB processing was complete. */
396             if (test_bit(US_FLIDX_TIMED_OUT, &us->flags))
397                 complete(&(us->notify));
398
399             /* finished working on this command */
400 SkipForDisconnect:
401             us->srb = NULL;
402             scsi_unlock(host);
```

看 401 行,us->srb=NULL,还是老一套.这才是为什么之前要用 scsi_lock 的真正原因.因为我要跳转,而跳过去以后需要写 us->srb,所以要获得 host lock,当然我们可以把 scsi_lock 写在 test_bit()之后,比如

```
321             /* has the command timed out *already* ? */
322             if (test_bit(US_FLIDX_TIMED_OUT, &us->flags)) {
323                 us->srb->result = DID_ABORT << 16;

这里插入一行==>      scsi_lock(host);
324                 goto SkipForAbort;
```

这样改动代码也可以,显得更加严谨,不过改不改意义并不大.

当然,这一段测试超时的代码本身被使用上的可能性也不大,因为这里命令还没有开始执行呢,试想,一个命令还没有开始就超时,这种情况论几率,和中国国家男足杀进世界杯差不多吧.

再来看第二组,关于 us->dev_semaphore,它更多的体现出来是一种进程间的同步机制.2.6.10 的内核代码中一共有 6 处使用了这个信号量.分别是在 device_reset()中,bus_reset()中,usb_stor_control_thread()中,usb_stor_acquire_resources()中,usb_stor_release_resources()中,storage_disconnect()中.这几样东西之间是存在一种相互制约的关系.

比如,一种情景是,在 device_reset()或者 bus_reset()中上了锁,所以 usb_stor_control_thread()这边就不可以在这个时候执行命令,usb_stor_acquire_resources()中的 GET_MAX_LUN 也不能执行,因为很显然,正在 reset 呢,就好比你的电脑正在重起你当然不可能执行一个打开浏览器的操作.

而另一种情景,当 usb_stor_control_thread()正在执行命令,那么当然 usb_stor_release_resources()函数就不能释放资源了,甚至 storage_disconnect()也得等待,得等你当前这个命令执行完了,它才会去执行断开的代码.就好比你正在考场上战战兢兢的答题,老师却强行把你的试卷收上去,你说你会不会很愤怒?(当然,如果是因为你作弊被发现,那就另当别论了.)

总之,这种关系都是相互的,相互制约,同时也保证了整个系统正常运转,如果谁违规了,那么伤害的是大家的利益.这就是 Linux 内核的同步机制.

好了.我的故事讲完了.蓦然回首，发现，其实，我一直在寻觅，寻觅这个故事的结局，寻觅自己灵魂的出路，最终，追寻到了前者，却一直没有找到后者.