Linux 那些事儿
系列丛书
之
我是 Hub

卷首语: *Those who think themselves as being full of niubility and like to play zhuangbility merely reflect their shability.*

摘要:

2005 年 6 月,毕业典礼上,老师问我:此时此刻你有何感想?我回了一句:复旦害了我四年,我要用复旦学到的知识害社会一辈子!

关键词：*Linux, Kernel, 2.6.22.1, bus, usb, device driver, hub, 中断传输, port, usb 设备树, host controller, power management, 徐静蕾, 董卿*

故事梗概:在中关村买了一块外置声卡,路遇卖A片的女子,一番对话后买了A片回去看,初衷是借A片以测试声卡,不料却发现声卡居然指示灯不亮,推测是声卡所接的Hub的驱动程序的问题,百般无奈之下看了一下 Linux 内核中 2.6.22.1 的 Hub 驱动程序的源代码,结果却发现问题并不在 Hub 驱动中,失意之余遂行文记录这段读代码的感受,虽然自己心情郁闷,但是写下这些东西来或许对其他在 Linux 下看 A 片的人有一点点的参考价值吧,仅此而已.

目录

引子

天有不测风云,人有旦夕祸福.在 2007 年的夏天,在全国人民欢天喜地的迎接北京奥运会倒计时一周年之际,我那可爱的电脑声卡却很不和谐的坏了.

朋友给我推荐了一款飞利浦的外置声卡,PSC805.老实说,声卡还用外置的的确是让我觉得新鲜,直接用 USB 连接.价钱也还可以.所以我去了一趟中关村买了一块.如果买完了我直接坐上公交车回家那么故事就此将结束.

问题是,我是骑自行车去的.所以我接下来骑车回家.沿着中关村大街往南骑,很快我就遇上了传说中的卖 A 片的,老实说,以前我只是在网上看过一些关于在中关村卖 A 的人的笑话,并没有真正见识过,而这次也只是我来北京半年后第一次去中关村.虽说走南闯北这么多年,这次经历仍然是一次令我瞠目结舌的.

首先我所惊讶的是,这个女子竟然和我年龄相仿,而且还戴一幅眼镜,她也骑一辆自行车,车前也和我一样有一个筐,只不过我的筐里装的是一个盒子,里面装的是一块声卡,她的车筐里装的是一个不透明的袋子,里面装的什么东西是不言自明的.我从北向南骑,她从南向北骑,在我们的车子即将面对面的一刹那,我们之间的对话开始了:

她: 光盘要吗?
我: (先是一愣,后是一惊,然后是很冷静很沉着的问) 都有什么?
她: 你要 VCD 还是 DVD?
我: DVD 吧.
她从袋子里掏出三张来: 三张 30 块钱,都是正版的.
我拿过来一看,看见第一张上面有一个名字,叫做快活林,我忍住没笑.但我注意到,光盘上面只有那个标题和一些不堪入目的图片,但是却没有剧情介绍,于是我问: 有情节吗?
她斩钉截铁的说: 当然有.
我: 那为何没有剧情介绍?
她语气异常坚定的说: 你放心,不需要剧情介绍,保证是你喜欢的.
我差点吐了,"保证是你喜欢的"这句话太让我无语了.
老实说,没有剧情介绍的片子我是不可能买的,我摇了摇头,叹口气:没有剧情介绍就很难说了.
她很自信的说:这个你不用担心,我又不是只卖你一个人,我告诉你吧,我在中关村这里已经卖了两年了,基本上我都有比较稳定的客源,这附近,你像北大,清华,人大,他们的学生都经常到我这里买,你不信你可以去校园里问问.

听到这话我就觉得不对劲,我感觉她这话很不可信,以我在复旦四年的经历,我觉得现如今的大学生要看这些片子根本不需要来买光盘,校内资源异常丰富,想当年我们复旦校内有一个 ftp 知名站点叫做每日一毛,号称是每天更新一部精彩的毛片.所以在复旦念书期间我们根本不需要去外面买光盘,校内都有下.我觉得清华北大人大这些精英云集的兄弟学校怎么说也不会比我们那边的资源少.如果真的要来买盘,那也一定是那些女博士们,既想看,又不懂校内下载的那些工具,还不好意思问男生.

我正思考着呢,她又说了:我看你也像一个刚毕业的大学生啊,哪个大学的?

要不怎么说我是久经考验的共产主义接班人呢,这一刻,如果换了别人,估计首先会是一愣,然后会犹豫一阵子,然而,我没有任何异常表现,也斩钉截铁的来了一句:海淀走读大学!

最让我匪夷所思的事情这时候发生了,她面露喜色,说:今天这盘你买定了,我们是校友.

我发誓当时我真想抽自己一嘴巴,可是我毕竟还是见多识广,岂能轻易相信她的鬼话,我猜我要是说自己是清华的她是不是也会说自己是清华的呢.

可是她连让我怀疑的机会都没有给我,她从自己钱包里掏出一张足以让我闭嘴的小本,海淀走读大学学生证…

都说本命年会很倒霉,但是像这样的事情我确实无话可说,我认了,我怕继续下去她会叫我师兄,会追问我是哪个系毕业的,于是我匆匆忙忙给了钱,拿了光盘就要骑车走,她却还叫住我,说:校友啊,还没找钱呢,要不这样吧,我再送你两张,这可是武藤兰 2007 年拍得最精彩的几集,最近卖的超级火,原价 15,我 10 块给你得了,谁叫咱们是校友呢…

匆匆离去之后,我想今天这事也太邪了点,算了,正好今天买了声卡,看几部 A 片正好可以测试一下效果,我一直觉得,测试显卡可以用游戏,而测试声卡最好还是用 A 片,声卡好不好看一部精彩的 A 片就知道了.何况,武藤兰的片子我毕业之后就没有看过了,我只想知道这个日本 AV 女优的天后级人物这两年是否又有新的突破.而且我记得去年早些时候坊间曾经谣传武藤兰去世的消息,虽说这是假消息,但当时传的沸沸扬扬,想必对她也有一定影响吧,真的想知道这次她的新片会不会给人新的惊喜.

然而,人世间不如意之事十之八九.在店家那里好好的声卡回来之后居然连指示灯都不亮,根本没法用.不是完全不亮,一开始会亮,然后就不亮了.凭一种男人的直觉,我判定这是软件的问题,而且我用的 Linux 操作系统,从我用这个操作系统的第一天我就发现到处都有问题,而且干什么都特麻烦,也算是造了孽了,要不是工作得在 Linux 上进行,我才不会学什么 Linux 呢.

从指示灯这个现象来看,我估计是电源管理部分的问题,我听说 Linux 内核 2.6.20 左右在 USB 部分开始加入了电源管理的代码,我觉得这部分代码不够成熟,问题多多也是很正常的,只是没想到我成了试验品.

算了,不看 A 片了,但是我很理智的否定了这一幼稚的想法.作为一名名牌大学毕业的学生,如果遇到这点困难就放弃了,那么我对得起复旦四年的教育吗?况且不看 A 片事小,声卡不能用事大,总不能白买了一块声卡吧.想到卖 A 片的那个女生,想到她那种从容,那种乐观,那种淡定,那种理直气壮那种义正言辞那种斩钉截铁那种正气凛然,我心想,如果不能搞定这个问题,那么我枉为复旦人矣.

我很冷静地分析问题,首先这块声卡是使用 USB 接口的,供电有问题那么应该是 USB 驱动部分的问题而不应该是声卡驱动的问题,声卡驱动是 snd-usb-audio,察看了一下日志文件,实际上问题出现在这个模块被加载之前,所以可以排除声卡驱动的问题.然后我觉得问题可能出现在 Linux 中 Hub 驱动的部分,也可能出现在主机控制器驱动的部分.这下子问题稍微麻烦了点,我完全不清楚究竟应该分析谁,于是我做了一次选择,我猜测问题会在 Hub 驱动那边,看了一下 Hub 驱动也就是三千多行代码,看了看时间,一个晚上应该是能看完的,如果效率高的话把问

题解决了还有时间看片.狠一狠心,真就看了起来.

引子写到这也就该结束了,大多数人也许都会觉得我最后一定是通过看代码解决了问题,然后才会写下下面的这些文字,实际上不是的,我花了一夜看完了 Hub 驱动的代码,然而并没有发现任何异常,后来我终于知道,这个问题并不是出在Hub驱动的部分,它实际上与UHCI主机控制器的驱动代码有关,算是UHCI驱动的一个Bug.但是既然我看了Hub驱动的代码,也不妨用文字把它记录下来,就算是为了纪念这样一个夜晚吧.

跟我走吧,现在就出发

最早知道 hub 是在大学里,复旦的 4 人间宿舍,条件真好,每个人一张书桌,书桌下面一个网口,但是有时候网口坏了,那可急死人了,要知道当初我们买电脑初衷虽说是为了学习 C 语言,可是买了之后,C 倒是没学,先学了 CS.printf 还没学会呢,倒是先学会了怎么在 CS 里喊 go go go, fire in the hole!网口坏了就意味着 CS 不能玩了,当时对人生真的很绝望,后来有人介绍,说 6 楼几个哥们也有这种情况,然后人家买了 hub.所以网口坏了也没事.当时我这个兴奋啊!比过了六级还要high...

事情过了几年,今天我再次回过头来看hub,不过这里看的是usb中的hub.但是很显然,和我们在以太网用的 hub 不是一回事,但是角色是一样的,都说红花需要绿叶配,在 usb 的世界里,hub 永远都只是绿叶,她不可能是红花,她的存在只是为了支持更多的设备连接到 usb 总线上来,谁也不会为了使用 hub 而购买 hub,买 hub 的原因是为了要使用别的设备,而 hub 就像嫁妆,仅此而已.就像当年我们买 hub 是为了使用网卡,而不是 hub 本身.

也许设计代码的人和我一样,希望大家能够更多的给 hub 一点关注,所以,关于 hub 的代码在 usb core 的目录下面.

这个故事中使用的是 2.6.22.1 的内核代码,这是此刻我能从 kernel.org 上面获得的最新的内核版本.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,在这个目录中的 usb 子目录包含了所有 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-2.6.22.1/drivers # cd usb/

localhost:/usr/src/linux-2.6.22.1/drivers/usb # ls

Kconfig Makefile README atm class core gadget host image misc mon 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.古训有云,再牛 B 的肖邦也弹不出老子的忧伤,我想说的是,再牛 B 的黑客如果不看 Makefile 不看 Kconfig,也别想搞清楚这里的结构.很多年轻人喜欢研究 usb-skeleton.c,据说这个文件对他们很有启发,所以这里偶也友情推荐一下这个文件.有时间有兴趣的可以看一看,其实就是一个简单的 usb 设备驱动程序的框架.

关于 Makefile,在当年我们讲 usb storage 的时候已经讲了一些,这里就不重复了,反反复复讲 Makefile 是不是太没品了点?要是我室友知道了,肯定会问,是不是男人?不过一个很显然的事实摆在这里,在这众多的目录中,我们一眼就能看到有一个叫做 core 的目录,虽然四六级都是刚刚擦

边过,可是 core 这个词咱们可没少用.当年,和这个词一起走红的另外两个英语单词是 simple,naïve.

好,我们来看 core 目录.关于 usb,有一个很重要的模块,她的名字耐人追寻,usbcore.如果你的电脑是装了 Linux,那么你用 lsmod 命令看一下,有一个模块叫做 usbcore.当然,你要是玩嵌入式系统的高手,那么也许你的电脑里没有 usb 模块,不过我听说如今玩嵌入式的人也喜欢玩 usb 的,因为 usb 设备很合嵌入式的口味.看看我的 lsmod 的输出吧,

```
localhost:/usr/src/linux-2.6.22.1/drivers/usb/core # lsmod
Module                  Size  Used by
af_packet              55820  2
raw                    89504  0
nfs                   230840  2
lockd                  87536  2 nfs
nfs_acl                20352  1 nfs
sunrpc                172360  4 nfs,lockd,nfs_acl
ipv6                  329728  36
button                 24224  0
battery                27272  0
ac                     22152  0
apparmor               73760  0
aamatch_pcre           30720  1 apparmor
loop                   32784  0
usbhid                 60832  0
dm_mod                 77232  0
ide_cd                 57120  0
hw_random              22440  0
ehci_hcd               47624  0
cdrom                  52392  1 ide_cd
uhci_hcd               48544  0
shpchp                 61984  0
bnx2                  157296  0
usbcore               149288  4 usbhid,ehci_hcd,uhci_hcd
e1000                 130872  0
pci_hotplug            44800  1 shpchp
reiserfs              239616  2
edd                    26760  0
fan                    21896  0
thermal                32272  0
processor              50280  1 thermal
qla2xxx               149484  0
firmware_class         27904  1 qla2xxx
scsi_transport_fc      51460  1 qla2xxx
sg                     52136  0
megaraid_sas           47928  3
```

```
piix                   27652  0 [permanent]
sd_mod                   34176  4
scsi_mod                                                163760         5
qla2xxx,scsi_transport_fc,sg,megaraid_sas,sd_mod
ide_disk                 32768  0
ide_core                164996  3 ide_cd,piix,ide_disk
```

找到了 usbcore 那一行吗?core 就是核心,基本上你要在你的电脑里用 usb 设备,那么两个模块是必须的,一个是 usbcore,这就是核心模块,另一个是主机控制器的驱动程序,比如这里 usbcore 那一行我们看到的 ehci_hcd 和 uhci_hcd,这里偶的是 Dell2950 的服务器,里边有两个 usb host controller.一个是 echi 的,三个是 uhci 的.你问我什么是 ehci?什么是 ohci?就是 host controller 的接口.从硬件上来说,usb 设备要想工作,除了外设本身,必须还有一个咚咚叫做 usb host controller.一般来说,一个电脑里有一个 usb host controller 就可以了,她就可以控制很多个设备了,比如 u 盘,比如 usb 键盘,比如 usb 鼠标.所有的外设都把自己的请求提交给 usb host controller.然后让 usb host controller 统一来调度.而设备怎么连到 host controller 上?哎,这就是我们故事的主角,hub,乳名叫做集线器.

关于 hub 的代码,在 drivers/usb/core/目录下面,有一个叫做 hub.c 的文件.这个文件可不简单.

localhost:/usr/src/linux-2.6.22.1/drivers/usb/core # wc -l hub.c

3122 hub.c

傻眼了,就 hub 这么一个玩艺儿竟然有三千多行代码,真要是按行计费,写代码的那些家伙还不发财发死?还好不是那样,真要是那样,还不个个都跟赵丽华似的,一个个都成了诗人,开源社区该改成开源诗社了.

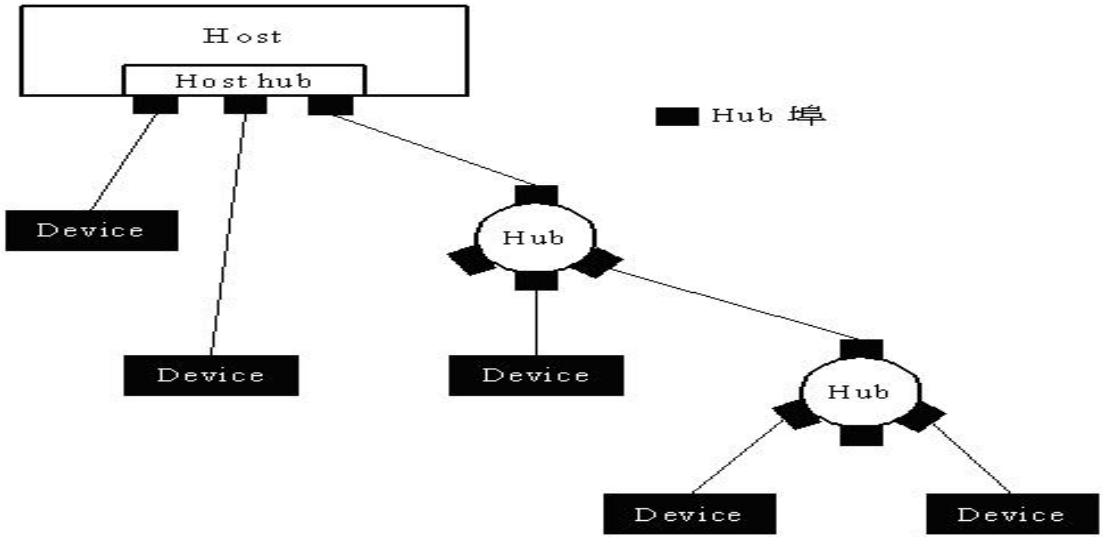
三千多行就三千多行吧,总不能见困难就退吧.小的时候我们可都是听着雷锋叔叔的故事长大的,雷锋叔叔的螺丝钉的精神虽然我们学不到,但是雷锋叔叔说的好,对待女同志,要像春天般的温暖!同样,我们对待代码,也要像春天般的温暖.嗯,跟我走吧,现在就出发.

特别的爱给特别的 Root Hub

莎士比亚曾经说过,不懂 hub 是怎么工作的就等于不知道 usb 设备驱动是怎么工作的.这句话一点没错,因为 usb 设备的初始化都是 hub 这边发起的,通常我们写 usb 设备驱动程序都是在已经得到了一个 struct usb_interface 指针的情况下开始 probe 工作,可是我要问你,你的 struct usb_interface 从哪来的?你以为你的设备天生丽质?长得比较帅?一插入 usb 口就有了?如果有人说是的,那么我只能说,相信这个人的嘴,不如相信世上有鬼!老实说,要想知道从 usb 设备插入 usb 口的那一刻开始,这个世界发生了什么,你必须知道 hub 是怎么工作的,Linux 中 hub 驱动程序是怎么工作的.

要说 usb hub,那得从 Root hub 说起.什么是 Root hub?冤有头债有主,不管你的电脑里连了多少个 usb 设备,她们最终是有根的,她们不会像孙猴那样从石头里蹦出来的,不会像北京电视台纸包子的新闻凭空就可以造出来.所有的 usb 设备最终都是连接到了一个叫做 Root Hub 的冬冬上,或者说所有的根源都是从这里开始的.Root Hub 上可以连接别的设备,可以连接 U 盘,可以连接 usb 鼠标,同样也可以连接另一个 hub.所谓 hub,就是用来级连.但是普通的 hub,她一头接上级 hub, 另一头可以有多个口,多个口就可以级连多个设备,也可以只有一个口,一个口的就像我们

宿舍里常用的那种延长线.而 Root Hub 呢?她比较特殊,她当然也是一种 usb 设备,但是她属于一人之下万人之上的角色,她只属于被 Host Controller,换言之,通常做芯片的同志们会把 Host Controller 和 Root Hub 集成在一起.特别是 PC 主机上,通常你就只能看到接口,看不到 Root Hub,因为她躺在 Host Controller 的怀里.这一对狗男女躲在机箱里面,你当然看不到.下面这张图看得很清楚,



其中的那个 host hub 其实就是 root hub.当然,我们应该更加准备的评价 host 和 root hub 的关系,她们是双生花,不会残酷地吸纳彼此的幸福,只会一直不离不弃地在风中相依,直到凋零...(虽然言论自由,但是我必须声明,我绝对没有影射最近牺牲在车里的徐新贤书记以及那个和他一起裸死在车里的妇联主席潘丽航的意思,请不要误会,我个人还是很欣赏这种死了都要爱的境界的,这种追求真爱,追求自由的思想正是整个开源社区所推崇的,也正是开源精神的灵魂所在,我顶!)

Ok,别扯远了,继续回来,既然 root hub 享有如此特殊的地位,那么很显然,整个 usb 子系统得特别的爱给特别的你,一开始就会要初始化 hub.所以我们从 usb 子系统的初始化代码开始看起.也就是 usb 的核心代码.来自 drivers/usb/core/usb.c:

```
938 subsys_initcall(usb_init);
939 module_exit(usb_exit);
```

很显然,这样两行正是 Linux 中 usb 子系统的初始化代码,这里我们看到一个 subsys_initcall,它也是一个宏,我们可以把它理解为 module_init,只不过因为这部分代码比较核心,开发者们把它看作一个子系统,而不仅仅是一个模块,这也很好理解,usbcore 这个模块它代表的不是某一个设备,而是所有 usb 设备赖以生存的模块,Linux 中,像这样一个类别的设备驱动被归结为一个子系统.比如 pci 子系统,比如 scsi 子系统,基本上,drivers/目录下面第一层的每个目录都算一个子系统,因为它们代表了一类设备.subsys_initcall(usb_init)的意思就是告诉我们 usb_init 是我们真正的初始化函数,而 usb_exit()将是整个 usb 子系统的结束时的清理函数.于是我们就从 usb_init 开始看起,

```
860 /*
861  * Init
862  */
863 static int __init usb_init(void)
```

```
864 {
865         int retval;
866         if (nousb) {
867                 pr_info("%s: USB support disabled\n", usbcore_name);
868                 return 0;
869         }
870
871         retval = ksuspend_usb_init();
872         if (retval)
873                 goto out;
874         retval = bus_register(&usb_bus_type);
875         if (retval)
876                 goto bus_register_failed;
877         retval = usb_host_init();
878         if (retval)
879                 goto host_init_failed;
880         retval = usb_major_init();
881         if (retval)
882                 goto major_init_failed;
883         retval = usb_register(&usbfs_driver);
884         if (retval)
885                 goto driver_register_failed;
886         retval = usb_devio_init();
887         if (retval)
888                 goto usb_devio_init_failed;
889         retval = usbfs_init();
890         if (retval)
891                 goto fs_init_failed;
892         retval = usb_hub_init();
893         if (retval)
894                 goto hub_init_failed;
895                  retval  =  usb_register_device_driver(&usb_generic_driver,
THIS_MODULE);
896         if (!retval)
897                 goto out;
898
899         usb_hub_cleanup();
    900 hub_init_failed:
901         usbfs_cleanup();
902 fs_init_failed:
903         usb_devio_cleanup();
904 usb_devio_init_failed:
905         usb_deregister(&usbfs_driver);
906 driver_register_failed:
```

```
907         usb_major_cleanup();
908 major_init_failed:
909         usb_host_cleanup();
910 host_init_failed:
911         bus_unregister(&usb_bus_type);
912 bus_register_failed:
913         ksuspend_usb_cleanup();
914 out:
915         return retval;
916 }
```

好家伙,一堆的函数,每一个都够我看的了.写代码的无非就是想吓唬吓唬我,共产党员死都不怕,这点小事还能吓住我们吗? 我们不用仔细看,扫一眼就能发现,许多名叫*init*的函数,很显然,就是做一些初始化.其中我们要关注的是 hub,它也是在这里初始化的,换言之,随着 usb core 的初始化,hub 也就开始了她的初始化之路.这就是 1490 行,usb_hub_init()函数得到调用.这个函数来自 drivers/usb/core/hub.c:

```
2854 int usb_hub_init(void)
2855 {
2856         if (usb_register(&hub_driver) < 0) {
2857                 printk(KERN_ERR "%s: can't register hub driver\n",
2858                         usbcore_name);
2859                 return -1;
2860         }
2861
2862         khubd_task = kthread_run(hub_thread, NULL, "khubd");
2863         if (!IS_ERR(khubd_task))
2864                 return 0;
2865
2866         /* Fall through if kernel_thread failed */
2867         usb_deregister(&hub_driver);
2868         printk(KERN_ERR "%s: can't start khubd\n", usbcore_name);
2869
2870         return -1;
2871 }
```

一路走来的兄弟们不会对这样一段代码陌生,是不是有一种似曾相识燕归来的感觉?没有?没有就没有吧,同是天涯堕落人,相逢何必曾相识.

最后,补充一条背景知识介绍,本年度感动中国新闻之一:

杭州信息网最新消息:临安市清凉峰镇党委书记,优秀共产党员徐新贤日前因公牺牲.徐新贤书记因忙于公务,日夜操劳过度,终因体力不支,牺牲在小车上,同时牺牲的还有该镇优秀妇女干部一名.据悉,徐新贤书记和该名妇女干部在小汽车里谈论公务,因汽油燃尽,空调断电,车内闷热严重,车子又是停在车库里,二人为了解热,脱光了衣服裤子,还在坚持谈公事,一直谈到牺牲为止.有这样

认真负责的好书记,清凉峰镇的老百姓是幸福的.书记时刻为人民办事,牺牲前还在亲自做妇女工作,他的事迹,人们都记在了心里.

一样的精灵不一样的 API

usb_register()和 usb_deregister()这两个函数我们当初分析 usb storage 的时候就已经见到过了.当时我们就说过了,这个函数是用来向 usb 核心层,即 usb core,注册一个 usb 设备驱动的.那年我们注册了一个 struct usb_driver usb_storage_driver.而这里我们注册的是 hub 的驱动程序所对应的 struct usb_driver 结构体变量.定义于 drivers/usb/core/hub.c 中:

```
2841 static struct usb_driver hub_driver = {
2842         .name =          "hub",
2843         .probe =         hub_probe,
2844         .disconnect =    hub_disconnect,
2845         .suspend =       hub_suspend,
2846         .resume =        hub_resume,
2847         .pre_reset =     hub_pre_reset,
2848         .post_reset =    hub_post_reset,
2849         .ioctl =         hub_ioctl,
2850         .id_table =      hub_id_table,
2851         .supports_autosuspend = 1,
2852 };
```

和咱们 storage 那个情况一样,最重要的一个函数就是 hub_probe,对应咱们那里的 storage_probe().很多事情都在这期间发生了.不过有一点你必须明白,当初 storage_probe()被调用是发生在 usb-storage 模块被加载了并且检测到了有设备插入之后的情况下,也就是说有两个前提,第一个 usb-storage 被加载了,第二个设备插入了被检测到了,于是 storage_probe()被调用.而 hub,说她特别,我可绝不是忽悠你.hub 本身就是两种,一种是普通的 hub,一种是 root hub.对于普通 hub,它完全可能也是和 U 盘一样,在某个时刻被你插入,然后这种情况下 hub_probe 被调用,但是对于 root hub 就不需要这么多废话了,root hub 肯定是有的,只要你有 host controller,就一定会有 root hub,所以 hub_probe()基本上是很自然的就被调用了,不用说非得等待某个插入事件的发生,没这个必要.当然如果你要抬杠,你说那如果没有 usb host controller 那就不用初始化 root hub 了,这简直是废话,没有 usb host controller 就没有 usb 设备能工作.那么 usb core 这整个模块你就没有必要分析了.所以,只要你有 usb host controller,那么在 usb host controller 的驱动程序初始化的过程中,它就会调用 hub_probe()来探测 root hub,不管你的 host controller 是 ohci,uhci,还是 ehci 的接口.别急,慢慢来.

如果 register 一切顺利的话,那么返回 0.要不返回负数,返回负数就说明出错了.如果这一步都能出错,那我只能说你运气真的很好,中国队要有你这样的运气,早就拿世界杯冠军了,哪能像现在这样和缅甸进行生死战.Ok,假设这一步没有 fail,如果真的 fail 你也不用急,Linus 肯定比你急,Greg 更急,恐怕这位 Linux 中 pci/usb 的维护者一世英名就被这样你毁了.

Ok,2862 行,这行代码其实是很有技术含量的.不过对写驱动的人来说,其作用就和我们当年那个 kernel_thread()是的,或者如果你忘记了我们曾经讲过的创建 usb-storage 精灵进程的 kernel_thread(),那么你就把这个当作 fork 吧.世界在变,Linux 内核代码当然也在变,小时候我

们看周润发赵雅芝演的上海滩,长大了我们看孙俪黄晓明演的上海滩.都是上海滩,只是版本不同,只是内容不同.内核 kthread_run()就扮演着 2.6.10 内核那个 kernel_thread()的作用.不过当时 kernel_thread()返回值是一个 int 型的,而 kthread_run()返回的却是 struct task_struct 结构体指针.这里等号左边的 khubd_task 是我们自己定义的一个 struct task_struct 指针.

88 static struct task_struct *khubd_task;

struct task_struct 不用多说,记录进程的数据结构.每一个进程都用一个 struct task_struct 结构体变量来表示.所以这里所做的就是记录下创建好的内核进程,以便日后要卸载模块的时候可以用另一个函数来结束这个内核进程(你也可以叫内核线程,看自己喜欢啰,又没有人逼着我说哪一个.),到时候我们会调用 kthread_stop(khubd_task)来结束这个内核线程,这个函数的调用我们将会在 usb_hub_cleanup()函数里看到.而 usb_hub_cleanup()正是 usb hub 里面和 usb_hub_init()相对应的函数.

2863 行,判断一下 khubd_task,IS_ERR 是一个宏,用来判断指针的.当你创建了一个进程,你当然想知道这个进程创建成功了没有.以前我们注意到每次申请内存的时候都会做一次判断,你说创建进程是不是也要申请内存?不申请内存谁来记录 struct task_struct?很显然,要判断.以前我们判断的是指针是否为空,但是 Linux 内核不是上海滩,越老越经典.以后接触代码多了你会发现,我们以前在 usb storage 申请内存的时候用的都是 kmalloc(),但是其实 Linux 内核中有很多种内存申请的方式,而这些方式所返回的内存地址也是不一样的,所以并不是每一次我们都只要判断指针是否为空就可以了.事实上,每一次调用 kthread_run()之后,我们都会用一个 IS_ERR()来判断指针是否有效.IS_ERR()为 1 就表示指针有错,或者准确一点说叫做指针无效.什么叫指针无效?如果你和我一样,觉得生活很无聊,那么本节最后一段会专门给你个解释,其他人就不用去管 IS_ERR()了,让我们继续往下看,只需要记得,如果你不希望发生缺页异常这样的错误的话,那么请记住,每次调用完 kthread_run()之后要用 IS_ERR()来检测一下返回的指针.如果 IS_ERR()返回值是 0,那么说明没有问题,于是 return 0,也就是说 usb_hub_init()就这么结束了.反之,就会执行 usb_deregister(),因为内核线程没有成功创建,hub 就没法驱动起来了.于是就没有必要瞎耽误工夫了,该干嘛干嘛去.最后函数在 2870 行,返回-1.回到 usb_init()中我们会知道,接下来 usb_hub_cleanup()就会被调用.usb_hub_cleanup()同样定义于 drivers/usb/core/hub.c 中,

```
2873 void usb_hub_cleanup(void)
2874 {
2875         kthread_stop(khubd_task);
2876
2877         /*
2878          * Hub resources are freed for us by usb_deregister. It calls
2879          * usb_driver_purge on every device which in turn calls that
2880          * devices disconnect function if it is using this driver.
2881          * The hub_disconnect function takes care of releasing the
2882          * individual hub resources. -greg
2883          */
2884         usb_deregister(&hub_driver);
2885 } /* usb_hub_cleanup() */
```

这个函数我想没有任何必要解释了吧.kthread_stop()和刚才的 kthread_run()对应,usb_deregister()和 usb_register()对应.

总之,如果创建子进程出了问题,那么一切都免谈.啥也别玩了.歇菜了.

反之,如果成功了,那么 kthread_run()的三个参数就是我们要关注的了,第一个 hub_thread(),子进程将从这里开始执行.第二个是 hub_thread()的参数,传递的是 NULL,第三个参数就是精灵进程的名字,你 ps –el 看一下,比如像偶的这样子:

```
localhost:/usr/src/linux-2.6.22.1/drivers/usb/core # ps -el | grep khubd

1 S     0  1963    27  0  70  -5 -     0 hub_th ?        00:00:00 khubd
```

你就会发现有这么一个精灵进程运行着.所以,下一步,让我们进入 hub_thread()来看看这个子进程吧,很显然,关于父进程,我们没什么好看的了.

==========华丽的分割线==========

人的无聊,有时候很难用语言表达.以下关于 IS_ERR 的文字仅献给无聊的你.如果你对内存管理没有任何兴趣,就不用往下看了,跳到下一节吧.要想明白 IS_ERR(),首先你得知道有一种空间叫做内核空间,不清楚也不要紧,我也不是很清楚,曾经,在复旦,上操作系统这门课的时候,我一度以为我已经成为天使了,因为我天天上课都在听天书.后来,确切地说是去年,我去微软全球技术中心(GSTC)面试的时候,那个 manager 就要我解释这个名词,要我谈一谈对内核空间和用户空间的理解,其实我也挺纳闷的,我只不过是希望能成为微软的一名技术支持工程师,居然还要懂内核,你说这是什么世道?中学时候,老师不是跟我说只要学好数理化,走遍天下都不怕吗?算了,不去想这些伤心往事了.结合 IS_ERR()的代码来看,来自 include/linux/err.h:

```
8 /*
9  * Kernel pointers have redundant information, so we can use a
10  * scheme where we can return either an error code or a dentry
11  * pointer with the same return value.
12  *
13  * This should be a per-architecture thing, to allow different
14  * error and pointer decisions.
15  */
16 #define MAX_ERRNO       4095
17
18 #ifndef __ASSEMBLY__
19
20 #define IS_ERR_VALUE(x) unlikely((x) >= (unsigned long)-MAX_ERRNO)
21
22 static inline void *ERR_PTR(long error)
23 {
24         return (void *) error;
25 }
```

```
26
27 static inline long PTR_ERR(const void *ptr)
28 {
29          return (long) ptr;
30 }
31
32 static inline long IS_ERR(const void *ptr)
33 {
34          return IS_ERR_VALUE((unsigned long)ptr);
35 }
36
37 #endif
```

关于内核空间,我只想说,所有的驱动程序都是运行在内核空间,内核空间虽然很大,但总是有限的.要知道即便是我们这个幅员辽阔的伟大祖国其空间也是有限的,也只有 960 万平方公里,所以内核空间当然也是一个有限的空间,而在这有限的空间中,其最后一个 page 是专门保留的,也就是说一般人不可能用到内核空间最后一个 page 的指针.换句话说,你在写设备驱动程序的过程中,涉及到的任何一个指针,必然有三种情况,一种是有效指针,一种是 NULL,空指针,一种是错误指针,或者说无效指针.而所谓的错误指针就是指其已经到达了最后一个page.比如对于32bit的系统来说,内核空间最高地址 0xffffffff,那么最后一个 page 就是指的 0xfffff000~0xffffffff(假设4k一个page).这段地址是被保留的,一般人不得越雷池半步,如果你发现你的一个指针指向这个范围中的某个地址,那么恭喜你,你的代码肯定出错了.

那么你是不是很好奇,好端端的内核空间干嘛要留出最后一个 page?这不是缺心眼儿吗?明明自己有 1000 块钱,非得对自己说只能用 900 块.实在不好意思,你说错了,这里不仅不是浪费一个page,反而是充分利用资源,把一个东西当两个东西来用.

看见 16 行那个 MAX_ERRNO 了吗?一个宏,定义为 4095,MAX_ERRNO 就是最大错误号,Linux 内核中,出错有多种可能,因为有许许多多种错误,就像一个人进监狱,可能是像迟志强那样,在事业如日中天的时候强奸女孩,可能是像张君大哥那样,因为抢劫长沙友谊商城后又抢农业银行,亦或者是马加爵大侠那样,受同学的气,最终让铁锤来说话.关于 Linux 内核中的错误,我们看一下 include/asm-generic/errno-base.h 文件:

```
#define EPERM            1      /* Operation not permitted */
#define ENOENT           2      /* No such file or directory */
#define ESRCH            3      /* No such process */
#define EINTR            4      /* Interrupted system call */
#define EIO              5      /* I/O error */
#define ENXIO            6      /* No such device or address */
#define E2BIG            7      /* Argument list too long */
#define ENOEXEC          8      /* Exec format error */
#define EBADF            9      /* Bad file number */
#define ECHILD           10     /* No child processes */
#define EAGAIN           11     /* Try again */
#define ENOMEM           12     /* Out of memory */
```

```
#define EACCES          13      /* Permission denied */
#define EFAULT          14      /* Bad address */
#define ENOTBLK         15      /* Block device required */
#define EBUSY           16      /* Device or resource busy */
#define EEXIST          17      /* File exists */
#define EXDEV           18      /* Cross-device link */
#define ENODEV          19      /* No such device */
#define ENOTDIR         20      /* Not a directory */
#define EISDIR          21      /* Is a directory */
#define EINVAL          22      /* Invalid argument */
#define ENFILE          23      /* File table overflow */
#define EMFILE          24      /* Too many open files */
#define ENOTTY          25      /* Not a typewriter */
#define ETXTBSY         26      /* Text file busy */
#define EFBIG           27      /* File too large */
#define ENOSPC          28      /* No space left on device */
#define ESPIPE          29      /* Illegal seek */
#define EROFS           30      /* Read-only file system */
#define EMLINK          31      /* Too many links */
#define EPIPE           32      /* Broken pipe */
#define EDOM            33      /* Math argument out of domain of func */
#define ERANGE          34      /* Math result not representable */
```

最常见的几个是-EBUSY,-EINVAL,-ENODEV,-EPIPE,-EAGAIN,-ENOMEM,我相信不用说你写过代码调试过代码,只要你使用过 Linux 就有可能见过这几个错误,因为它们确实经常出现.这些是每个体系结构里都有的,另外各个体系结构也都定义了自己的一些错误代码.这些东西当然也都是宏,实际上对应的是一些数字,这个数字就叫做错误号.而对于Linux 内核来说,不管任何体系结构,最多最多,错误号不会超过 4095.而 4095 又正好是比 4k 小 1,即 4096 减 1.而我们知道一个 page 可能是 4k,也可能是更多,比如 8k,但至少它也是 4k,所以留出一个 page 出来就可以让我们把内核空间的指针来记录错误了.什么意思呢?比如我们这里的 IS_ERR(),它就是判断 kthread_run()返回的指针是否有错,如果指针并不是指向最后一个 page,那么没有问题,申请成功了,如果指针指向了最后一个 page,那么说明实际上这不是一个有效的指针,这个指针里保存的实际上是一种错误代码.而通常很常用的方法就是先用IS_ERR()来判断是否是错误,然后如果是,那么就调用 PTR_ERR()来返回这个错误代码.只不过咱们这里,没有调用 PTR_ERR()而已,因为起决定作用的还是 IS_ERR(),而 PTR_ERR()只是返回错误代码,也就是提供一个信息给调用者,如果你只需要知道是否出错,而不在乎因为什么而出错,那你当然不用调用 PTR_ERR()了,毕竟,男人,简单就好.当然,这里如果出错了的话,最终 usb_deregister()会被调用,并且 usb_hub_init()会返回-1.

那些队列,那些队列操作函数

这一节我们讲队列.

从前在乡下的时候是不用排队的,村里的人们都很谦让,而且人本来又不多.后来到了县城里,县城里不大,大家去走亲戚去串门去逛街不用坐车不用排队,除了街上的游戏厅人多一点以外,别的地方人都不是很多,陪妈妈去菜市场买菜也不用排队.后来到了上海,发现去食堂吃饭要排队,开学报道要排队,在人民广场等回复旦的 123 路公共汽车要排队,考试成绩不好去教务处交重修费要排队,甚至连追求一个女孩子也要排队.每次看见人群排成一条长龙时,才真正意识到自己是龙的传人.

随着子进程进入了我们的视野,我们来看其入口函数,hub_thread().这是一个令你大跌隐形眼镜的函数.

```
2817 static int hub_thread(void *__unused)
2818 {
2819         do {
2820                 hub_events();
2821                 wait_event_interruptible(khubd_wait,
2822                                 !list_empty(&hub_event_list) ||
2823                                 kthread_should_stop());
2824                 try_to_freeze();
2825         } while (!kthread_should_stop() || !list_empty(&hub_event_list));
2826
2827         pr_debug("%s: khubd exiting\n", usbcore_name);
2828         return 0;
2829 }
```

这就是 hub 驱动中最精华的代码.这几乎是一个死循环,但是关于 hub 的所有故事都发生在这里,没错,就在这短短几行代码中.

而这其中,最核心的函数自然是 hub_events().我们先不看 hub_events,先把外面这几个函数看明白了,kthread_should_stop()的意思很明显,就是字面意思,是不是该停掉,如果是,那么这里循环就结束了,hub_thread()返回 0,而要让 kthread_should_stop()为真,就是当我们调用 kthread_stop()的时候.这种情况,这个进程就该结束了.

再看 hub_event_list,同样来自 drivers/usb/core/hub.c:

```
83 static LIST_HEAD(hub_event_list);        /* List of hubs needing servicing */
```

我们来看一下 LIST_HEAD 吧,当你越接近那些核心的代码,你就会发现关于链表的定义会越多.其实在 usb-storage 里面,我们也提到过一些链表,但却并没有自己用 LIST_HEAD 来定义过链表,因为我们用不着.可是 hub 这边就有用了,当然 host controller 的驱动程序里也会有,使用链表的目的很明确,因为有很多事情要做,于是就把它放进链表里,一件事一件事的处理,还记得我们当初在 usb-storage 里面提交 urb request 了吗?你 U 盘不停的提交 urb request,他 usb 键盘也提交,我 usb 鼠标也提交,那 usb host controller 咋应付得过来呢?很简单,建一个队列,然后你每次提交就是往一个队列里边插入,然后 usb host controller 再统一去调度,一个一个来执行.那么这里 hub,它有什么事件?比如探测到一个设备连进来了,于是就会执行一些代码去初始化设备,所以就建一个队列.所以,再一次证明了,谭浩强大哥的 C 程序设计是我们学习 Linux 的有

力武器,书中对链表的介绍无疑是英明的,谭大哥,您不是一个人在战斗!关于 Linux 内核中的链表,可以专门写一篇文章了,我们简单介绍一下,来看 include/linux/list.h,

```
21 struct list_head {
22         struct list_head *next, *prev;
23 };
24
25 #define LIST_HEAD_INIT(name) { &(name), &(name) }
26
27 #define LIST_HEAD(name) \
28         struct list_head name = LIST_HEAD_INIT(name)
```

可以看出,我们无非就是定义了一个 struct list_head 的结构体,hub_event_list,而且其两个指针 next 和 prev 分别是指向自己,换言之,我们建立了一个链表,而且是双向链表,并且做了初始化,初始化成一个空队列.而对于这个队列的操作,内核提供了很多函数可以使用,不过在 usb hub 中,我们将会用到这么几个函数,它们是,

```
294 /**
295  * list_empty - tests whether a list is empty
296  * @head: the list to test.
297  */
298 static inline int list_empty(const struct list_head *head)
299 {
300         return head->next == head;
    301 }
```

不言自明,判断队列是否为空,我们说了,初始化的时候这个 hub_event_list()队列是空的.

有了队列,自然就要操作队列,要往队列里加东西,减东西,就像我们每个人每天都在不停的走进去,又走出来,似是梦境又不是梦境.一切都是不经意的.走进去是一年四季,走出来是春夏秋冬.来看第二个函数,list_add_tail(),这就是往队列里加东西.

```
76 /**
77  * list_add_tail - add a new entry
78  * @new: new entry to be added
79  * @head: list head to add it before
80  *
81  * Insert a new entry before the specified head.
82  * This is useful for implementing queues.
83  */
84 static inline void list_add_tail(struct list_head *new, struct list_head *head)
85 {
86         __list_add(new, head->prev, head);
87 }
```

继续跟着__list_add()看就会发现其实就是往队列的末尾加一个元素.

```
43 static inline void __list_add(struct list_head *new,
44                               struct list_head *prev,
45                               struct list_head *next)
46 {
47         next->prev = new;
48         new->next = next;
49         new->prev = prev;
50         prev->next = new;
51 }
```

搞懂谭浩强那本书之后看这些链表的代码那就是小菜一碟.再来看下一个,list_del_init(),队列里的元素不能只加不减,没用了的元素就该删除掉,把空间腾出来给别人.郭敬明说过,我生命里的温暖就那么多,我全部给了你,但是你离开了我,你叫我以后怎么再对别人笑...

```
250 /**
251  * list_del_init - deletes entry from list and reinitialize it.
252  * @entry: the element to delete from the list.
253  */
254 static inline void list_del_init(struct list_head *entry)
255 {
256         __list_del(entry->prev, entry->next);
257         INIT_LIST_HEAD(entry);
    258 }
```

从队列里删除一个元素,并且将该元素做初始化,首先看__list_del(),

```
148 /*
149  * Delete a list entry by making the prev/next entries
150  * point to each other.
151  *
152  * This is only for internal list manipulation where we know
153  * the prev/next entries already!
154  */
155 static inline void __list_del(struct list_head * prev, struct list_head * next)
156 {
157         next->prev = prev;
158         prev->next = next;
159 }
```

INIT_LIST_HEAD()其实就是初始化为最初的状态,即一个空链表.如下:

```
30 static inline void INIT_LIST_HEAD(struct list_head *list)
31 {
```

```
32          list->next = list;
33          list->prev = list;
34 }
```

当然了,还有超级经典的一个,list_entry(),和以上所有 list 方面的宏一样,也是来自 include/linux/list.h:

```
419 /**
420  * list_entry - get the struct for this entry
421  * @ptr:        the &struct list_head pointer.
422  * @type:       the type of the struct this is embedded in.
423  * @member:     the name of the list_struct within the struct.
424  */
425 #define list_entry(ptr, type, member) \
    426         container_of(ptr, type, member)
```

我相信,list_entry()这个宏在 Linux 内核代码中的地位,就相当于广告词中的任静付笛生的洗洗更健康,相当于大美女关之琳的一分钟轻松做女人,这都是耳熟能详妇孺皆知的,是经典中的经典.如果你说你不知道 list_entry(),那你千万别跟人说你懂 Linux 内核,就好比你不知道陈文登不知道任汝芬你就根本不好意思跟人说你考过研,要知道每个考研人都是左手一本陈文登右手一本任汝芬.

可惜,关于 list_entry,这个谭浩强老师的书里就没有了,当然你不能指责谭浩强的书不行,再好的书也不可能包罗万象.这个宏应该说还是有一定技术含量的.我印象中外企面试笔试中,有三次比较经典的链表考题,其中对 list_entry 的解释就是其中一次,威盛的笔试,而另外两次是摩托罗拉中国研发中心和 IBM.摩托罗拉那次是在中信泰富广场,问了我一道骨灰级的问题,给定一个链表,如何判断链表中是否有环,而 IBM 那次也是问的一道化石级的问题,说给定一个单向链表,请设计一个比较快的算法来找到该链表中的倒数第 m 个元素.像我这种低智商+不学无术的人怎么可能回答得出这样的问题.也许只有错过 IBM 这个我当时最心仪的公司才能给我留下刻骨铭心的伤痛吧.每一段文字都有一段刻骨铭心的经历,人们能看见我在屏幕上打的字,却看不见我流在键盘上的泪.我记得当年有一个房地产公司来我们学校招聘,是瑞安房地产,当时我一边投简历一边对同学说,要就别让我就了瑞安,进了瑞安我第一件事就是把 IBM 赶出瑞安广场,哼.算了,有些东西,注定和我们无缘,以前是,现在是,将来也是.

关于 list_entry(),让我们结合实例来看,我们在 hub 的故事里会接触到的一个重要的数据结构就是 struct usb_hub,来自 drivers/usb/core/hub.c

```
34 struct usb_hub {
35          struct device           *intfdev;       /* the "interface" device */
36          struct usb_device       *hdev;
37          struct urb              *urb;           /* for interrupt polling pipe */
38
39          /* buffer for urb ... with extra space in case of babble */
40          char                    (*buffer)[8];
```

```
41          dma_addr_t              buffer_dma;     /* DMA address for buffer
*/
42          union {
43                  struct usb_hub_status   hub;
44                  struct usb_port_status  port;
45          }                       *status;        /* buffer for status reports */
46          struct mutex            status_mutex;   /* for the status buffer */
47
48          int                     error;          /* last reported error */
49          int                     nerrors;        /* track consecutive errors */
50
51          struct list_head        event_list;     /* hubs w/data or errs ready */
52          unsigned long           event_bits[1];  /* status change bitmask */
53          unsigned long           change_bits[1]; /* ports with logical connect
54                                                     status change */
55          unsigned long           busy_bits[1];   /* ports being reset or
56                                                     resumed */
57 #if USB_MAXCHILDREN > 31 /* 8*sizeof(unsigned long) - 1 */
58 #error event_bits[] is too short!
59 #endif
60
61          struct usb_hub_descriptor *descriptor;  /* class descriptor */
62          struct usb_tt           tt;             /* Transaction Translator */
63
64          unsigned                mA_per_port;    /* current for each child */
65
66          unsigned                limited_power:1;
67          unsigned                quiescing:1;
68          unsigned                activating:1;
69
70          unsigned                has_indicators:1;
71          u8                      indicator[USB_MAXCHILDREN];
72          struct delayed_work     leds;
73 };
```

看到了吗,51 行,struct list_head event_list,这个结构体变量将为我们组建一个队列,或者说组建一个链表.我们知道,一个 struct usb_hub 代表一个 hub,每一个 struct usb_hub 有一个 event_list,即每一个 hub 都有它自己的一个事件列表,要知道 hub 可以有一个或者有多个,而 hub 驱动只需要一个,或者说 khubd 这个精灵进程永远都只有一个,而我们的做法是,不管实际上有多少个 hub,我们最终都会将其 event_list 挂入到全局链表 hub_event_list 中来统一处理(hub_event_list 是对于整个 usb 系统来说是全局的,但对于整个内核来说当然是局部的,毕竟它前面有一个 static).因为最终处理所有 hub 的事务的都是我们这一个精灵进程 khubd,它只需要判断 hub_event_list 是否为空,不为空就去处理.或者说就去出发 hub_events()函数,但当我们真的要处理 hub 的事件的时候,我们当然又要知道具体是哪个 hub 触发了这起事件.而

list_entry 的作用就是,从 struct list_head event_list 得到它所对应的 struct usb_hub 结构体变量.比如以下四行代码:

```
struct list_head *tmp;

struct usb_hub *hub;

tmp=hub_event_list.next;

hub=list_entry(tmp,struct usb_hub,event_list);
```

从全局链表 hub_event_list 中取出一个来,叫做 tmp,然后通过 tmp,获得它所对应的 struct usb_hub.

最后,总结陈词,本方论点:中学我们学习写议论文的时候,老师教过有这样几种结构,总分总式结构,对照式结构,层进式结构,并列式结构.而总分总式就是先提出中心论点,然后围绕中心,以不同角度提出分论点,展开论述,最后进行总结.而总分总具体来说又有总分,分总,总分总三种形式.以前我以为Linus只是技术比我强,现在我算是看明白了,这家伙语文学得也比我好.看出来了么?这里采用的就是我们议论文中的总分总结构,先设置一个链表,hub_event_list,设置一个总的函数 hub_events(),这是总,然后每一个 hub 都有一个 event_list,每当有一个 hub 的 event_list 出现了变化,就把它的 event_list 插入到 hub_event_list 中来,这是分,然后触发总函数 hub_events(),这又是总,然后在 hub_events()里又根据 event_list 来确定是哪个 struct usb_hub,或者说是哪个 hub 有事情,又针对该 hub 进行具体处理,这又是分.这就是 Linux 中 hub 驱动的中心思想.Linus,I 服了 U!从前我只佩服复旦大学中文系系主任陈思和,他早上 8 点的公选课学生 7 点去就没有了座位,堪称复旦的奇迹,而 Linus 你和你的那些伙伴们让我们知道了真正的文学泰斗是从来不写文学著作的!是从来不讲论语心得,从来不去品三国的!

最后的最后,提醒一下,struct usb_hub 这里贴出来了,稍后讲到这个结构体的时候就不会再贴了.

等待,只因曾经承诺

hub_thread()中还有一个函数没有讲.它就是 try_to_freeze().这是电源管理相关的.对大多数人来说,关于这个函数,了解就可以了.以下的内容就当科普性质吧,也算哥们儿为奥运做点贡献,提高国民科学文化知识水平.随着 Linux 开始支持 suspended 之后,西方的资本家们提倡,每一个内核进程都应该在适当的时候,调用 try_to_freeze().什么意思呢?有这样一个 flag,PF_NOFREEZE,如果你这个进程或者内核线程不想进入 suspended 状态,那么你就可以设置这个 flag,正如我们在 usb-storage 中,usb_stor_control_thread()中做的那样.那就属于众人皆醉你独醒的情况,就是说大家都挂起了,唯独你想表现一下自己作为 80 后的与众不同的个性.而对于大多数内核线程来说,目前主流的看法是希望你能在某个地方调用 try_to_freeze(),这个函数的作用是检测一个 flag 有没有设置,哪个 flag 呢,TIF_FREEZE,每个体系结构定义了自己的与这有关的 flags.比如 i386 的,include/asm-i386/thread_info.h 中:

```
120 /*
121  * thread information flags
```

```
122  * - these are process state flags that various assembly files may need to
access
123  * - pending work-to-be-done flags are in LSW
124  * - other flags in MSW
125  */
126 #define TIF_SYSCALL_TRACE        0       /* syscall trace active */
127 #define TIF_NOTIFY_RESUME        1       /* resumption notification
requested */
128 #define TIF_SIGPENDING           2       /* signal pending */
129 #define TIF_NEED_RESCHED         3       /* rescheduling necessary */
130 #define TIF_SINGLESTEP           4       /* restore singlestep on return to
user mode */
131 #define TIF_IRET                 5       /* return with iret */
132 #define TIF_SYSCALL_EMU          6       /* syscall emulation active */
133 #define TIF_SYSCALL_AUDIT        7       /* syscall auditing active */
134 #define TIF_SECCOMP              8       /* secure computing */
135 #define TIF_RESTORE_SIGMASK      9       /* restore signal mask in
do_signal() */
136 #define TIF_MEMDIE               16
137 #define TIF_DEBUG                17      /* uses debug registers */
138 #define TIF_IO_BITMAP            18      /* uses I/O bitmap */
    139 #define TIF_FREEZE               19      /* is freezing for suspend */
```

TIF 就是 thread info 的意思.众所周知,struct task_struct 是一个表示进程的结构体,而除此之外,又有一个叫做 struct thread_info 的结构体来代表内核线程,而 struct thread_info 中有一个成员 struct task_struct *task,所以我们很好理解,实际上就是说对普通进程来说,我们准备一个 struct task_struct 就可以描述了,而对于内核线程来说,它可能要包含更多的信息,光一个 struct task_struct 还不足以表达它的全部.所以就跳出来一个叫做 struct thread_info 的结构体,而 struct thread_info 中也有一个成员叫做 unsigned long flags.以上定义的这些宏就是和这个 flags 对应的.我们知道 struct task_struct 里边实际上已经有一个 unsigned long flags,而这两者的区别在于,struct thread_info 中的 flags 标志的是更为底层的信息,江湖中称之为 low level flags,理由是这些标志位通常并不是我们直接设置的,而是底层专门管电源管理的代码来设置.而我们能设置的就是像 PF_NOFREEZE 这样的 flag,它就是属于 struct task_struct 的,就像我们在 usb-storage 中做的那样,设置 current->flags.

所以我们就知道,try_to_freeze()就是判断这个 low level flags 里面是否设置了这么一位,TIF_FREEZE,那么谁会设置这一位呢?不是别人,正是你.你不承认是吧,你说计算机为什么会进入 suspended 状态?是不是你让它进入的?比如笔记本,你要是不合上笔记本,或者你不按你笔记本键盘上那个 suspend 键,或者说你不做任何触发 suspended 的操作,那么计算机肯定不会自己进入 suspended 状态,或者说进入挂起状态.说挂起可能还不是很准确,因为 Linux 中,suspend 是包括挂起和暂停的,也就是说包括了 Windows 下的那种休眠.在 Linux 下都称作 suspend,但是你可以选择是挂起还是暂停.所以我们还是直接说 suspend 吧,比如当你触发了 suspend 的操作,那么从内核代码来讲,它就会对每一个进程进行判断,如果设置了 PF_NOFREEZE 这个 flag,那么就没有必然让这个进程去进入 suspended 状态了,西方的人比较幽默,他们把这一举动称为使进程进入电冰箱,就是说把进程冷冻起来.如果没有设置,那么就说

明你并不反对内核让你进入 suspended,这就相当于一个人被别人侵犯,她的态度是半推半就,那么对方就会肆无忌惮的去做他想做的事情.具体来说,kernel 会做什么呢?最重要的就是设置了 TIF_FREEZE 这个 flag,然后给当前进程发送信号.理想情况下,hub_events()函数会执行完,然后调用 wait_event_interruptible()进入睡眠,然后在接到信号之后,wait_event_interruptible()会退出,然后 try_to_freeze()会被执行,try_to_freeze()实际上会判断 TIF_FREEZE 是否被设置,如果是,就调用 refrigerator()来设置一个叫做 PF_FROZEN 的 flag,从而进入 suspended 或者说进入 frozen,也就是说进入电冰箱,或者说进入冷冻状态.应该说,电源管理是 Linux 内核中目前比较热门的话题,关于电源管理的代码也是近年来才不断发展起来的,尤其在 2.6.20 开始,代码中绝大多数内核线程里都引入了 try_to_freeze()这么一个函数来配合 suspended 特性.try_to_freeze()被称为 suspend hook.而当系统从 suspended 状态恢复过来时候,一个叫做 thaw_processes()的函数会被调用,从而清除掉 PF_FROZEN 这个 flag,然后这里进入 frozen 的进程又会被恢复,会被解冻.即 refrigerator 函数会结束,然后 do/while 开始新的一轮循环.如果循环条件不满足,那么循环结束,pr_debug 就是一句打印语句,而 usbcore_name 就是一个字符串,"usbcore".然后 hub_thread()返回 0.故事就结束了.

小结一下:如果你对以上这些讨论电源管理的东西不感兴趣,不要紧,丝毫不要紧,Linux 中很多关于电源管理的东西至今为止还都是不成熟的,写代码的兄弟们天天都在修改.问题多多,革命尚未成功,同志仍需努力.值此北京奥运会倒计时一周年之际,就让我们在精神上支持一下以 Rafael J. Wysocki 为代表的战斗在 Linux 内核电源管理领域前线的同志们.等将来我们国家房价降下去了,大家生活压力不是那么大了,让我们也去加入这支队伍吧.不过我们 80 后是没希望了,房价还没降,猪肉价格倒是已经涨上来了.一句话,如果你不想支持电源管理,那么你编译内核的时候把 CONFIG_PM 给关了,一了百了.不过,有一个问题,usb 设备实际上是有节电这么一个特性的,也就是说你翻一翻 usb 的各种规范,其中就有一个 suspend 一个 resume,就是挂起和恢复,换言之,硬件本身有这样的特性,你要是软件不支持的话,那你不心虚吗?写出来的代码你敢给客户用吗?所以没有办法,接下来我们要是看到电源管理的代码还是会讲,我们只能认了.命苦不能怪政府,点背不能怪社会.一个利好消息是,除了这里这个 try_to_freeze()比较难一点外,剩下的在 usb 中出现的电源管理的代码实际上相对来说不是很难理解,毕竟那些东西和硬件规范是对应的,都有章可循,硬件怎么规定就怎么做,所以,不用太担心.

摆平了外面的这行代码,于是现在我们安心来看 hub_events()了.这又是一个变态的函数,好几百行,我简直怀疑写代码的人是不是吃了黄金搭档,补足了钙铁锌硒维生素,要不然怎么能精力这么充沛写了一个长函数又一个,而且个个都是有技术含量的,苦了我们这些看代码的了.算了,不跟这些人一般见识,记不记得小时候父母最伤我们自尊的事情就是经常拿比我们"行"的人刺激我们,这些事情总让我们很受伤,现在我们长大了,又何必再去自己寻找这些烦恼呢,我们这一代人的烦恼还少吗?我们小的时候,大学生是天子骄子,当我们大学毕业的时候,大学生是锅里的饺子.我们没能工作的时候,工作也是分配的,我们可以工作的时候,撞得头破血流才勉强找份饿不死人的工作做.不想这些了,乐观一点看,这些人比我们懂更多 C 语言,可是他们未必像我们一样懂那么多网络语言,我们未必懂他们写的东西,可是我们说火星,说 orz,说我顶你个肺,他们也不懂.hub_events(),还是来自 drivers/usb/core/hub.c,我们一段一段来看:

```
2595 static void hub_events(void)
2596 {
2597         struct list_head *tmp;
2598         struct usb_device *hdev;
2599         struct usb_interface *intf;
```

```
2600        struct usb_hub *hub;
2601        struct device *hub_dev;
2602        u16 hubstatus;
2603        u16 hubchange;
2604        u16 portstatus;
2605        u16 portchange;
2606        int i, ret;
2607        int connect_change;
2608
2609        /*
2610         *  We restart the list every time to avoid a deadlock with
2611         * deleting hubs downstream from this one. This should be
2612         * safe since we delete the hub from the event list.
2613         * Not the most efficient, but avoids deadlocks.
2614         */
2615        while (1) {
2616
2617                /* Grab the first entry at the beginning of the list */
2618                spin_lock_irq(&hub_event_lock);
2619                if (list_empty(&hub_event_list)) {
2620                        spin_unlock_irq(&hub_event_lock);
2621                        break;
2622                }
2623
2624                tmp = hub_event_list.next;
2625                list_del_init(tmp);
2626
2627                hub = list_entry(tmp, struct usb_hub, event_list);
2628                hdev = hub->hdev;
2629                intf = to_usb_interface(hub->intfdev);
2630                hub_dev = &intf->dev;
2631
2632                   dev_dbg(hub_dev, "state %d ports %d chg %04x evt
%04x\n",
2633                                hdev->state, hub->descriptor
2634                                        ? hub->descriptor->bNbrPorts
2635                                        : 0,
2636                                /* NOTE: expects max 15 ports... */
2637                                (u16) hub->change_bits[0],
2638                                (u16) hub->event_bits[0]);
2639
2640                usb_get_intf(intf);
2641                spin_unlock_irq(&hub_event_lock);
```

仔细一看这段代码你会发现我们中国古代军事思想中的声东击西又被人偷师了.2615 行,一个 while(1)循环,2619 行,判断 hub_event_list 是否为空,是不是觉得很有趣,第一次调用这个函数的时候,hub_event_list 就是初值,我们说过初值为空.所以这里就是空,即 list_empty()返回 1,然后 break 语句跳出 while 循环,你知道 while 循环的结尾在哪里吗?就是这个 hub_events()函数的结尾,也就是说这里几百行的代码我们就结束了,我们直接退出这个函数,返回到 hub_thread()中,调用 wait_event_interruptible()进入睡眠,然后等待有事件的发生,对于 hub 来说,事件的发生,比如你插入一个设备到 hub 口里,就会触发一件事件.而第一件事件的发生其实是 hub 驱动程序本身的初始化,即我们说过,由于 root hub 的存在,所以 hub_probe 必然会被调用,确切地说,就是在 host controller 的驱动程序中,一定会调用 hub_probe 的.如果你问我到底什么时候会调用,那么我无可奉告,因为这是在 host controller 的驱动程序中,不管你的 host controller 是属于 OHCI 的,还是 UHCI 的,还是 EHCI 的,最终在它们的初始化代码中都会调用一个叫做 hcd_register_root()的函数,进而转战 usb_register_root_hub(),几经周转,最终 hub_probe 就会被调用.所以你根本不用担心这个函数什么时刻会被调用,反正总会有这么一个时刻.正如半生缘里说的一样,我要叫你知道,这个世界上有一个人会一直等你,无论在什么时候,无论你在什么地方,反正总会有这样一个人.

所以,我们就转战去到 hub_probe 吧,这里 hub_events()只是虚晃一枪,不过你别忘了,等到 hub_event_list 里面有东西了之后,我们还会回来的.要知道 hub_events()这个函数才是真正的 hub 驱动的核心函数,所有的故事都是在这里发生的.所以,就像你给了某人一个承诺,承诺你还会回来.有了承诺,时间似乎有了刻度,有了承诺,等待也被赋予了意义.只是,也许,有一种爱经不起伤害,有一种爱经不起等待.

最后需要记住的是 wait_event_interruptible()的第一个参数是&khubd_wait,关于这个函数我们在 usb-storage 里面已经看过多次了,其中 khubd_wait 定义于 drivers/usb/core/hub.c:

```
85 /* Wakes up khubd */
86 static DECLARE_WAIT_QUEUE_HEAD(khubd_wait);
```

这无非就是一个等待队列头,所以我们很清楚,将来要唤醒这个睡眠进程的一定是类似这样的一行代码:wake_up(&khubd_wait),没错,you got it!整个内核代码中只有一个地方会调用这个代码,那就是 kick_khubd(),不过调用 kick_khubd()的地方可不少,我们走着瞧.

最熟悉的陌生人--probe

话说因为 hub 驱动无所事事,所以 hub_thread()进入了睡眠,直到某一天,hub_probe 被调用.所以我们来看 hub_probe().这个函数来自 drivers/usb/hub.c,其作用就如同当初我们在 usb-storage 中的那个 storage_probe()一样.

```
887 static int hub_probe(struct usb_interface *intf, const struct usb_device_id *id)
888 {
889         struct usb_host_interface *desc;
890         struct usb_endpoint_descriptor *endpoint;
891         struct usb_device *hdev;
892         struct usb_hub *hub;
893
```

```
894         desc = intf->cur_altsetting;
895         hdev = interface_to_usbdev(intf);
896
897 #ifdef  CONFIG_USB_OTG_BLACKLIST_HUB
898         if (hdev->parent) {
899                 dev_warn(&intf->dev, "ignoring external hub\n");
900                 return -ENODEV;
901         }
902 #endif
903
904         /* Some hubs have a subclass of 1, which AFAICT according to the */
905         /*  specs is not defined, but it works */
906         if ((desc->desc.bInterfaceSubClass != 0) &&
907             (desc->desc.bInterfaceSubClass != 1)) {
908 descriptor_error:
909                 dev_err (&intf->dev, "bad descriptor, ignoring hub\n");
910                 return -EIO;
911         }
912
913         /* Multiple endpoints? What kind of mutant ninja-hub is this? */
914         if (desc->desc.bNumEndpoints != 1)
915                 goto descriptor_error;
916
917         endpoint = &desc->endpoint[0].desc;
918
919         /* If it's not an interrupt in endpoint, we'd better punt! */
920         if (!usb_endpoint_is_int_in(endpoint))
921                 goto descriptor_error;
922
923         /* We found a hub */
924         dev_info (&intf->dev, "USB hub found\n");
925
926         hub = kzalloc(sizeof(*hub), GFP_KERNEL);
927         if (!hub) {
928                 dev_dbg (&intf->dev, "couldn't kmalloc hub struct\n");
    929                 return -ENOMEM;
930         }
931
932         INIT_LIST_HEAD(&hub->event_list);
933         hub->intfdev = &intf->dev;
934         hub->hdev = hdev;
935         INIT_DELAYED_WORK(&hub->leds, led_work);
936
937         usb_set_intfdata (intf, hub);
```

```
938             intf->needs_remote_wakeup = 1;
939
940             if (hdev->speed == USB_SPEED_HIGH)
941                     highspeed_hubs++;
942
943             if (hub_configure(hub, endpoint) >= 0)
944                     return 0;
945
946             hub_disconnect (intf);
947             return -ENODEV;
948 }
```

幸运的是这个函数还不是很长.看过 usb-storage 的兄弟姐妹们应该不难看懂这个函数.尤其是 894,895 这几行经典的赋值.尽管当年我们看的 usb-storage 是 2.6.10 的内核,而斗转星移,如今我们看的是 2.6.22.1 的内核,但是江山会变,四季会变,有些经典并不会改变.894 行,desc,是这个函数里定义的一个 struct usb_host_interface 结构体指针,其实这就相当于当年的那个 altsetting,只是换了个名字,别以为披上马夹咱就不认识它了.struct usb_host_interface 结构体的定义依然还是当初那样,鉴于子曾经曰过温故而知新,我们这里再贴一次这个结构体吧,来自 include/linux/usb.h:

```
69 /* host-side wrapper for one interface setting's parsed descriptors */
70 struct usb_host_interface {
71             struct usb_interface_descriptor desc;
72
73             /* array of desc.bNumEndpoint endpoints associated with this
74              * interface setting.  these will be in no particular order.
75              */
76             struct usb_host_endpoint *endpoint;
77
78             char *string;               /* iInterface string, if present */
79             unsigned char *extra;   /* Extra descriptors */
80             int extralen;
81 };
```

需要注意的是,71 行,这里有一个成员,对应接口描述符的结构体,struct usb_interface_descriptor desc,刚才我们的那个指针也叫做 desc,所以一会我们就会看到,desc->desc 这样的用法.

同样 895 行这个赋值我们也是很眼熟, interface_to_usbdev()这个宏就是为了从一个 struct usb_interface 的结构体指针得到那个与它相关的 struct usb_device 结构体指针.这里等号右边的 intf 自不必说,而左边的 hdev 正是我们这里为了 hub 而定义的一个 struct usb_device 结构体指针.

897 到 902 行,这是为 OTG 而准备的,为了简化问题,在这里我做一个伟大的假设,即假设我们不支持 OTG.在内核编译选项中有一个叫做 CONFIG_USB_OTG 的选项,OTG 就是 On The Go 的意思,正在进行中的意思,随着 USB 传输协议的诞生以及它的迅速走红,人们不再满足于以前那种一个设备要么就是主设备,要么就是从设备的现状,也就是说要么是 Host,或者叫主设备,要么

是外设,也叫 Slave,或者叫从设备.那个年代里,只有当一台 Host 与一台 Slave 连接时才能实现数据的传输,而后来善良的开发者们又公布了 USB OTG 规范,于是出现了 OTG 设备,即既可以充当 Host,亦能充当 Slave 的设备.就是说你有一个数码相机,你有一台打印机,它们各有一个 USB 接口,把这两个口连接起来,然后就可以把你偷拍的美女照片打印出来了.不过我们为了省事,还是别玩这种高科技了吧,省点时间玩几盘 CS 不是挺好么?所以我只能假设我们不打开支持 OTG 的编译开关,而这里我们看到的 CONFIG_USB_OTG_BLACKLIST_HUB,其实就是一 CONFIG_OTG 下面的子选项,不选后者根本就见不到前者,因此咱们也不用看.

904 到 911 行,这我真是无话可说了,每一个 USB 设备它属于哪个类,以及哪个子类,这都是上天注定的,自打盘古开天地那会儿就已经确定下来了,比如 hub 的子类就是 0,即 desc->desc 这个 interface 描述符里边的 bInterfaceSubClass 就该是 0.所以这里本是判断如果 bInterfaceSubClass 不为 0 那就出错了,那就甭往下走了,返回吧,返回值是-EIO.就像一个人如果连自己是哪一类物种都能弄错,那还活个什么劲呢?不过我真正来气的是偏偏有些没事找抽型的企业愣是把自己家生产的 hub 里边的描述符中的 bInterfaceSubClass 这一位弄成了 1,完了实践证明该 hub 也还能工作,别的方面都还正常,你说你要是调试设备驱动程序老是碰上这样的设备是不是非得急死你?

914,915 行,其实干的事情是差不多的,针对接口描述符再做一次判断,这次是判断这个 hub 有几个端点,或者说 Endpoint.spec 规定了 hub 就是一个 endpoint,中断 endpoint,因为 hub 的传输是中断传输.当然还有控制传输,但是因为控制传输是每一个设备都必须支持的,即每一个 usb 设备都会有一个控制端点,所以在 desc->desc.bNumEndpoints 中是不包含那个大家都有的控制端点的.因此如果这个值不为 1,那么就说明又出错了,仍然只能是返回.

917 行,就是得到这个唯一的端点所对应的端点描述符,920 和 921 行就是判断这个端点是不是中断端点,如果不是那还是一样,返回报错吧.

如果以上几种常见的错误都没有出现,那这时候我们才开始正式的去做一些事情,让我们继续,向前进,向前进,战士的责任重,妇女的冤仇深.

924 行,打印调试信息.

926 行,申请 hub 的数据结构 struct usb_hub.早些时候我们已经贴出来这个结构体的定义了,不记得的回去看看.不过 926 行有一个很新潮的函数,kzalloc().其实这个函数就是原来的两个函数的整合,即原来我们每次申请内存的时候都会这么做,先是用 kmalloc()申请空间,然后用 memset()来初始化,而现在省事了,一步到位,直接调用 kzalloc(),效果等同于原来那两个函数,所有申请的元素都被初始化为 0.其实对写驱动的来说,知道现在应该用 kzalloc()代替原来的 kmalloc()和 memset()就可以了,这是内核中内存管理部分做出的改变,确切的说是改进,负责内存管理那部分的兄弟们的目标无非就是让内核跑起来更快一些,而从 kmalloc/memset 到 kzalloc 的改变确实也是为了实现这方面的优化.所以自从 2005 年底内核中引入 kzalloc 之后,忽如一夜春风来,整个内核代码的许多模块里面都先后把原来的 kmalloc/memset 统统换成了 kzalloc().咱们这里就是其中一处.927 到 930 行不用说了,如果没申请成功那就挂了,返回 ENOMEM.

932 行,还记得咱们之前说了什么吗,总分的想法,一个总的事件队列,hub_event_list,然后各个 hub 都有一个分的事件队列,就是这里的 hub->event_list,前面咱们初始化了全局的那个 hub_event_list,而这里咱们针对单个 hub 就得为其初始化一个 event_list.

933 行和 934 行,struct usb_hub 中的两个成员,struct device *intfdev, struct usb_device *hdev,干嘛用的想必不用多说了吧,第一个,甭管你是 usb 设备也好,pci 设备也好,scsi 设备也好,Linux 内核中都为你准备一个 struct device 结构体来描述,所以 intfdev 就是和咱们这 usb hub 相关联的 struct device 指针,第二个,甭管你是 hub 也好,u 盘也好,移动硬盘也好,usb 鼠标也好,usb core 都为你准备一个 struct usb_device 来描述,所以 hdev 就将是与咱们这个 hub 相对应的 struct usb_device 指针.而这些,在我们调用 hub_probe 之前就已经建立好了,都在那个参数 struct usb_interface *intf 中,具体怎么得到的,对于 root hub 来说,这涉及到 host controller 的驱动程序,咱们先不去理睬.但对于一个普通的外接的 hub,那咱们一会儿会看到如何得到它的 struct usb_interface,因为建立并初始化一个 usb 设备的 struct usb_interface 正是 hub 驱动里做的事情.其实也就是我们对 hub 驱动最好奇的地方.因为找到了这个问题的答案,我们就知道了对于一个 usb 设备驱动,其 probe 指针是在什么情况下被调用的,比如咱们这里的 hub_probe 对于普通 hub 来说是谁调用的?比如咱们之前那个 usb-storage 中最有意思的函数 storage_probe()究竟是谁调用的?这正是我们想知道的.

蝴蝶效应

朋友,你相信,一只蝴蝶在北京拍拍翅膀,将使得纽约几个月后出现比狂风还厉害的龙卷风吗?看过那部经典的影片蝴蝶效应的朋友们一定会说,这不就是蝴蝶效应吗.没错.蝴蝶效应其实是混沌学理论中的一个概念.它是指对初始条件敏感性的一种依赖现象.蝴蝶效应的原因在于蝴蝶翅膀的运动,导致其身边的空气系统发生变化,并引起微弱气流的产生,而微弱气流的产生又会引起它四周空气或其它系统产生相应的变化,由此引起连锁反应,最终导致其它系统的极大变化.

自从 1979 年 12 月麻省理工的洛仑兹大侠在美国科学促进会上作了关于蝴蝶效应的报告之后,从此蝴蝶效应很快风靡全球,其迷人的美学色彩和深刻的科学内涵令许多人着迷,激动,同时发人深省.蝴蝶效应被引入了各个领域,比如军事,比如政治,比如经济,再后来也被引入到了企业管理,甚至我们的人生历程里也存在.当然 Linux 中也不会放过如此有哲学魅力的理论.从本质上来说,蝴蝶效应给人一种对未来行为不可预测的危机感.而 Linux 内核代码中这种感觉更是强烈,几乎到了无处不在的程度.很多函数,特别是那种做初始化的函数,你根本就不知道它在干什么,只有当你在未来某个时刻,看到了另一个函数,你才会回过头来看,原来当初是这个函数设置了初始条件.假如你改变了初始条件,那么后来你某个地方的某个函数的某个行为就会发生改变.但问题是,你并不知道这个行为将在午夜 12 点发生还是在下午 3 点半发生.

是不是觉得很玄?像思念一样玄?那好,我们来看点具体的,比如 935 行,INIT_DELAYED_WORK().这是一张新面孔.同志们大概注意到了,在 hub 这个故事里,我们的讲解风格略有变化,对于那些旧的东西,对于那些在 usb-storage 里面讲过很多次的东西,我们不会再多提,但是对于新鲜的东西,我们会花大把的笔墨去描摹.这样做的原因很简单,男人嘛,有几个不是喜新厌旧呢,要不然也不会结婚前觉得适合自己的女人很少,结婚后觉得适合自己的女人很多.

所以本节我们就用大把的笔墨来讲述老百姓自己的故事.就讲这一行,935 行.INIT_DELAYED_WORK()是一个宏,我们给它传递了两个参数.&hub->leds 和 led_work.对设备驱动熟悉的人不会觉得 INIT_DELAYED_WORK()很陌生,其实鸦片战争那会儿就有这个宏了,只不过从 2.6.20 的内核开始这个宏做了改变,原来这个宏是三个参数,后来改成了两个参数,所以经常在网上看见一些同志抱怨说最近某个模块编译失败了,说什么 make 的时候遇见这么一个错误:

error: macro "INIT_DELAYED_WORK" passed 3 arguments, but takes just 2

当然更为普遍的看到下面这个错误:

error: macro "INIT_WORK" passed 3 arguments, but takes just 2

于是就让我们来仔细看看 INIT_WORK 和 INIT_DELAYED_WORK.其实前者是后者的一个特例,它们涉及到的就是传说中的工作队列.这两个宏都定义于 include/linux/workqueue.h 中:

```
79 #define INIT_WORK(_work, _func)                                          \
80         do {                                                             \
81                 (_work)->data = (atomic_long_t) WORK_DATA_INIT();
\
82                 INIT_LIST_HEAD(&(_work)->entry);                         \
83                 PREPARE_WORK((_work), (_func));                          \
84         } while (0)
85
86 #define INIT_DELAYED_WORK(_work, _func)                                  \
87         do {                                                             \
88                 INIT_WORK(&(_work)->work, (_func));                      \
89                 init_timer(&(_work)->timer);                             \
90         } while (0)
```

有时候特怀念谭浩强那本书里的那些例子程序,因为那些程序都特简单,不像现在看到的这些,动不动就是些复杂的函数复杂的数据结构复杂的宏,严重挫伤了我这样的有志青年的自信心.就比如眼下这几个宏吧,宏里边还是宏,一个套一个,不是说看不懂,因为要看懂也不难,一层一层展开,只不过确实没必要非得都看懂,现在这样一种朦胧美也许更美,有那功夫把这些都展开我还不如去认认真真学习三个代表呢.总之,关于工作队列,就这么说吧,Linux 内核实现了一个内核线程,直观一点,ps 命令看一下您的进程,

```
localhost:/usr/src/linux-2.6.22.1/drivers/usb/core # 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     4     1  0 -40   - -     0 migrat ?        00:00:00 migration/1
1 S     0     5     1  0  94  19 -     0 ksofti ?        00:00:00 ksoftirqd/1
1 S     0     6     1  0 -40   - -     0 migrat ?        00:00:00 migration/2
1 S     0     7     1  0  94  19 -     0 ksofti ?        00:00:00 ksoftirqd/2
1 S     0     8     1  0 -40   - -     0 migrat ?        00:00:00 migration/3
1 S     0     9     1  0  94  19 -     0 ksofti ?        00:00:00 ksoftirqd/3
1 S     0    10     1  0 -40   - -     0 migrat ?        00:00:00 migration/4
1 S     0    11     1  0  94  19 -     0 ksofti ?        00:00:00 ksoftirqd/4
1 S     0    12     1  0 -40   - -     0 migrat ?        00:00:00 migration/5
1 S     0    13     1  0  94  19 -     0 ksofti ?        00:00:00 ksoftirqd/5
1 S     0    14     1  0 -40   - -     0 migrat ?        00:00:00 migration/6
1 S     0    15     1  0  94  19 -     0 ksofti ?        00:00:00 ksoftirqd/6
```

```
1 S     0    16     1   0 -40   - -      0 migrat ?        00:00:00 migration/7
1 S     0    17     1   0  94  19 -      0 ksofti ?        00:00:00 ksoftirqd/7
5 S     0    18     1   0  70  -5 -      0 worker ?        00:00:00 events/0
1 S     0    19     1   0  70  -5 -      0 worker ?        00:00:00 events/1
5 S     0    20     1   0  70  -5 -      0 worker ?        00:00:00 events/2
5 S     0    21     1   0  70  -5 -      0 worker ?        00:00:00 events/3
5 S     0    22     1   0  70  -5 -      0 worker ?        00:00:00 events/4
1 S     0    23     1   0  70  -5 -      0 worker ?        00:00:00 events/5
5 S     0    24     1   0  70  -5 -      0 worker ?        00:00:00 events/6
5 S     0    25     1   0  70  -5 -      0 worker ?        00:00:00 events/7
```

瞅见最后这几行了吗,events/0 到 events/7,0 啊 7 啊这些都是处理器的编号,每个处理器对应其中的一个线程.要是您的计算机只有一个处理器,那么您只能看到一个这样的线程,events/0,您要是双处理器那您就会看到多出一个 events/1 的线程.哥们儿这里 Dell PowerEdge 2950 的机器,8 个处理器,所以就是 events/0 到 events/7 了.

那么究竟这些 events 代表什么意思呢?或者说它们具体干嘛用的?这些 events 被叫做工作者线程,或者说 worker threads,更确切的说,这些应该是缺省的工作者线程.而与工作者线程相关的一个概念就是工作队列,或者叫 work queue.工作队列的作用就是把工作推后,交由一个内核线程去执行,更直接的说就是如果您写了一个函数,而您现在不想马上执行它,您想在将来某个时刻去执行它,那您用工作队列准没错.您大概会想到中断也是这样,提供一个中断服务函数,在发生中断的时候去执行,没错,和中断相比,工作队列最大的好处就是可以调度可以睡眠,灵活性更好.

就比如这里,如果我们将来某个时刻希望能够调用 led_work()这么一个我们自己写的函数,那么我们所要做的就是利用工作队列.如何利用呢?第一步就是使用 INIT_WORK()或者 INIT_DELAYED_WORK()来初始化这么一个工作,或者叫任务,初始化了之后,将来如果咱们希望调用这个 led_work()函数,那么咱们只要用一句 schedule_work()或者 schedule_delayed_work()就可以了,特别的,咱们这里使用的是 INIT_DELAYED_WORK(),那么之后我们就会调用 schedule_delayed_work(),这俩是一对.它表示,您希望经过一段延时然后再执行某个函数,所以,咱们今后会见到 schedule_delayed_work()这个函数的,而它所需要的参数,一个就是咱们这里的&hub->leds,另一个就是具体自己需要的延时.&hub->leds 是什么呢?struct usb_hub 中的成员,struct delayed_work leds,专门用于延时工作的,再看 struct delayed_work,这个结构体定义于 include/linux/workqueue.h:

```
35 struct delayed_work {
36         struct work_struct work;
37         struct timer_list timer;
38 };
```

其实就是一个 struct work_struct 和一个 timer_list,前者是为了往工作队列里加入自己的工作,后者是为了能够实现延时执行,咱们把话说得更明白一点,您看那些 events 线程,它们对应一个结构体,struct workqueue_struct,也就是说它们维护着一个队列,完了您要是想利用工作队列这么一个机制呢,您可以自己创建一个队列,也可以直接使用 events 对应的这个队列,对于大多数情况来说,都是选择了 events 对应的这个队列,也就是说大家都共用这么一个队列,怎么用呢?先初始化,比如调用 INIT_DELAYED_WORK(),这么一初始化吧,实际上就是为一个 struct work_struct 结构体绑定一个函数,就比如咱们这里的两个参数,&hub->leds 和 led_work()

的关系,就最终让hub_leds这个struct work_struct结构体和函数led_work()相绑定了起来,您问怎么绑定的?您瞧,struct work_struct 也是定义于 include/linux/workqueue.h:

```
24 struct work_struct {
25          atomic_long_t data;
26 #define WORK_STRUCT_PENDING 0          /* T if work item pending execution */
27 #define WORK_STRUCT_FLAG_MASK (3UL)
28 #define WORK_STRUCT_WQ_DATA_MASK (~WORK_STRUCT_FLAG_MASK)
29          struct list_head entry;
30          work_func_t func;
31 };
```

瞅见最后这个成员 func 了吗,初始化的目的就是让 func 指向 led_work(),这就是绑定,所以以后咱们调用 schedule_delayed_work()的时候,咱们只要传递 struct work_struct 的结构体参数即可,不用再每次都把 led_work()这个函数名也给传递一次,一旦绑定,人家就知道了,对于led_work(),那她就嫁鸡随鸡,嫁狗随狗,嫁混蛋随混蛋了.您大概还有一个疑问,为什么只要这里初始化好了,到时候调用 schedule_delayed_work()就可以了呢?事实上,events 这么一个线程吧,它其实和 hub 的内核线程一样,有事情就处理,没事情就睡眠,也是一个死循环,而schedule_delayed_work()的作用就是唤醒这个线程,确切的说,是先把自己的这个 struct work_struct 插入 workqueue_struct 这个队列里,然后唤醒昏睡中的 events.然后 events 就会去处理,您要是有延时,那么它就给您安排延时以后执行,您要是没有延时,或者您设了延时为 0,那好,那就赶紧给您执行.咱这里不是讲了两个宏吗,一个 INIT_WORK(),一个INIT_DELAYED_WORK(),后者就是专门用于可以有延时的,而前者就是没有延时的,这里咱们调用的是 INIT_DELAYED_WORK(),不过您别美,过一会您会看见 INIT_WORK()也被使用了,因为咱们 hub 驱动中还有另一个地方也想利用工作队列这么一个机制,而它不需要延时,所以就使用 INIT_WORK()进行初始化,然后在需要调用相关函数的时候调用 schedule_work()即可.此乃后话,暂且不表.

基本上这一节咱们就是介绍了 Linux 内核中工作队列机制提供的接口,两对函数 INIT_DELAYED_WORK() 对 schedule_delayed_work(),INIT_WORK() 对 schedule_work().

关于工作队列机制,咱们还会用到另外两个函数,它们是 cancel_delayed_work(struct delayed_work *work)和 flush_scheduled_work().其中 cancel_delayed_work()的意思不言自明,对一个延迟执行的工作来说,这个函数的作用是在这个工作还未执行的时候就把它给取消掉.而 flush_scheduled_work()的作用,是为了防止有竞争条件的出现,虽说哥们儿也不是很清楚如何防止竞争,可是好歹大二那年学过一门专业课,数字电子线路,尽管没学到什么有用的东西,怎么说也还是记住了两个专业名词,竞争与冒险.您要是对竞争条件不是很明白,那也不要紧,反正基本上每次 cancel_delayed_work 之后您都得调用 flush_scheduled_work()这个函数,特别是对于内核模块,如果一个模块使用了工作队列机制,并且利用了 events 这个缺省队列,那么在卸载这个模块之前,您必须得调用这个函数,这叫做刷新一个工作队列,也就是说,函数会一直等待,直到队列中所有对象都被执行以后才返回.当然,在等待的过程中,这个函数可以进入睡眠.反正刷新完了之后,这个函数会被唤醒,然后它就返回了.关于这里这个竞争,可以这样理解,events 对应的这个队列,人家本来是按部就班的执行,一个一个来,您要是突然把您的模块给卸载了,或者说你把你的那个工作从工作队列里取出来了,那 events 作为队列管理者,它可能根

本就不知道,比如说它先想好了,下午 3 点执行队列里的第 N 个成员,可是您突然把第 N-1 个成员给取走了,那您说这是不是得出错?所以,为了防止您这种唯恐天下不乱的人做出冒天下之大不韪的事情来,提供了一个函数,flush_scheduled_work(),给您调用,以消除所谓的竞争条件,其实说竞争太专业了点,说白了就是防止混乱吧.

Ok,关于这些接口就讲到这里,日后咱们自然会在 hub 驱动里见到这些接口函数是如何被使用的.到那时候再来看.这就是蝴蝶效应.当我们看到 INIT_WORK/INIT_DELAYED_WORK()的时候,我们是没法预测未来会发生什么的.所以我们只能拭目以待.又想起了那句老话,大学生活就像被强奸,如果不能反抗,那就只能静静的去享受它.

While You Were Sleeping(一)

最近看了热播的电视剧<<奋斗>>,赵宝刚导演的转型之作.里面李小璐和文章演的那对小夫妻甚是搞笑.这部片子其实号称励志篇但实际上一点也不励志,就是搞笑,像我这种严肃的人向来不喜欢这些搞笑,不过里面李小璐扮演的杨晓芸对文章演的那个向南的一番对话倒是让我觉得颇为感慨.杨晓芸一心希望向南能够有理想有目标,而向南却非常满足于现状,而这种矛盾间接导致了杨晓芸对丈夫的失望并且最终两个人走向了离婚.其实我就是一个没有目标的人,整天混日子,从前进复旦的时候是一个字,混,后来离开复旦的时候是两个字,混混.

看了这部片子之后,我决定做一个有目标的人,首先我的目标不是像其他男人那样庸俗,什么农妇山泉有点田,作为复旦大学高材生,我有一个更为宏伟的目标,就是一鼓作气,跟踪 hub_probe,直到找到那句唤醒 hub_thread()的代码为止.(画外音:我汗...这也叫宏伟?)

继续沿着 hub_probe()往下走,937 至 941 行,937 行我们在 usb-storage 里已然见过了,usb_set_intfdata(intf,hub)的作用就是让 intf 和 hub 关联起来,从此以后,我们知道 struct usb_interface *intf,就可以追溯到与之关联的 struct usb_hub 指针.这种思想是很纯朴的,很简单的,但也是很重要的,这就好比在网络时代的我们,应该熟练掌握以 google 为代表的搜索引擎的使用方法,要学会如何从一根狗毛追溯到狗的主人曾经得过什么病.

938 行,设置 intf 的 need_remote_wakeup 为 1.

940 行,如果这个设备,确切地说是这个 hub 是高速设备,那么让 highspeed_hubs 加一.highspeed_hubs 是 hub 一个 drivers/usb/core/hub.c 中的全局变量,确切地说应该还是局部变量,其定义是这样的,

```
848 static unsigned highspeed_hubs;
```

static,静态变量.其实就是 hub.c 这个文件里的全局.至于这几个变量是干嘛用的,您暂时甭管,用到了再说.

943 到 947 行,结束了这几行的话,hub_probe 就算完了.我们先不用细看每个函数,很显然,hub_configure 这个函数是用来配置 hub 的,返回值小于 0 就算出错了,这里的做法是,没出错那么 hub_probe 就返回 0,否则,那就执行 hub_disconnect(),断开,并且返回错误代码-ENODEV.hub_disconnect 其实就是和 hub_probe()对应的函数,其暧昧关系就像当初 storage_probe()和 storage_disconnect 的关系一样.我们先来看 hub_configure().这个函数简直是帅呆了,又是一个 300 来行的函数.同样还是来自 drivers/usb/core/hub.c:

```
595 static int hub_configure(struct usb_hub *hub,
```

```
596         struct usb_endpoint_descriptor *endpoint)
597 {
598         struct usb_device *hdev = hub->hdev;
599         struct device *hub_dev = hub->intfdev;
600         u16 hubstatus, hubchange;
601         u16 wHubCharacteristics;
602         unsigned int pipe;
603         int maxp, ret;
604         char *message;
605
606                 hub->buffer = usb_buffer_alloc(hdev, sizeof(*hub->buffer),
GFP_KERNEL,
607                         &hub->buffer_dma);
608         if (!hub->buffer) {
609                 message = "can't allocate hub irq buffer";
610                 ret = -ENOMEM;
611                 goto fail;
612         }
613
614         hub->status = kmalloc(sizeof(*hub->status), GFP_KERNEL);
615         if (!hub->status) {
616                 message = "can't kmalloc hub status buffer";
617                 ret = -ENOMEM;
618                 goto fail;
619         }
620         mutex_init(&hub->status_mutex);
621
622         hub->descriptor = kmalloc(sizeof(*hub->descriptor), GFP_KERNEL);
623         if (!hub->descriptor) {
624                 message = "can't kmalloc hub descriptor";
625                 ret = -ENOMEM;
626                 goto fail;
627         }
628
629         /* Request the entire hub descriptor.
630          * hub->descriptor can handle USB_MAXCHILDREN ports,
631          * but the hub can/will return fewer bytes here.
632          */
633         ret = get_hub_descriptor(hdev, hub->descriptor,
    634                         sizeof(*hub->descriptor));
635         if (ret < 0) {
636                 message = "can't read hub descriptor";
637                 goto fail;
638         } else if (hub->descriptor->bNbrPorts > USB_MAXCHILDREN) {
```

```
639                 message = "hub has too many ports!";
640                 ret = -ENODEV;
641                 goto fail;
642         }
643
644         hdev->maxchild = hub->descriptor->bNbrPorts;
645         dev_info (hub_dev, "%d port%s detected\n", hdev->maxchild,
646                 (hdev->maxchild == 1) ? "" : "s");
647
648                                                 wHubCharacteristics      =
le16_to_cpu(hub->descriptor->wHubCharacteristics);
649
650         if (wHubCharacteristics & HUB_CHAR_COMPOUND) {
651                 int      i;
652                 char     portstr [USB_MAXCHILDREN + 1];
653
654                 for (i = 0; i < hdev->maxchild; i++)
655                         portstr[i] = hub->descriptor->DeviceRemovable
656                                         [((i + 1) / 8)] & (1 << ((i + 1) % 8))
657                                 ? 'F' : 'R';
658                 portstr[hdev->maxchild] = 0;
659                 dev_dbg(hub_dev, "compound device; port removable status:
%s\n", portstr);
660         } else
661                 dev_dbg(hub_dev, "standalone hub\n");
662
663         switch (wHubCharacteristics & HUB_CHAR_LPSM) {
664                 case 0x00:
665                         dev_dbg(hub_dev, "ganged power switching\n");
666                         break;
667                 case 0x01:
668                                 dev_dbg(hub_dev, "individual port power
switching\n");
669                         break;
670                 case 0x02:
671                 case 0x03:
672                         dev_dbg(hub_dev, "no power switching (usb 1.0)\n");
673                         break;
674         }
675
676         switch (wHubCharacteristics & HUB_CHAR_OCPM) {
677                 case 0x00:
678                                 dev_dbg(hub_dev, "global over-current
protection\n");
```

```
679                     break;
680             case 0x08:
681                         dev_dbg(hub_dev, "individual port over-current
protection\n");
682                     break;
683             case 0x10:
684             case 0x18:
685                     dev_dbg(hub_dev, "no over-current protection\n");
686                     break;
687         }
688
689         spin_lock_init (&hub->tt.lock);
690         INIT_LIST_HEAD (&hub->tt.clear_list);
691         INIT_WORK (&hub->tt.kevent, hub_tt_kevent);
692         switch (hdev->descriptor.bDeviceProtocol) {
693             case 0:
694                     break;
695             case 1:
696                     dev_dbg(hub_dev, "Single TT\n");
697                     hub->tt.hub = hdev;
698                     break;
699             case 2:
700                     ret = usb_set_interface(hdev, 0, 1);
701                     if (ret == 0) {
702                             dev_dbg(hub_dev, "TT per port\n");
703                             hub->tt.multi = 1;
704                     } else
705                                 dev_err(hub_dev, "Using single TT (err
%d)\n",
706                                     ret);
707                     hub->tt.hub = hdev;
708                     break;
709             default:
710                         dev_dbg(hub_dev, "Unrecognized hub protocol
%d\n",
711                             hdev->descriptor.bDeviceProtocol);
712                     break;
713         }
714
715         /* Note 8 FS bit times == (8 bits / 12000000 bps) ~= 666ns */
716         switch (wHubCharacteristics & HUB_CHAR_TTTT) {
717             case HUB_TTTT_8_BITS:
718                     if (hdev->descriptor.bDeviceProtocol != 0) {
    719                             hub->tt.think_time = 666;
```

```
720                             dev_dbg(hub_dev, "TT requires at most %d "
721                                             "FS bit times (%d ns)\n",
722                                     8, hub->tt.think_time);
723                     }
724                     break;
725             case HUB_TTTT_16_BITS:
726                     hub->tt.think_time = 666 * 2;
727                     dev_dbg(hub_dev, "TT requires at most %d "
728                                     "FS bit times (%d ns)\n",
729                             16, hub->tt.think_time);
730                     break;
731             case HUB_TTTT_24_BITS:
732                     hub->tt.think_time = 666 * 3;
733                     dev_dbg(hub_dev, "TT requires at most %d "
734                                     "FS bit times (%d ns)\n",
735                             24, hub->tt.think_time);
736                     break;
737             case HUB_TTTT_32_BITS:
738                     hub->tt.think_time = 666 * 4;
739                     dev_dbg(hub_dev, "TT requires at most %d "
740                                     "FS bit times (%d ns)\n",
741                             32, hub->tt.think_time);
742                     break;
743     }
744
745     /* probe() zeroes hub->indicator[] */
746     if (wHubCharacteristics & HUB_CHAR_PORTIND) {
747             hub->has_indicators = 1;
748             dev_dbg(hub_dev, "Port indicators are supported\n");
749     }
750
751     dev_dbg(hub_dev, "power on to power good time: %dms\n",
752             hub->descriptor->bPwrOn2PwrGood * 2);
753
754     /* power budgeting mostly matters with bus-powered hubs,
755      * and battery-powered root hubs (may provide just 8 mA).
756      */
757         ret = usb_get_status(hdev, USB_RECIP_DEVICE, 0, &hubstatus);
758     if (ret < 2) {
759             message = "can't get hub status";
760             goto fail;
761     }
762     le16_to_cpus(&hubstatus);
763     if (hdev == hdev->bus->root_hub) {
```

```
764                 if (hdev->bus_mA == 0 || hdev->bus_mA >= 500)
765                         hub->mA_per_port = 500;
766                 else {
767                         hub->mA_per_port = hdev->bus_mA;
768                         hub->limited_power = 1;
769                 }
770         } else if ((hubstatus & (1 << USB_DEVICE_SELF_POWERED)) == 0) {
771                     dev_dbg(hub_dev, "hub controller current requirement:
%dmA\n",
772                         hub->descriptor->bHubContrCurrent);
773                 hub->limited_power = 1;
774                 if (hdev->maxchild > 0) {
775                         int remaining = hdev->bus_mA -
776                                         hub->descriptor->bHubContrCurrent;
777
778                         if (remaining < hdev->maxchild * 100)
779                                 dev_warn(hub_dev,
780                                         "insufficient power available "
781                                         "to use all downstream ports\n");
782                         hub->mA_per_port = 100;          /* 7.2.1.1 */
783                 }
784         } else {        /* Self-powered external hub */
785                 /* FIXME: What about battery-powered external hubs that
786                  * provide less current per port? */
787                 hub->mA_per_port = 500;
788         }
789         if (hub->mA_per_port < 500)
790                     dev_dbg(hub_dev, "%umA bus power budget for each
child\n",
791                             hub->mA_per_port);
792
793         ret = hub_hub_status(hub, &hubstatus, &hubchange);
794         if (ret < 0) {
795                 message = "can't get hub status";
796                 goto fail;
    797         }
798
799         /* local power status reports aren't always correct */
800                             if    (hdev->actconfig->desc.bmAttributes    &
USB_CONFIG_ATT_SELFPOWER)
801                 dev_dbg(hub_dev, "local power source is %s\n",
802                         (hubstatus & HUB_STATUS_LOCAL_POWER)
803                         ? "lost (inactive)" : "good");
804
```

```
805         if ((wHubCharacteristics & HUB_CHAR_OCPM) == 0)
806                 dev_dbg(hub_dev, "%sover-current condition exists\n",
807                         (hubstatus & HUB_STATUS_OVERCURRENT) ? "" : "no
");
808
809         /* set up the interrupt endpoint
810          * We use the EP's maxpacket size instead of (PORTS+1+7)/8
811          * bytes as USB2.0[11.12.3] says because some hubs are known
812          * to send more data (and thus cause overflow). For root hubs,
813          * maxpktsize is defined in hcd.c's fake endpoint descriptors
814          * to be big enough for at least USB_MAXCHILDREN ports. */
815         pipe = usb_rcvintpipe(hdev, endpoint->bEndpointAddress);
816         maxp = usb_maxpacket(hdev, pipe, usb_pipeout(pipe));
817
818         if (maxp > sizeof(*hub->buffer))
819                 maxp = sizeof(*hub->buffer);
820
821         hub->urb = usb_alloc_urb(0, GFP_KERNEL);
822         if (!hub->urb) {
823                 message = "couldn't allocate interrupt urb";
824                 ret = -ENOMEM;
825                 goto fail;
826         }
827
828         usb_fill_int_urb(hub->urb, hdev, pipe, *hub->buffer, maxp, hub_irq,
829                 hub, endpoint->bInterval);
830         hub->urb->transfer_dma = hub->buffer_dma;
831         hub->urb->transfer_flags |= URB_NO_TRANSFER_DMA_MAP;
832
833         /* maybe cycle the hub leds */
834         if (hub->has_indicators && blinkenlights)
835                 hub->indicator [0] = INDICATOR_CYCLE;
836
837         hub_power_on(hub);
838         hub_activate(hub);
839         return 0;
840
841 fail:
842         dev_err (hub_dev, "config failed, %s (err %d)\n",
843                         message, ret);
844         /* hub_disconnect() frees urb and descriptor */
845         return ret;
    846 }
```

不过这个函数虽然长,但是逻辑非常简单,无非就是对 hub 进行必要的配置,然后就启动 hub.这个函数的关键就是调用了另外几个经典的函数.我们一点点来看.

While You Were Sleeping(二)

老实说,从函数一个开始的 598 行直到 627 行都没有什么可说的.其中需要一提的是,606 行,调用 usb_buffer_alloc()申请内存,赋给 hub->buffer.614 行,调用 kmalloc()申请内存,赋给 hub->status.622 行,调用 kmalloc()申请内存,赋给 hub->descriptor.当然也别忘了这中间的某行,初始化一把互斥锁,hub->status_mutex.以后咱们会用得着的,到时候我们就会看到 mutex_lock/mutex_unlock()这么一对函数来获得互斥锁/释放互斥锁.不过这把锁用得不多,总共也就两个函数里调用了.咱们走着瞧.

633 行,get_hub_descriptor().应该说,从这一刻起,我们将跨入一个新的时代,这一行是里程碑的一行,其意义就好比 1992 年有一位老人在中国的南海边画了一个圈,其重要性是不言而喻的.这一行,带领我们步入了 hub 协议.从此摆在我们面前的将不仅仅是 usb 协议本身了,又加了另一座大山,那就是 hub 协议,其实 hub 协议也算 usb 协议,但是由于 hub 作为一种特殊的 usb 设备,在 usb spec 中,专门有一章是关于 hub 的,而这一章有 150 多页.这就是 usb spec 2.0 中的第十一章.简明起见,下面我将把这一章称作 hub 协议.而接下来我们的一言一行,都必须遵守 hub 协议了.来,先看 get_hub_descriptor,这就是发送一个 request,或者说一个控制传输的控制请求,以获得 hub 的描述符.基本上 hub 驱动的这些函数,大多都是来自 drivers/usb/core/hub.c 中:

```
137 /* USB 2.0 spec Section 11.24.4.5 */
138 static int get_hub_descriptor(struct usb_device *hdev, void *data, int size)
139 {
140         int i, ret;
141
142         for (i = 0; i < 3; i++) {
143                 ret = usb_control_msg(hdev, usb_rcvctrlpipe(hdev, 0),
144                                 USB_REQ_GET_DESCRIPTOR, USB_DIR_IN |
USB_RT_HUB,
145                         USB_DT_HUB << 8, 0, data, size,
146                         USB_CTRL_GET_TIMEOUT);
147                 if (ret >= (USB_DT_HUB_NONVAR_SIZE + 2))
148                         return ret;
149         }
150         return -EINVAL;
    151 }
```

看过 usb-storage 的你,一定能看懂这个函数,不过有一点我说出来一定会让你吃惊,usb_control_msg()是我们第一次遇到,不信的话可以回去查一下,usb-storage 里面根本没有用到这个函数,事实上 usb-storage 里面自己写了这么一个函数,叫做 usb_stor_control_msg().理由是,usb_control_msg()不允许你在调用这个函数的时候取消一个 request,而在 usb-storage 里我们经常做这样的事情,或者说我们有这样的需求,那么我们

当然就得自己写一个函数了.不过 hub 里面没这么多乱七八糟的需求,插拔 U 盘是一件很正常的事情,可是你说谁没事总是插拔 hub?吃多了哈药六厂的新盖中盖吗?

不过有一个好消息,即 usb_control_msg 和我们讲过的 usb_stor_control_msg 的参数是一模一样的,所以我们无需再多说.每一个 request 怎么设置都是在 spec 里边规定的.对于 GET_DESCRIPTOR 这个 request,如图所示:

bmRequestType	bRequest	wValue	wIndex	wLength	Data
10100000B	GET_DESCRIPTOR	Descriptor Type and Descriptor Index	Zero	Descriptor Length	Descriptor

hub spec 规定, bmRequestType 必须是 10100000B,normally,GET_DESCRIPTOR 的时候,bmRequestType 应该等于 10000000B.D7 为方向位,1 就说明时 Device-to-host,即 IN,D6...5 这两位表示 Request 的类型,可以是标准的类型,或者是 Class 特定的,或者是 Vendor 特定的,01 就表示 Class 特定的.D4...0 表示接受者,可以是设备,可以是接口,可以是端点.这里为 0 表示接收者是设备.(这里就体现了相同的 request,不同的 requesttype 的意义了.)

USB_DT_HUB 等于 29,而 hub spec 规定 GET_DESCRIPTOR 这个请求的 wValue 就该是 Descriptor Type 和 Descriptor Index,wValue 作为一个 word,有 16 位,所以高 8 位放 Descriptor Type,而低 8 位放 Descriptor Index,usb 2.0 spec 11.24.2.10 里规定了,hub descriptor 的 descriptor index 必须为 0.而实际上,usb spec 9.4.3 里也规定了,对于非 configuration descriptor 和 string descriptor 的其他几种标准 descriptor,其 descriptor index 必须为 0.而对于 configuration descriptor 和 string descriptor 来说,这个 descriptor index 用来表征他们的编号,比如一个设备可以有多个 configuration descriptor.编号从 0 开始.所以对于 hub descriptor 来说,wValue 就是高 8 位表示 Descriptor Type,而低 8 位设成 0 就可以了.这就是为什么"<<8"了,而 Descriptor Type,spec 中 Table 9-5 定义了各种 Descriptor 的 type,如图:

Table 9-5. Descriptor Types

Descriptor Types	Value
DEVICE	1
CONFIGURATION	2
STRING	3
INTERFACE	4
ENDPOINT	5
DEVICE_QUALIFIER	6
OTHER_SPEED_CONFIGURATION	7
INTERFACE_POWER[1]	8

比如 Device 是 1,Configuration 是 2,而 hub 这里规定了,它的 Descriptor type 是 29h.而 USB_DT_HUB=(USB_TYPE_CLASS|0x09),USB_TYPE_CLASS 就是 0x01<<5.所以这里 USB_DT_HUB 就是 0x29,即 29h,至于他为什么不直接写成 29h 那就只有天知道了.也许写代码的人想故意迷惑一下读代码的人吧,因为事实上 usb 2.0 spec 里边很清楚地直接用 29h 来表示 hub 的 descriptor type,根本不存在两个咚咚的组合.这样写代码完全是对广大观众的不负责,你说我们是不是该鄙视一下这些写代码的同志.

USB_CTRL_GET_TIMEOUT 是一个宏,值为 5000,即表示 5 秒超时.

USB_DT_HUB_NONVAR_SIZE 也是一个宏,值为 7,为啥为 7 呢?首先请你明白,我们这里要获得的是 hub descriptor,这是只有 hub 才有的一个 descriptor,就是说对于 hub 来说,除了通常的 usb 设备有的那些设备描述符,接口描述符,端点描述符以外,hub spec 自己也定义了一个描述符,这就叫 hub 描述符.而 hub 描述符的前 7 个字节是固定的,表示什么意思也是确定的,但从第八个字节开始,一切就都充满了变数.所以前 7 个字节被称为非变量,即 NON-Variable,而从第 7 个开始以后的被称为变量,即 Variable.这些变量取决于 hub 上面端口的个数.所谓的变量不是说字节的内容本身是变化的,而是说 descriptor 具体有几个字节是变化的,比如 hub 上面有两个端口,那么这个 hub 的 descriptor 的字节数和 hub 上面有 4 个端口的情况就是不一样的,显然,有四个端口就要纪录更多的信息,当然描述符里的内容就多一些,从而描述符的字节长度也不一样,这超好理解.usb_control_msg()如果执行成功,那么返回值将是成功传输的字节长度,对这里来说,就是传输的 hub 描述符的字节长度.结合 hub spec 来看,这个长度至少应该是 9,所以这里判断如果大于等于 9,那就返回.当然你要问为什么这里要循环三次,这是为了防止通信错误.hub 驱动中很多地方都这样做了,主要是因为很多设备在发送它们的描述符的时候总是出错.所以咱们没辙了,多试几次呗.

While You Were Sleeping(三)

get_hub_descriptor()结束了,然后就返回 hub_configure()中来.635 到 642 行,判断刚才的返回值,小于零当然是出错了,大于零也还要多判断一次, USB_MAXCHILDREN 是咱们自己定义的一个宏,值为 31.看 include/linux/usb.h:

```
324 #define USB_MAXCHILDREN         (31)
```

其实 hub 可以接一共 255 个端口,不过实际上遇到的 usb hub 最多的也就是说自己支持 10 个端口的.所以 31 基本上够用了.当然你要是心血来潮把这个宏改成 100,200,那也不会出事,我就不信你能在一台机器上连上百个 usb 设备.真要那样,你绝对是没事找抽型的.

我们来看一下 hub 描述符对应的数据结构,struct usb_hub 中有一个成员,struct usb_hub_descriptor *descriptor, 就是表征 hub 描述符的,来看 struct usb_hub_descriptor,定义于 drivers/usb/core/hub.h,

```
130 struct usb_hub_descriptor {
131         __u8  bDescLength;
132         __u8  bDescriptorType;
133         __u8  bNbrPorts;
134         __le16 wHubCharacteristics;
135         __u8  bPwrOn2PwrGood;
136         __u8  bHubContrCurrent;
137                 /* add 1 bit for hub status change; round to bytes */
138         __u8  DeviceRemovable[(USB_MAXCHILDREN + 1 + 7) / 8];
139         __u8  PortPwrCtrlMask[(USB_MAXCHILDREN + 1 + 7) / 8];
140 } __attribute__ ((packed));
```

看见了没有,至少 9 个字节吧,接下来我们会用到 bNbrPorts,它代表 Number of downstream facing ports that this hub supports,就是说这个 hub 所支持的下行端口,刚才这里判断的就是这个值是不是比 31 还大,如果是,那么对不起,出错了.bHubContrCurrent 是 Hub 控制器的最大电流需求,DeviceRemoveable 是用来判断这个端口连接的设备是否是可以移除的,每一个 bit 代表一个端口,如果该 bit 为 0,则说明可以被移除,为 1,就说明不可以移除.而 wHubCharacteristics 就相对来说麻烦一点了,它记录了很多信息,后面有相当一部分的代码都是在判断这个值,我们这里把 hub spec 里 hub descriptor 的定义给贴出来,如下图所示,一会儿让我们对照这张图来看 wHubCharacteristics 相关的代码:

Table 11-13. Hub Descriptor

Offset	Field	Size	Description
0	*bDescLength*	1	Number of bytes in this descriptor, including this byte
1	*bDescriptorType*	1	Descriptor Type, value: 29H for hub descriptor
2	*bNbrPorts*	1	Number of downstream facing ports that this hub supports
3	*wHubCharacteristics*	2	D1...D0: Logical Power Switching Mode 00: Ganged power switching (all ports' power at once) 01: Individual port power switching 1X: Reserved. Used only on 1.0 compliant hubs that implement no power switching D2: Identifies a Compound Device 0: Hub is not part of a compound device. 1: Hub is part of a compound device. D4...D3: Over-current Protection Mode 00: Global Over-current Protection. The hub reports over-current as a summation of all ports' current draw, without a breakdown of individual port over-current status. 01: Individual Port Over-current Protection. The hub reports over-current on a per-port basis. Each port has an over-current status. 1X: No Over-current Protection. This option is allowed only for bus-powered hubs that do not implement over-current protection.

Offset	Field	Size	Description
			D6...D5: TT Think TIme 00: TT requires at most 8 FS bit times of inter transaction gap on a full-/low-speed downstream bus. 01: TT requires at most 16 FS bit times. 10: TT requires at most 24 FS bit times. 11: TT requires at most 32 FS bit times. D7: Port Indicators Supported 0: Port Indicators are not supported on its downstream facing ports and the PORT_INDICATOR request has no effect. 1: Port Indicators are supported on its downstream facing ports and the PORT_INDICATOR request controls the indicators. See Section 11.5.3. D15...D8: Reserved
5	*bPwrOn2PwrGood*	1	Time (in 2 ms intervals) from the time the power-on sequence begins on a port until power is good on that port. The USB System Software uses this value to determine how long to wait before accessing a powered-on port.
6	*bHubContrCurrent*	1	Maximum current requirements of the Hub Controller electronics in mA.
7	*DeviceRemovable*	Variable, depending on number of ports on hub	Indicates if a port has a removable device attached. This field is reported on byte-granularity. Within a byte, if no port exists for a given location, the field representing the port characteristics returns 0. Bit value definition: 0B - Device is removable. 1B - Device is non-removable This is a bitmap corresponding to the individual ports on the hub: Bit 0: Reserved for future use. Bit 1: Port 1 Bit 2: Port 2 .... Bit *n*: Port *n* (implementation-dependent, up to a maximum of 255 ports).
Variable	*PortPwrCtrlMask*	Variable, depending on number of ports on hub	This field exists for reasons of compatibility with software written for 1.0 compliant devices. All bits in this field should be set to 1B. This field has one bit for each port on the hub with additional pad bits, if necessary, to make the number of bits in the field an integer multiple of 8.

648 行,用一个临时变量 wHubCharacteristics 来代替描述符里的那个 wHubCharacteristics, 从 650 行就开始判断了,首先判断是不是复合设备.在 drivers/usb/core/hub.h 中定义了如下一些宏,

```
95 /*
96  * wHubCharacteristics (masks)
97  * See USB 2.0 spec Table 11-13, offset 3
98  */
99 #define HUB_CHAR_LPSM                0x0003 /* D1 .. D0 */
```

```
100 #define HUB_CHAR_COMPOUND       0x0004 /* D2       */
101 #define HUB_CHAR_OCPM           0x0018 /* D4 .. D3 */
102 #define HUB_CHAR_TTTT           0x0060 /* D6 .. D5 */
103 #define HUB_CHAR_PORTIND        0x0080 /* D7       */
```

结合上面这张图,意思很明显.650 到 661 行,如果是复合设备,符合设备就是说这个设备它可能是几种设备绑在一起的,比如既可以当 hub 用又可以有别的功能,那么就用一个数组 portstr[]来记录每一个端口的设备是否可以被移除.然后打印出调试信息来.不要说你看不懂,把 i 用 0,1,2,3 这些数代入进去就明白了.

663 至 674 行, HUB_CHAR_LPSM, 表征电源切换的方式,不同的 hub 有不同的 power switching 的方式,ganged power switching 指的是所有 port 一次性上电,individual port power switching 当然就是各人自扫门前雪,哪管他人瓦上霜.而 usb 1.0 的 hub 压根儿就没有 power switching 这么一个说法.

676 到 687 行, HUB_CHAR_OCPM,表征过流保护模式,其实也没啥,不明白也无所谓,这几行无非就是打印一些调试信息.

689 到 691 行,先是初始化一个自旋锁,hub->tt.lock,struct usb_hub 中的成员,struct usb_tt tt,

```
166 /*
167  * As of USB 2.0, full/low speed devices are segregated into trees.
168  * One type grows from USB 1.1 host controllers (OHCI, UHCI etc).
169  * The other type grows from high speed hubs when they connect to
170  * full/low speed devices using "Transaction Translators" (TTs).
171  *
172  * TTs should only be known to the hub driver, and high speed bus
173  * drivers (only EHCI for now).  They affect periodic scheduling and
174  * sometimes control/bulk error recovery.
175  */
176 struct usb_tt {
177         struct usb_device       *hub;   /* upstream highspeed hub */
178         int                     multi;  /* true means one TT per port */
179         unsigned                think_time;     /* think time in ns */
180
181         /* for control/bulk error recovery (CLEAR_TT_BUFFER) */
182         spinlock_t              lock;
183         struct list_head        clear_list;     /* of usb_tt_clear */
184         struct work_struct                      kevent;
    185 };
```

知道 tt 干嘛的吗?tt 叫做 transaction translator.你可以把它想成一块特殊的电路,是 hub 里面的电路,确切的说是高速 hub 中的电路,我们知道 usb 设备有三种速度的,分别是 low speed,full speed,high speed.即所谓的低速/全速/高速,抗日战争那会儿,这个世界上只有 low speed/full speed 的设备,没有 high speed 的设备,后来解放后,国民生产力的大幅度提升催生了一种 high speed 的设备,包括主机控制器,以前只有两种接口的,OHCI/UHCI,这都是在 usb spec 1.0 的时候,后来 2.0 推出了 EHCI,高速设备应运而生.Hub 也有高速 hub 和过去的

hub,但是这里就有一个兼容性问题了,高速的 hub 是否能够支持低速/全速的设备呢?一般来说是不支持的,于是有了一个叫做 TT 的电路,它就负责高速和低速/全速的数据转换,于是,如果一个高速设备里有这么一个 TT,那么就可以连接低速/全速设备,如不然,那低速/全速设备没法用,只能连接到 OHCI/UHCI 那边出来的 hub 口里.我们先不多说,看代码.690 行,初始化一个队列,hub->tt.clear_list.然后 691 行,yeah,终于见到了 INIT_WORK(),我没忽悠你吧,hub->tt.kevent 是一个 struct work_struct 的结构体,而 hub_tt_kevent 是我们定义的函数,将会在未来某年某月的某一天去执行.另外,tt 有两种,一种是 single tt,一种是 multi tt.前者表示整个 hub 就是一个 TT,而 multi tt 表示每个端口都配了一个 TT.大多数 hub 是 single TT,因为一个 hub 一个 TT 就够了,国家资源那么紧张,何必铺张浪费.使用 single TT 就是支持国家反腐倡廉!

692 行的 switch, hdev->descriptor.bDeviceProtocol,别看走眼了,刚才咱们一直是判断 hub->descriptor,而这里是 hdev->descriptor,hdev 是 struct usb_device 结构体指针,咱们一进入这个 hub_configure()函数就赋了值的,其实就是和这个 hub 相关的那个 struct usb_device 指针.所以这里判断的描述符是标准的 usb 设备描述符,而其中 bDeviceProtocol 的含义在 hub spec 里有专门的规定.这一段就是为了设置 tt,对照 hub spec 可知,full/low speed 的 hub 的 bDeviceProtocol 是 0,这种 hub 就没有 tt.所以直接 break,啥也不用设.对于 high speed 的 hub,其 bDeviceProtocol 为 1 表示是 single tt 的.为 2 表示是 multiple tt 的.对于 case 2.这里调用了 usb_set_interface,根据 usb spec 2.0,11.23.1 一节,对于 full/low speed 的 hub,其 device descriptor 中的 bDeviceProtocol 为 0,而 interface descriptor 中的 bInterfaceProtocol 也为 0.而对于 high speed 的 hub,其中,single TT 的 hub 其 device descriptor 中的 bDeviceProtocol 是 1,而 interface descriptor 的 bInterfaceProtocol 则是 0.然而,multiple TT hub 另外还有一个 interface descriptor 以及相应的一个 endpoint descriptor,它的 device descriptor 的 bDeviceProtocol 必须设置成 2.其第一个 interface descriptor 的 bInterfaceProtocol 为 1.而第二个 interface descriptor 的 bInterfaceProtocol 则是 2.hubs 只有一个 interface,但是可以有两种 setting.usb_set_interface 就是把这个 interface(interface 0)设置成 setting 1.因为默认都是 setting 0.关于 SET_INTERFACE 这个 request,是 usb spec 2.0 的一个错误,errata 里边有更正.另外,hub->tt.hub 就是 struct usb_device 的结构体指针.hub->tt.multi 就是一个 int 型的 flag,设为 1 就表示这是一个 multi tt 的 hub.

716 行, HUB_CHAR_TTTT,后两个 TT 就是 think time 的意思.就是说 TT 在处理两个低速/全速的交易之间需要一点点时间来缓冲.这个最大的间隔就叫做 TT think time.这个时间当然不会很长.不过需要注意,这里用的单位是 FS bit time,我们知道 FS 就是 Full Speed,其速度是 12Mbps,其实也就是 1200 0000bps,8 FS bit time 就是 8bits / 1200 0000 bits per second,即约等于 666ns.所以这里就用 666ns 来记录了.不过以后你会发现,其实 think_time 这个值我们就没用过.

746 到 749 行, HUB_CHAR_PORTIND,这个表征一个叫做 port indicator 的冬冬.0 说明不支持,1 说明支持.indicator 是干嘛用的?考考你.虽说六级只考了 69 分,可是我还是知道 indicator 就是 hub 上面的那个指示灯,一闪一闪亮晶晶的指示灯.通常是两种颜色,绿色和琥珀色.你是不是还经常看见红色?这我不发表评论,其实什么颜色无所谓,萝卜白菜各有所爱,不过 usb spec 上面是给出的这两种颜色.具体实现其实就是一个 LED 灯,提供两种颜色,或者两个 LED 灯.有一定生活常识的人就应该知道,其实大多数 hub 是有指示灯的,不管 usb hub 还是别的 hub,或者 switch 什么的,统统有指示灯,因为指示灯对于工程师调试硬件产品是很有帮助的.产品出了问题,首先看看指示灯也许就知道怎么回事了,我记得以前在家里上网的时候,网络坏了,打上海电信的电话,人家首先就是问我那几个指示灯是如何亮的.所以说,不支持 port indicator 的公司一定是脑子进水了.

757 行,usb_get_status(),来自 drivers/usb/core/message.c:

```
878 /**
879  * usb_get_status - issues a GET_STATUS call
880  * @dev: the device whose status is being checked
881  * @type: USB_RECIP_*; for device, interface, or endpoint
882  * @target: zero (for device), else interface or endpoint number
883  * @data: pointer to two bytes of bitmap data
884  * Context: !in_interrupt ()
885  *
886  * Returns device, interface, or endpoint status.  Normally only of
887  * interest to see if the device is self powered, or has enabled the
888  * remote wakeup facility; or whether a bulk or interrupt endpoint
889  * is halted ("stalled").
890  *
891  * Bits in these status bitmaps are set using the SET_FEATURE request,
892  * and cleared using the CLEAR_FEATURE request.  The usb_clear_halt()
893  * function should be used to clear halt ("stall") status.
894  *
895  * This call is synchronous, and may not be used in an interrupt context.
896  *
897  * Returns the number of bytes received on success, or else the status code
898  * returned by the underlying usb_control_msg() call.
899  */
900 int usb_get_status(struct usb_device *dev, int type, int target, void *data)
901 {
902         int ret;
903         u16 *status = kmalloc(sizeof(*status), GFP_KERNEL);
904
905         if (!status)
906                 return -ENOMEM;
907
908         ret = usb_control_msg(dev, usb_rcvctrlpipe(dev, 0),
909                 USB_REQ_GET_STATUS, USB_DIR_IN | type, 0, target,
status,
910                 sizeof(*status), USB_CTRL_GET_TIMEOUT);
911
912         *(u16 *)data = *status;
913         kfree(status);
914         return ret;
    915 }
```

又是一个控制传输,发送的是一个控制请求,GET_STATUS 是 USB 的标准请求之一,

bmRequestType	bRequest	wValue	wIndex	wLength	Data
10000000B 10000001B 10000010B	GET_STATUS	Zero	Zero Interface Endpoint	Two	Device, Interface, or Endpoint Status

这个请求的类型有三种,一种是获得Device的状态,一种是获得Interface的状态,一种是获得端点的状态,这里咱们传递的是 USB_RECIP_DEVICE,也就是获得 Device 的状态.那么函数返回之后,Device 的状态就被保存在了 hubstatus 里面了.

763 至 788 行,就是一个 if/elseif/else 组合.首先 if 判断当年这个 device 是不是 root hub,如果是 root hub,然后判断 hdev->bus_mA,这个值是在 host controller 的驱动程序中设置的,通常来讲,计算机的 usb 端口可以提供 500mA 的电流,不过 host controller 那边有一个成员 power_budget,在 host controller 的驱动程序中,root hub 的 hdev->bus_mA 被设置为 500mA与power_budget中较小的那一个,即如果你有兴趣看一下drivers/usb/core/hcd.c,你会注意到在 usb_add_hcd 这个函数中有这么两行,

```
1637        /* starting here, usbcore will pay attention to this root hub */
1638        rhdev->bus_mA = min(500u, hcd->power_budget);
```

power_budget 是一个 host controller 自己提供的,它可以是 0,如果是 0,表示没有限制.所以我们这里判断是不是等于 0,或者是不是大于等于 500mA,如果是的,那么就设置 hub->mA_per_port 为 500mA,mA_per_port 就是提供给每一个 port 的电流.那么如果说 bus_mA 是0到500之间的某个数,那么说明这个hub没法提供达到500mA的电流,就是host controller 那边提供不了这么大的电流,那么 hub->mA_per_port 就设置为 hdev->bus_mA,这种小市民的想法很简单,钱多就多花点,钱少就少花点,没钱就不花.同时,对于这种 host controller 那边限制了电流的情况,记录下来, hub->limited_power 这么一个标志位设置为 1.

那么如果不是 root hub 呢,又有两种情况, USB_DEVICE_SELF_POWERED,hubstatus 里的这一位表征这个 hub 是不是自己供电的,因为外接的 hub 也有两种供电方式,自己供电或者选择吃大锅饭的形式,即请求总线供电.770行如果满足,那就说明这又是一个要总线供电的hub,于是 limited_power 也设置为 1.774 行,maxchild>0 那简直是一定的.然后定义了一个 int 变量 remaining 来记录剩下多少电流,hdev->bus_mA 就是这个 hub(不是 root hub)上行口的电流,而 bHubContrCurrent 我们说过了,就是 hub 需要的电流.两者相减就是剩下的.但是在 usb 端口上,最小的电流负载就是 100mA,这个叫做单元负载(unit load),这个我还是比较清楚的,怎么说大学四年第一次考进系里前 5 名的课程就是模拟电子线路的期中考试,至今还记得,88 分.不吹了,继续,所以 778 行的意思很显然,比如你这个 hub 有四个口,即 maxchild 为 4,那么你最起码你得剩下个400mA电流吧,因为如果某个端口电流小于100mA的话,设备是没法正常工作的.然后,782 行,警告归警告,最终还是设置 mA_per_port 为 100mA.

784 行,如果是自己供电的那种 hub,那没得说,直接设置为 500mA 吧.

793 行,hub_hub_status(),这个函数还是来自 drivers/usb/core/hub.c:

```
531 static int hub_hub_status(struct usb_hub *hub,
532                 u16 *status, u16 *change)
533 {
```

```
534         int ret;
535
536         mutex_lock(&hub->status_mutex);
537         ret = get_hub_status(hub->hdev, &hub->status->hub);
538         if (ret < 0)
539                 dev_err (hub->intfdev,
540                         "%s failed (err = %d)\n", __FUNCTION__, ret);
541         else {
542                 *status = le16_to_cpu(hub->status->hub.wHubStatus);
543                 *change = le16_to_cpu(hub->status->hub.wHubChange);
544                 ret = 0;
545         }
546         mutex_unlock(&hub->status_mutex);
547         return ret;
548 }
```

和刚才那个 get_hub_status 不一样的是,刚才那个 GET_STATUS 是标准的 usb 设备请求,每个设备都会有的,但是现在这个请求是 hub 自己定义的,其格式如下图所示:

bmRequestType	bRequest	wValue	wIndex	wLength	Data
10100000B	GET_ STATUS	Zero	Zero	Four	Hub Status and Change Status

最后状态保存在 status 和 change 里面,即 hubstatus 和 hubchange 里面.而函数 hub_hub_status 的返回值正常就是 0,否则就是负的错误码.

799 至 807 行都是打印调试信息.飘过.

809 行开始就是真正的干正经事了.我们知道 usb-storage 里面最常见的传输方式就是 control/bulk 传输,而对于 hub,它的传输方式就是 control/interrupt,而最有特色的正是它的中断传输.注意咱们在调用 hub_configure 的时候传递进来的第二个参数是 endpoint,前面我们已经说了,这正是代表着 hub 的中断端点,所以 815 行,一路走过来的兄弟们应该一眼就能看出这一行就是获得连接 host 与这个端点的这条管道,这条中断传输的管道.不过注意,hub 里面的中断端点一定是 IN 的而不是 OUT 的.别问我,usb spec 就这么规定的,我想改人家也不同意.

816 行,usb_maxpacket 我们倒是第一次遇见,其实它就是获得一个 endpoint 描述符里面的 wMaxPacketSize,赋给 maxp,一个端点一次最多传输的数据就是 wMaxPacketSize.不可以超过它.咱们前面为 hub->buffer 申请了内存,这里 maxp 如果大于这个 size,那么不可以,就让它等于 hub->buffer 的 size.

然后 821 行开始就是老套路了,申请一个 urb,然后填充一个 urb,usb_alloc_urb()不用多说,usb_fill_int_urb()可以看一下,来自 include/linux/usb.h:

```
1224 /**
1225  * usb_fill_int_urb - macro to help initialize a interrupt urb
1226  * @urb: pointer to the urb to initialize.
1227  * @dev: pointer to the struct usb_device for this urb.
```

```
1228  * @pipe: the endpoint pipe
1229  * @transfer_buffer: pointer to the transfer buffer
1230  * @buffer_length: length of the transfer buffer
1231  * @complete_fn: pointer to the usb_complete_t function
1232  * @context: what to set the urb context to.
1233  * @interval: what to set the urb interval to, encoded like
1234  *      the endpoint descriptor's bInterval value.
1235  *
1236  * Initializes a interrupt urb with the proper information needed to submit
1237  * it to a device.
1238  * Note that high speed interrupt endpoints use a logarithmic encoding of
1239  * the endpoint interval, and express polling intervals in microframes
1240  * (eight per millisecond) rather than in frames (one per millisecond).
1241  */
1242 static inline void usb_fill_int_urb (struct urb *urb,
1243                                      struct usb_device *dev,
1244                                      unsigned int pipe,
1245                                      void *transfer_buffer,
1246                                      int buffer_length,
1247                                      usb_complete_t complete_fn,
1248                                      void *context,
1249                                      int interval)
1250 {
1251         spin_lock_init(&urb->lock);
1252         urb->dev = dev;
1253         urb->pipe = pipe;
1254         urb->transfer_buffer = transfer_buffer;
1255         urb->transfer_buffer_length = buffer_length;
1256         urb->complete = complete_fn;
1257         urb->context = context;
1258         if (dev->speed == USB_SPEED_HIGH)
1259                 urb->interval = 1 << (interval - 1);
1260         else
1261                 urb->interval = interval;
1262         urb->start_frame = -1;
1263 }
```

对比一下形参和实参,重点关注这么几项,transfer_buffer 就是 hub->buffer,transfer_buffer_length 就是 maxp,complete 就是 hub_irq,context 被赋为了 hub,而 interval 被赋为 endpoint->bInterval,不过这里做了一次判断,如果是高速设备(高速 hub),那么 interval 就等于多少多少,否则 interval 又等于多少多少.至于为什么不一样,其实是因为单位不一样,早年,提到 usb 协议,人们会提到 frame,即帧,改革开放之后,出现了一个新的名词,叫做微帧,即 microframe.一个帧是 1 毫秒,而一个微帧是八分之一毫秒,也就是 125 微秒.要知道我们刚才说了,这里传递进来的 interval 实际上是 endpoint->bInterval,要深刻认识这段代码背后的哲学意义,需要知道 usb 中断传输究竟是怎么进行的.

While You Were Sleeping(四)

我们说过,hub 里面的中断端点是 IN 的,不是 OUT 的.但这并不说明凡是中断传输数据一定是从设备到主机,没这种说法,别起哄.不过 hub 需要的确实只是 IN 的传输.首先,每一个男人都应该知道,中断是由设备产生的.在 usb 的世界里两个重要角色,主机,设备.主机就像一个人民公仆,设备就像人民群众,公仆常常日理万机,或者日理万鸡,他也许不会去理会每一个子民的水深火热,群众如果要想引起公仆的注意,只能做一些有创意的事情,他们知道,像许茹芸唱的那样,"没有星星的夜里,我用泪光吸引你",其实是没有意义的,他们知道只有一些诸如山西黑砖窑事件,如无锡绿藻泛滥事件,如九江大桥坍塌事件,如唐山的黑军车事件,如安徽的粽子事件,才是有意义的,一次事件就可以引发公仆的一次中断,会引发一些事情(数据传输).

那么什么是 interval 呢?interval 就是间隔期,啥叫间隔期?首先你要明白,中断传输绝对不是一种周期性的传输,你说黑砖窑事件会不会一个月来一次?绿藻泛滥事件会不会?黑军车事件会不会?肯定不会对不对.真要是每个月总有几天发生这些事件,那大家都别活了,也甭买安尔乐护舒宝了.那为什么还要有一个间隔期呢?实际上是这样的,尽管中断本身不会定期发生,但是有一个事情是周期性的,对于 IN 的中断端点,主机会定期向设备问寒问暖,就好比一个父母官定期微服私访,去民间了解民情,那么如果人民生活很艰苦,饱受通货膨胀之苦,买不起房子,人民哭爹叫娘的,父母官就会知道,然后就会进行处理,会想办法去拯救万民于水火之中,会严惩万恶的房地产商.只要每一个当官的都能这样做,何愁天下不太平,人民不安居不乐业.

那么具体来讲,在 usb 的世界里,体现的是一种设计者们美好的愿望.就是说,设计者们眼看着现实世界中的公仆们都很让人失望,所以在设计 spec 的时候是这么规定的,host 必须要体恤民情,而且,很重要的一点,人民可以提出自己的愿望,比如你希望父母官多久来探望你一次,一个端点的端点描述符里 bInterval 这么一项写的很清楚,就是她渴望的总线访问周期,或者说她期望 host 隔多久就能看望一下她.设备要进行中断传输,需要提交一个 urb,里面注明这个探访周期,这算是一种民意,而 host 答应不答应呢,这个在 host controller 的驱动程序里才知道,我们不管,host 当然有一个标准,中断传输可以用在低速/全速/高速设备中,而高速传输可以接受在每个微帧有多达 80%的时间是进行定期传输,(包括中断传输和等时传输),而全速/低速传输则可以接受每个帧最多有 90%的时间来进行定期传输.所以呢,host 那边会统一来安排,他会听取每一个群众(设备)的意见或者说愿望,然后他来统一安排,日程安排得过来就给安排,安排不过来那么就返回错误.这些都是 host controller 驱动做的,暂且不表.那么如果说 host 同意,或者说 host controller 接受了群众的意见,比如你告诉他你希望他每周来看你一次,那好,他每周来一次,来问候你一下,问你有没有什么困难(中断).

从技术的角度来讲,就是,host controller 定期给你发送一个 IN token,就是说发送这么一个包,而你如果有中断等在那里,那你就告诉她,你有了(中断).同时你会把与这个中断相关的数据发送给主机,这就是中断传输的数据阶段,显然,这就是 IN 方向的传输.然后主机接收到数据之后他会发送回来一个包,向你确认一下.那么如果 host controller 发送给你一个 IN token 的时候,你没有中断,那怎么办呢?你还是要回应一声,说你没有.当然还有另一种情况是,你可能人不在家,你出差去了,那么你可以在家留个条,告诉人家你没法回答.以上三种情况,对应专业术语来说就是,第一种,你回应的是 DATA,主机回应的是 ACK 或者是 Error.第二种,你回应的是 NAK,第三种,你回应的是 STALL.咱们别玩深沉的,没必要说这些专业术语.就这么说吧,用一段电话对白来解释中断传输的过程中,设备需要做些什么:

对话一: 问:复旦有处女吗? 答:有.在女生肚子里. (DATA)

对话二: 问:复旦有处女吗? 答:没有. (NAK)

对话三: 问:复旦有处女吗? 答:对不起,您所呼叫的用户不在服务区,暂时无法接通. (STALL)

三种情况,你都会有回应,但是意义明显不同.

顺便解释一下 OUT 类型的中断端点是如何进行数据传输,虽然 Hub 里根本就没有这种款式的端点.分三步走,第一步,host 发送一个 OUT token 包,然后第二步就直接发送数据包,第三步,设备回应,也许回应 ACK,表示成功接收到了数据,也许回应 NAK,表示失败了,也许回应 STALL,表示设备端点本身有问题,传输没法进行.

Ok,现在可以回到刚才那个 interval 的问题来了,因为不同速度的 interval 的单位不一样,所以同样一个数字表达的意思也不一样,那么对于高速设备来说,比如它的端点的 bInterval 的值为 n,那么这表示它渴望的周期是 2 的(n-1)次方个微帧,比如 n 为 4,那么就表示 2 的 3 次方个微帧,即 8 个 125 微秒,换句话说就是 1 毫秒.对于高速设备来说,spec 里规定,n 的取值必须在 1 到 16 之间,而对于全设备来说,其渴望周期在 spec 里有规定,必须是在 1 毫秒到 255 毫秒之间,对于低速设备来说,其渴望周期必须在 10 毫秒到 255 毫秒之间.可见,对于全速/低速设备来说,不存在这种指数关系,所以 urb->interval 直接被赋值为 bInterval,而高速设备由于这种指数关系,bInterval 的含义就不是那么直接,而是表示那个幂指数.而 start_frame 是专门给等时传输用的,所以我们不用管了,这里当然直接设置为-1 即可.

终于,我们明白了这个中断传输,明白了这个 usb_fill_int_urb()函数,于是我们再次回到 hub_configure()函数中来,830 和 831 行,这个没什么好说的,usb-storage 里面也就这么设的,能用 DMA 传输当然要用 DMA 传输.

834 行,has_indicators 不用说了,刚刚才介绍的,有就设置为了 1,没有就是 0,不过 hub 驱动里提供了一个参数,叫做 blinkenlights,指示灯这东西有两种特性,一个是亮,一个是闪,我们常说的一闪一闪亮晶晶,有灯了,亮了,但是不一定会闪,所以 blinkenlights 就表示闪不闪,这个参数可以在我们加载模块的时候设置,默认值是 0,在 drivers/usb/core/hub.c 中有定义:

```
90 /* cycle leds on hubs that aren't blinking for attention */
91 static int blinkenlights = 0;
92 module_param (blinkenlights, bool, S_IRUGO);
93 MODULE_PARM_DESC (blinkenlights, "true to cycle leds on hubs");
```

以上都是和模块参数有关的.如果这两个条件都满足,就设置 hub->indicator [0]为 INDICATOR_CYCLE.这么设置有什么用,咱们以后会看到.不过老实说,指示灯这东西怎么工作根本不是我们感兴趣的,作为一个有志青年我们更应该关心关心国家大事,关心关心巴以冲突,关心关心韩国人质,哪有闲功夫去理睬这种指示灯的事呢.

837 行,hub_power_on(),这个函数的意图司马昭之心,路人皆知.无非就是相当于打开电视机开关.不过这里涉及到一个重要的函数,暂且不细讲,稍后会专门讲.

838 行, hub_activate().在讲这个函数之前,先看一下 hub_configure()中剩下的最后几行,hub_activate()之后就 return 0,返回了.841 行的 fail 是行标,之前那些出错的地方都有 goto fail 语句跳转过来,而且错误码也记录在了 ret 里面,于是返回 ret.好,让我们来看一下 hub_activate().这个函数不长,依然来自 drivers/usb/core/hub.c:

```
514 static void hub_activate(struct usb_hub *hub)
515 {
```

```
516         int     status;
517
518         hub->quiescing = 0;
519         hub->activating = 1;
520
521         status = usb_submit_urb(hub->urb, GFP_NOIO);
522         if (status < 0)
523                 dev_err(hub->intfdev, "activate --> %d\n", status);
524         if (hub->has_indicators && blinkenlights)
525                 schedule_delayed_work(&hub->leds, LED_CYCLE_PERIOD);
526
527         /* scan all ports ASAP */
528         kick_khubd(hub);
529 }
```

quiescing 和 activating 就是两个标志符.activating 这个标志的意思不言自明,而 quiescing 这个标志的意思就容易让人疑惑了,永垂不朽的国产软件金山词霸告诉我们,quiescing 是停顿,停止,停息的意思,咋一看 activating 和 quiescing 就是一对反义词,可是如果真的是起相反的作用,那么一个变量不就够了么?可以为 1,可以为 0,不就表达了两种意思了吗?嗯,套用小龙人的歌词,就不告诉你,就不告诉你.至少现在不用告诉你,后面用上了再说.

512 行,这个我们太熟悉了,非常熟悉,相当熟悉.前面我们调用 usb_fill_int_urb()填充好了一个 urb,这会儿就该提交了,然后 host controller 就知道了,然后如果一切顺利的话,host controller 就会定期来询问 hub,问它有没有中断,有的话就进行中断传输,这个我们前面讲过了.

524 行,又和刚才一样的判断,不过这次判断条件满足了以后就会执行一个函数,schedule_delayed_work(),终于看到这个函数被调用了,不枉费我们前面专门分析的蝴蝶效应吧,前面分析清楚了,这里一看我们就知道,延时调用,调用的是 leds 对应的那个 work 函数,即我们当初注册的那个 led_work().这里 LED_CYCLE_PERIOD 就是一个宏,表明延时多久,这个宏在 drivers/usb/core/hub.c 里定义好了,

```
208 #define LED_CYCLE_PERIOD        ((2*HZ)/3)
```

关于这个指示灯的代码我们以后再分析,趁着年轻,我们应该赶紧把该做的事情做了,像指示灯相关的代码以后再看也不迟,我们真正需要花时间关注的是 hub 作为一个特殊的 usb 设备它是如何扮演好连接主机和设备的作用的.席慕蓉说:这个世界上有许多事情,你以为明天一定可以再继续做的,有很多人,你以为一定可以再见到面的.于是,在你暂时放下手,或者暂时转过身的时候,你心中所想的,只是明日又将重聚的希望.有时候,甚至连这种希望都感觉不到.因为,你以为日子既然这样一天一天过来,当然也应该就这样一天一天过去.昨天,今天,明天应该是没有什么不同的.但是,就会有那么一次,在你一放手,一转身的一刹那,有的事情就完全改变了.太阳落下去,而在它重新开始以前,有些人有些事就从此和你永别.

唉,怎么说着说着这么伤感,继续看,528 行,kick_khubd(hub),来自 drivers/usb/core/hub.c,

```
316 static void kick_khubd(struct usb_hub *hub)
317 {
318         unsigned long    flags;
319
```

```
320         /* Suppress autosuspend until khubd runs */
321         to_usb_interface(hub->intfdev)->pm_usage_cnt = 1;
322
323         spin_lock_irqsave(&hub_event_lock, flags);
324         if (list_empty(&hub->event_list)) {
325                 list_add_tail(&hub->event_list, &hub_event_list);
326                 wake_up(&khubd_wait);
327         }
328         spin_unlock_irqrestore(&hub_event_lock, flags);
329 }
```

这才是我们真正期待的一个函数,看见 wake_up()函数了吧,一看到这里就知道怎么回事了吧,先看 321 行,int pm_usage_cnt 是 struct usb_interface 的一个成员,pm 就是电源管理,usage_cnt 就是使用计数,这里的意图很明显,hub 要使用了,就别让电源管理把它给挂起来了,不明白?用过笔记本吧?2005 年进 Intel 的时候每人发一本 IBM T42,我发现很多时候我合上笔记本它就会自动进入休眠,可是有时候我会发现我合上笔记本以后,笔记本并没有休眠,后来总结出来规律了,下 A 片的时候基本上计算机是不会进入睡眠的,理由很简单,它发现你有活动着的网络线程,那时候用的是 Windows XP,当然 Windows 也知道计数,别把人家想的那么傻.不过,这里,我们计数的目的不是记录网络线程,而是告诉 usb core,咱们拒绝自动挂起,具体的处理会由 usb core 来统一操作,不用咱们瞎操心,usb core 那边自然会判断 pm_usage_cnt 的,只要我们这里设置了就可以了.

324 到 327 行这段 if 判断语句,很显然,现在我们是第一次来到这里, 不用说,hub->event_list 是空的,所以,条件满足,于是 list_add_tail()会被执行,关于队列操作函数,咱们也讲过了,所以这里也不用困惑了,就是往那个总的队列 hub_event_list 里面加入 hub->event_list,然后调用 wake_up(&khubd_wait)去唤醒那个昏睡了 N 多年的 hub_thread().如我们前面说过的那样,从此 hub_events()函数将再次被执行.

323 行和 328 行,关于自旋锁,老规矩,放到最后面去讲.统一讲.

至此为止,整个关于 hub 的配置就讲完了,从现在开始,hub 就可以正式上班了.而我们也终于完成了一个目标,唤醒了那个该死的 hub_thread(),进入 hub_events().

再向虎山行

徐志摩说:轻轻的我穿衣,正如我轻轻的脱;

后来徐志摩又说:轻轻的我走了,正如我轻轻的来.

hub_events(),没错,胡汉三又回来了.

再一次进入 while 这个(该)死(的)循环.

第一次来这里的时候,hub_event_list 是空的,可是这一次不是了,我们刚刚在 kick_khubd()里面才执行了往这个队列里插入的操作,所以我们不会再像第一次一样,从 2621 行的 break 跳出循环.相反,我们直接走到 2624 行,把刚才插入队列的那个节点取出来,存为 tmp,然后把 tmp 从队列里删除掉.(是从队列里删除,不是把 tmp 本身给删除.)

2627 行,list_entry(),这个经典的函数,或者说宏,就像复旦南区食堂的大排,永恒的经典.通过这个宏这里得到的是那个触发 hub_events()的 hub.然后 2628 行,同时用局部变量 hdev 记录 hub->hdev.2629 行,又得到对应的 struct usb_interface 和 struct device,这下好了,什么都得到了,该付出了吧.

2640 行, usb_get_intf(),看仔细了,别和我们当年在 usb-storage 里面调用的那个 usb_get_intfdata()混淆了,这里 usb_get_intf 只是一个引用计数.是 usb core 提供的一个函数,以前黑客们推荐用另一个引用计数的函数 usb_get_dev(),但在当今世界这样一种现状下,随着一个 usb device 越来越成为多个 interface 耦合的情况的出现,struct usb_device 实际上已经快淡出历史舞台了,现在在驱动程序里关注的最多的就是 interface,而不是 device.和 usb_get_intf()对应的有另一个函数,叫做 usb_put_intf(),很显然,一个是增加引用计数一个减少引用计数.这个函数我们马上就能看到.

前面我们贴 hub_events()只贴到 2641 行,现在继续贴,贴完这个粉恐怖的函数.

```
2642
2643                /* Lock the device, then check to see if we were
2644                 * disconnected while waiting for the lock to succeed. */
2645                if (locktree(hdev) < 0) {
2646                        usb_put_intf(intf);
2647                        continue;
2648                }
2649                if (hub != usb_get_intfdata(intf))
2650                        goto loop;
2651
2652                /* If the hub has died, clean up after it */
2653                if (hdev->state == USB_STATE_NOTATTACHED) {
2654                        hub->error = -ENODEV;
2655                        hub_pre_reset(intf);
2656                        goto loop;
2657                }
2658
2659                /* Autoresume */
2660                ret = usb_autopm_get_interface(intf);
2661                if (ret) {
2662                        dev_dbg(hub_dev, "Can't autoresume: %d\n", ret);
2663                        goto loop;
2664                }
2665
2666                /* If this is an inactive hub, do nothing */
2667                if (hub->quiescing)
2668                        goto loop_autopm;
2669
2670                if (hub->error) {
2671                        dev_dbg (hub_dev, "resetting for error %d\n",
2672                                hub->error);
```

```
2673
2674                        ret = usb_reset_composite_device(hdev, intf);
2675                        if (ret) {
2676                                dev_dbg (hub_dev,
2677                                        "error resetting hub: %d\n", ret);
2678                                goto loop_autopm;
2679                        }
2680
2681                        hub->nerrors = 0;
2682                        hub->error = 0;
2683                }
2684
2685                /* deal with port status changes */
2686                for (i = 1; i <= hub->descriptor->bNbrPorts; i++) {
2687                        if (test_bit(i, hub->busy_bits))
2688                                continue;
2689                        connect_change = test_bit(i, hub->change_bits);
2690                        if (!test_and_clear_bit(i, hub->event_bits) &&
2691                                                        !connect_change
&& !hub->activating)
2692                                continue;
2693
2694                        ret = hub_port_status(hub, i,
2695                                        &portstatus, &portchange);
2696                        if (ret < 0)
2697                                continue;
2698
2699                        if (hub->activating && !hdev->children[i-1] &&
2700                                        (portstatus &
2701
USB_PORT_STAT_CONNECTION))
2702                                connect_change = 1;
2703
2704                        if (portchange & USB_PORT_STAT_C_CONNECTION)
{
2705                                clear_port_feature(hdev, i,
2706                                        USB_PORT_FEAT_C_CONNECTION);
2707                                connect_change = 1;
2708                        }
2709
2710                        if (portchange & USB_PORT_STAT_C_ENABLE) {
2711                                if (!connect_change)
2712                                        dev_dbg (hub_dev,
2713                                                "port %d enable change, "
```

```
2714                                    "status %08x\n",
2715                                    i, portstatus);
2716                            clear_port_feature(hdev, i,
2717                                    USB_PORT_FEAT_C_ENABLE);
2718
2719                            /*
2720                             * EM interference sometimes causes badly
2721                             * shielded USB devices to be shutdown by
2722                             * the hub, this hack enables them again.
2723                             * Works at least with mouse driver.
2724                             */
2725                            if (!(portstatus & USB_PORT_STAT_ENABLE)
2726                                && !connect_change
2727                                && hdev->children[i-1]) {
2728                                    dev_err (hub_dev,
2729                                        "port %i "
2730                                        "disabled by hub (EMI?), "
2731                                        "re-enabling...\n",
2732                                            i);
2733                                    connect_change = 1;
2734                            }
2735                    }
2736
2737                    if (portchange & USB_PORT_STAT_C_SUSPEND) {
2738                            clear_port_feature(hdev, i,
2739                                    USB_PORT_FEAT_C_SUSPEND);
2740                            if (hdev->children[i-1]) {
2741                                    ret = remote_wakeup(hdev->
2742                                                    children[i-1]);
2743                                    if (ret < 0)
2744                                            connect_change = 1;
2745                            } else {
2746                                    ret = -ENODEV;
2747                                    hub_port_disable(hub, i, 1);
2748                            }
2749                            dev_dbg (hub_dev,
2750                                    "resume on port %d, status %d\n",
2751                                    i, ret);
2752                    }
2753
2754                    if (portchange & USB_PORT_STAT_C_OVERCURRENT)
{
2755                            dev_err (hub_dev,
2756                                    "over-current change on port %d\n",
```

```
2757                                    i);
2758                            clear_port_feature(hdev, i,
2759
USB_PORT_FEAT_C_OVER_CURRENT);
2760                            hub_power_on(hub);
2761                    }
2762
2763                    if (portchange & USB_PORT_STAT_C_RESET) {
2764                            dev_dbg (hub_dev,
2765                                    "reset change on port %d\n",
2766                                    i);
2767                            clear_port_feature(hdev, i,
2768                                    USB_PORT_FEAT_C_RESET);
2769                    }
2770
2771                    if (connect_change)
2772                            hub_port_connect_change(hub, i,
2773                                            portstatus, portchange);
2774            } /* end for i */
2775
2776            /* deal with hub status changes */
2777            if (test_and_clear_bit(0, hub->event_bits) == 0)
2778                    ;       /* do nothing */
2779            else if (hub_hub_status(hub, &hubstatus, &hubchange) < 0)
2780                    dev_err (hub_dev, "get_hub_status failed\n");
2781            else {
2782                    if (hubchange & HUB_CHANGE_LOCAL_POWER) {
2783                            dev_dbg (hub_dev, "power change\n");
2784                                    clear_hub_feature(hdev,
C_HUB_LOCAL_POWER);
2785                                            if (hubstatus &
HUB_STATUS_LOCAL_POWER)
2786                                    /* FIXME: Is this always true? */
2787                                    hub->limited_power = 0;
2788                            else
2789                                    hub->limited_power = 1;
2790                    }
2791                    if (hubchange & HUB_CHANGE_OVERCURRENT) {
2792                                    dev_dbg (hub_dev, "overcurrent
change\n");
2793                            msleep(500);    /* Cool down */
2794                                    clear_hub_feature(hdev,
C_HUB_OVER_CURRENT);
2795                            hub_power_on(hub);
```

```
2796                            }
2797                    }
2798
2799                    hub->activating = 0;
2800
2801                    /* If this is a root hub, tell the HCD it's okay to
2802                     * re-enable port-change interrupts now. */
2803                    if (!hdev->parent && !hub->busy_bits[0])
2804                            usb_enable_root_hub_irq(hdev->bus);
2805
2806 loop_autopm:
2807                    /* Allow autosuspend if we're not going to run again */
2808                    if (list_empty(&hub->event_list))
2809                            usb_autopm_enable(intf);
2810 loop:
2811                    usb_unlock_device(hdev);
2812                    usb_put_intf(intf);
2813
2814            } /* end while (1) */
2815 }
```

我真想诚恳的问候一下写代码的人的女性长辈.当裴多菲说:"若为自由故,两者皆可抛",我懂得了作为人的价值;当鲁迅说:"不在沉默中爆发,就在沉默中灭亡",我懂得人应具有反抗精神;当白朗宁说:"拿走爱,世界将变成一座坟墓",我懂得了为他人奉献爱心的重要;当简爱说:"我们是平等的,我不是无感情的机器",我懂得了作为女性的自尊;当我看到这段代码,我却完全不懂写代码的人为什么总要写一些恐怖的函数出来.一定要吓唬吓唬我他们就开心么?我是复旦的,不是厦(吓)大的.

树,是什么样的树?

同学们,今天我们来讲一棵树.

记得小时候我们看<<白眉大侠>>,记得那段精彩的对白:刀,是什么样的刀?金丝大环刀!剑,是什么样的剑?闭月羞光剑!招,是什么样的招?天地阴阳招!人,是什么样的人?飞檐走壁的人!情,是什么样的情?美女爱英雄!

而今天我们要问的是:树,是什么样的树?答:USB 设备树.这是怎样一棵树?让我慢慢的道来.

苏格拉底曾经说过:为人不识谭浩强,精通内核也枉然.

还记得谭浩强大哥书中那个汉诺塔的例子么?那时候我天真的以为这个例子没什么意义,可是今天我终于明白了.谭大哥早就知道我们学了他的书一定会在今后的生活工作中用得到,这不,hub_events()里面第 2645 行,locktree(),用的就是汉诺塔里的那个经典思想,递归.谭大哥

您要是穿裙子的话,我一定第一个拜倒在您的石榴裙下!locktree()定义于drivers/usb/core/hub.c:

```
985 /* grab device/port lock, returning index of that port (zero based).
986  * protects the upstream link used by this device from concurrent
987  * tree operations like suspend, resume, reset, and disconnect, which
988  * apply to everything downstream of a given port.
989  */
990 static int locktree(struct usb_device *udev)
991 {
992         int                     t;
993         struct usb_device       *hdev;
994
995         if (!udev)
996                 return -ENODEV;
997
998         /* root hub is always the first lock in the series */
999         hdev = udev->parent;
1000         if (!hdev) {
1001                 usb_lock_device(udev);
1002                 return 0;
1003         }
1004
1005         /* on the path from root to us, lock everything from
1006          * top down, dropping parent locks when not needed
1007          */
1008         t = locktree(hdev);
1009         if (t < 0)
1010                 return t;
1011
1012         /* everything is fail-fast once disconnect
1013          * processing starts
1014          */
1015         if (udev->state == USB_STATE_NOTATTACHED) {
1016                 usb_unlock_device(hdev);
1017                 return -ENODEV;
1018         }
1019
1020         /* when everyone grabs locks top->bottom,
1021          * non-overlapping work may be concurrent
1022          */
1023         usb_lock_device(udev);
1024         usb_unlock_device(hdev);
1025         return udev->portnum;
```

1026 }

咱们传递进来的是咱们这个 hub 对应的 struct usb_device 指针,995 行自然不必说,就是防止那些唯恐天下不乱的人调用这个函数.

999 行,parent,struct usb_device 结构体的 parent 自然也是一个 struct usb_device 指针.1000 行,判断 udev 的 parent 指针,你一定觉得奇怪,好像之前从来没有看到过 parent 指针啊,为何它突然之间出现了?它指向什么呀?对此我只能说抱歉,其实生活中发生的一切,都是那么的突然.其实我也没有办法,Hub 驱动作为一个驱动程序,它并非是孤立存在的,没有主机控制器的驱动,没有 usb core,Hub 驱动的存在将没有任何意义.其实我以前就说过,Hub,准确的说应该是说,Root Hub,它和 Host Controller 是绑定在一起的,专业一点说叫做集成在一起的,所以它根本不会像我一样除了拥有孤独其它的一无所有.有时候我真的想知道,当全世界孤独的人团结在一起,孤独是会加深,还是消失...

因为 Hub 驱动不孤立,所以具体来说,作为 Root Hub,它的 parent 指针在 Host Controller 的驱动程序中就已经赋了值,这个值就是 NULL,换句话说,对于 Root Hub,它不需要再有父指针了,这个父指针本来就是给从 Root Hub 连出来的节点用的,让我们打开天窗说亮话,这里这个函数名字叫做 locktree,这个意思就很直接了,locktree 嘛,顾名思义,锁住一棵树,这棵树就是 USB 设备树.很显然,USB 设备是从 Root Hub 开始,一个一个往外面连的,比如 Root Hub 有 4 个口,每个口连一个 USB 设备,比如其中有一个是 Hub,那么这个 Hub 有可以继续有多个口,于是一级一级的往下连,最终肯定会连成一棵树,八卦两句,自从某年某月某一天,我家 Intel 提出了 EHCI 的规范以来,当今 USB 世界的发展趋势是,硬件厂商们总是让 EHCI 主机控制器里面拥有尽可能多的端口,换言之,理想情况就是希望大家别再用外接的 Hub 了,有一个 Root Hub 就够用了,也就是说,真的到了那种情况,USB 设备树的就不太像树了,顶多就是两级,一级是 Root Hub,下一级就是普通设备,严格来说,对于咱们普通人来说,这样子也就够用了,假设你的 Root Hub 有 8 个口,你说你够用不够用?鼠标,键盘,音响,U 盘,存储卡,还有啥啊?8 个口对正常人来说肯定够了,当然你要是心理变态那么另当别论,说不准就有人愿意往一台电脑里插入个三五十个 USB 设备的,林子大了,什么鸟没有?

所以说写代码也不是一件容易的事情,除了保证你的代码能让正常人正常使用,还得保证变态也能使用.locktree()的想法就是这样,在 hub_events()里面加入 locktree()的理由很简单,如果你的电脑里有两个 hub,一个叫 parent 一个叫 child,child 接在 parent 的某个口上,那么 parent 在执行下面这段代码的时候,child hub 就不要去执行这段代码,否则会引起混乱,为何会引起混乱?要知道,对于一个 hub 来说,其所有正常的工作都是在 hub_events()这个函数里进行的,那么这些工作就比如,一种情况是,删除一个子设备, 这将有可能直接导致 USB 设备树的拓扑结构发生变化,或者另一种情况,遍历整个子树去执行一个 resume 或者 reset 之类的操作,那么很显然,在这种情况下,一个 parent hub 在进行这些操作的时候,不希望受到 child hub 的影响,所以在这样一个政治背景下,2004 年的夏天,作为 Linux 内核开发中 USB 子系统的三剑客之一的 David Brownell 大侠,决定加入 locktree 这么一个函数,这个函数的哲学思想很简单,实际上就是借用了我国古代军事思想中的擒贼仙擒王,用 David Brownell 本人的话说,就是"lock parent first".每一个 Hub 在执行 hub_events()中下面的那些代码时,(特指 locktree 那个括号以下的那些代码.)都得获得一把锁,锁住自己,而在锁住自己之前,又先得获得父亲的锁,确切的说,是尝试获得父亲的锁,如果能够获得父亲的锁,那么说明父亲当前没有执行 hub_events()(因为否则就没有办法获得父亲的锁),那么这种情况下子 hub 才可以执行自己的 hub_events(),但是需要注意,在执行自己的代码之前,先把父 hub 的锁给释放掉,因为我们说了,我们的目的是尝试获得

父亲的锁,这个尝试的目的是为了保证在我们执行 hub_events()之前的那一时刻,父 hub 并不是正在执行 hub_events(),而至于我们已经开始执行了 hub_events(),我们就不在乎父 hub 是否也想开始执行 hub_events()了.

那么有人问,父hub复父hub,父hub何其多?刚才不是说了吗?Root Hub就是整棵树的根,Root Hub 就没有父 hub,所以,整个递归到了父 Hub 就可以终止了.这也正是为什么 1000 行那句 if 语句,在判断出该设备是一个 Root Hub 之后,马上就执行锁住该设备.而如果不是 Root Hub,那么继续往下走,递归调用 locktree(),对于 locktree(),正常情况下它的返回值大于等于 0,所以小于 0 就算出错了.

然后 1015 行判断一下,如果我们把父 hub 锁住了,可是自己却被断开了,即 disconnect 函数被执行了,那么就立刻停止,把父 hub 的锁释放,然后返回吧错误代码-ENODEV.

最后 1023 行,锁住自己,1024 行,释放父设备.

1025 行,返回当前设备的 portnum.portnum 就是端口号,您一定奇怪,没见过什么时候为 portnum 赋值了啊?这就不好意思了,别忘了咱们走到今天这里是在讨论 Root Hub,对于 Root Hub 来说,它本身没有 portnum 这么一个概念,因为它不插在别的 Hub 的任何一个口上.所以对于 Root Hub 来说,它的 portnum 在 Host Controller 的驱动程序里给设置成了 0.而对于普通的 Hub,它的 portnum 在哪里赋的值呢?我们后面就会看到的.别急.不过友情提醒一下,对于 Root Hub 来说,这里根本就不会执行到 1025 行来,刚才说了,对于 Root Hub,实际上 1002 行那里就返回了,而且返回值就是 0.

就这样,我们看完了 locktree()这个函数,接下来我们又该返回到 hub_events()里面去了.再次爆料一下,2004 年,当时的内核还只是 2.6.8,David Brownell 大侠在那个夏天提出来 locktree()的时候,遭到了另一位同是三剑客之一的 Alan Stern 大侠强烈反对,当时 Alan Stern 认为 David Brownell 的算法并不好,没能真正解决并发事件的相互干扰问题,Alan Stern 自己有他的算法,于是两个人各自坚持自己的意见,在开源社区轰轰烈烈争论了 N 久,差点没打起来.那个夏天这两个人从 7 月 30 日一直吵到了 8 月 18 日,要不是 8 月 15 日我在珠江电影制片厂外景地看张柏芝拍大运摩托的那个广告,我说不准就会上去劝架,不过像我这样的人去劝架估计人家也不会理睬,人家会问:你丫谁呀?

最终 locktree()被加入到了内核里面,而且不止加了一处,除了加入到了 hub_events()之外,在另外两个函数 usb_suspend_device()和 usb_resume_device()里面也有调用 locktree().有趣的是,从 2.6.9 一直到 2.6.22.1 的内核,我们都能看到 locktree()这么一个函数,但是后来,usb_suspend_device/usb_resume_device 中没有了这个函数,就比如我们现在看到的 2.6.22.1,搜索整个内核,只有 hub_events()这一个地方调用了 locktree(),对此,Alan Stern 给出的说法是,内核中关于 USB 挂起的支持有了新的改进,不需要再调用 locktree 了.但是从 2.6.23 的内核开始,估计整个内核中就不会再有 locktree 了,Alan Stern 大侠在这个夏天,提交了一个 patch,最终把 locktree 相关的代码全部从内核中移除了.如果您对锁机制特别感兴趣,那么研究一下 Linux 2.6 以来 Hub 驱动的历史发展,这个过程锁机制的变更,或者专业一点说,叫同步机制算法的不断改进,足以让你写一篇能在国内核心刊物发表的文章来,要知道首都有一所叫做清华的名牌大学,其硕士研究生也就是一篇国内核心就可以毕业了.

没完没了的判断

看着这代码,空虚的代码,麻木的走在崩溃边缘.最讨厌这种没完没了的判断了.记得有一次在中信泰富广场去摩托罗拉中国研发中心面试,也是问了一道挺简单的题目,我就把基本的算法说了一下,然后面试官就说为什么没有错误判断.你说像我这种根本不怎么懂编程的人好不容易能回答出一道题,已经很不错了,为何那些企业要求都这么高呢?一个人因为没有工作经验而不能得到一个工作,但是他又因为没有一个工作而得不到工作经验.算了,现实充满残忍,何必太认真.算法没错,程序没错,世界没错,我错了.

2645 行我们返回了,前面说了,正常都是返回 0 或者正数,如果小于 0 那就说明失败了,前面我们还说了,我们在使用 interface 之前会调用 usb_get_intf()来增加引用计数,而与之对应的是 usb_put_intf(),这里我们就调用了 usb_put_intf()来减少引用计数.continue 的意思开始新的一轮 while 循环,就是踏上奈何桥,喝下孟婆汤,卸下前世的情仇,忘却今生的爱人,睁开眼又是一生,如果 hub_event_list 里还有东西的话就继续处理,没有那就歇息吧,日子还是像原来那样一天天的过着.

2649 行,usb_get_intfdata(),判断一下,得到的是不是 hub,你问为什么得到的是 hub?回过去看 hub_probe(),别忘了那时候我们调用过 usb_set_intfdata 从而把 intf 和 hub 联系了起来的,这两个函数当年我们在 usb-storage 里面就这样用的,不过当时我们不会判断 usb_get_intfdata()得到的到底是什么,而这里我们却要判断,因为在 hub_disconnect()中,又这么一句,usb_set_intfdata(intf,NULL),而 hub_events()和 hub_disconnect()是异步执行的,就是说你执行你的,我执行我的,换言之,当咱们这里 hub_events()正执行着呢,hub_disconnect()那边可能就已经取消了 intf 和 hub 之间建立起来的那层美好的关系,所以咱们这里需要判断一下.

2653 行,现在是时候该说一说 USB_STATE_NOTATTACHED 这个宏了,在 include/linux/usb/ch9.h 中:

```
557 enum usb_device_state {
558         /* NOTATTACHED isn't in the USB spec, and this state acts
559          * the same as ATTACHED ... but it's clearer this way.
560          */
561         USB_STATE_NOTATTACHED = 0,
562
563         /* chapter 9 and authentication (wireless) device states */
564         USB_STATE_ATTACHED,
565         USB_STATE_POWERED,                  /* wired */
566         USB_STATE_UNAUTHENTICATED,          /* auth */
567         USB_STATE_RECONNECTING,             /* auth */
568         USB_STATE_DEFAULT,                  /* limited function */
569         USB_STATE_ADDRESS,
570         USB_STATE_CONFIGURED,               /* most functions */
571
572         USB_STATE_SUSPENDED
```

```
573
574             /* NOTE:   there are actually four different SUSPENDED
575              * states, returning to POWERED, DEFAULT, ADDRESS, or
576              * CONFIGURED respectively when SOF tokens flow again.
577              */
578 };
```

定义了这么一堆的宏,其中 USB_STATE_NOTATTACHED 的意思很明显,设备没有插在端口上,在代码里,有几个函数会把设备的状态设置成这个,一个是汇报 Host Controller 异常死机的函数,usb_hc_died(),一个是咱们 hub 驱动自己提供的函数,hub_port_disable(),用于关掉一个端口的函数,还有就是用来断开设备的函数 usb_disconnect(),总之这几个函数没一个是好鸟,只要它们执行了,那么咱们的设备肯定就没法工作了,所以这里在干正经事之前,先判断设备的状态是不是 USB_STATE_NOTATTACHED,如果是的那么就设置错误代码为-ENODEV,然后调用 hub_pre_reset(),这个函数是与 reset 相关的,咱们先不理睬.以后再来看.

2660 行,usb_autopm_get_interface(),这个函数是 usb core 提供的,又是一个电源管理的函数,这个函数所做的事情就是让这个 usb interface 的电源引用计数加一,也就是说,只要这个引用计数大于 0,这个设备就不允许 autosuspend.autosuspend 就是当用户在指定的时间内没有什么活动的话,就自动挂起.应该说,在 usb 中引入 autosuspend/autoresume 这还是最近的事情了,最初有这个想法是在去年,2006 年的 5 月底,Alan Stern 大侠在开源社区提出,同志们,俺最近打算开始在 USB 里面实现对 autosuspend/autoresume 的支持.所谓的 autosuspend/autoresume,实际上是一种运行时的电源管理方式.而这些事情将由驱动程序来负责,即,当驱动程序觉得它的设备闲置了,它就会触发 suspend 事件,而当驱动程序要使用一个设备了,但该设备正处于 suspended 状态,那么驱动程序就会触发一个 resume 事件,suspend 事件和 resume 事件很显然是相对应的,一个挂起一个恢复.这里有一个很关键的理念,又涉及到了前面讲的那个设备树,即,当一个设备挂起的时候,它必须通知它的 parent,而 parent 就会决定看 parent 是不是也自动挂起,反过来,如果一个设备需要 resume,那么它必须要求它的 parent 也 resume,就是说这里有这么一种逻辑关系,一个 parent 要想 suspend,只有在它的 children 都 suspend 它才可以 suspend,而一个 child 想要 resume,只有在它的 parent 先 resume 了它才可以 resume.还不明白?举个例子,我和我的室友放寒假在复旦南区澡堂洗澡,每人一个水龙头,洗着洗着,管理员说国家下发了文件说要建设节约型社会,考虑到寒假期间洗澡的人数比较少,决定把水龙头的总闸调小,但是也不能让我们正在洗的人洗不了,所以,它就得先问我们,只有满足了我们在洗的人的那几个水龙头的水量,才可以关小总闸,否则我们肯定得跟他急.同样,开学了以后,大家都来洗澡,可是总闸还是那么小,那大家不干了,但是不干了你得跟管理员说调整总闸,你不能说每个人就调整自己的那个开关,那样肯定没用,总的流量就那么小,时不时还来点侧漏,你说你能洗吗?所以这种情况下就得先开了总闸你单个的开关的调节才有意义.

对于 hub 来说,当 hub_events()处于运行的状态,那么这个 hub interface 就是在使用,这种情况下是不可以进行 autosuspend 的.对于咱们这个上下文来说,usb_autopm_get_interface()返回的就是 0.但是如果咱们的 hub 是处于 suspended 状态,那么这里首先就会把 hub 唤醒,即会执行 resume.先不多说了,继续往下看吧.

2667 行,判断 quiescing,以前咱们说过,struct usb_hub 里面有两个成员,quiescing 和 activiating,并且咱们在 hub_activate()中已经看到了,我们把 quiescing 设置成了 0,而把

activating 设置成了 1.现在是时候来说一说这两个变量的含义了.我们说了 quiescing 是停止的意思,在 reset 的时候我们会设置它为 1,在 suspend 的时候我们也会把它设置为 1,一旦把它设置成了 1,那么 hub 驱动程序就不会再提交任何 URB,而如果我们把 activating,那么 hub 驱动程序就会给每个端口发送一个叫做 Get Port Status 的请求,通常情况下,hub 驱动只有在一个端口发生了状态变化的情况下才会去发送 Get Port Status 从而去获得端口的状态.所以就是说,正常情况下,这两个 flag 都是不会设置的.即正常情况下这两个 flag 都应该是 0.

2671 行,以咱们这个情景来到这里,hub->error 当然是 0,但是如果今后我们正式工作以后,再次来到这里的话,hub->error 可能就不再是 0 了.对于那种情况,咱们需要调用 usb_reset_composite_device(),这个函数是咱们自己定义的,目的就是把设备 reset,我们来具体看一下,来自 drivers/usb/core/hub.c:

```
3041 /**
3042  * usb_reset_composite_device - warn interface drivers and perform a USB
port reset
3043  * @udev: device to reset (not in SUSPENDED or NOTATTACHED state)
3044  * @iface: interface bound to the driver making the request (optional)
3045  *
3046  * Warns all drivers bound to registered interfaces (using their pre_reset
3047  * method), performs the port reset, and then lets the drivers know that
3048  * the reset is over (using their post_reset method).
3049  *
3050  * Return value is the same as for usb_reset_device().
3051  *
3052  * The caller must own the device lock.  For example, it's safe to use
3053  * this from a driver probe() routine after downloading new firmware.
3054  * For calls that might not occur during probe(), drivers should lock
3055  * the device using usb_lock_device_for_reset().
3056  *
3057  * The interface locks are acquired during the pre_reset stage and released
3058  * during the post_reset stage.  However if iface is not NULL and is
3059  * currently being probed, we assume that the caller already owns its
3060  * lock.
3061  */
3062 int usb_reset_composite_device(struct usb_device *udev,
3063                 struct usb_interface *iface)
3064 {
3065         int ret;
3066         struct usb_host_config *config = udev->actconfig;
3067
3068         if (udev->state == USB_STATE_NOTATTACHED ||
3069                         udev->state == USB_STATE_SUSPENDED) {
3070                 dev_dbg(&udev->dev, "device reset not allowed in state
%d\n",
3071                                 udev->state);
```

```
3072                return -EINVAL;
3073        }
3074
3075        /* Prevent autosuspend during the reset */
3076        usb_autoresume_device(udev);
3077
3078        if (iface && iface->condition != USB_INTERFACE_BINDING)
3079                iface = NULL;
3080
3081        if (config) {
3082                int i;
3083                struct usb_interface *cintf;
3084                struct usb_driver *drv;
3085
3086                for (i = 0; i < config->desc.bNumInterfaces; ++i) {
3087                        cintf = config->interface[i];
3088                        if (cintf != iface)
3089                                down(&cintf->dev.sem);
3090                        if (device_is_registered(&cintf->dev) &&
3091                                        cintf->dev.driver) {
3092                                drv = to_usb_driver(cintf->dev.driver);
3093                                if (drv->pre_reset)
3094                                        (drv->pre_reset)(cintf);
3095                        }
3096                }
3097        }
3098
3099        ret = usb_reset_device(udev);
3100
3101        if (config) {
3102                int i;
3103                struct usb_interface *cintf;
3104                struct usb_driver *drv;
3105
3106                for (i = config->desc.bNumInterfaces - 1; i >= 0; --i) {
3107                        cintf = config->interface[i];
3108                        if (device_is_registered(&cintf->dev) &&
3109                                        cintf->dev.driver) {
3110                                drv = to_usb_driver(cintf->dev.driver);
3111                                if (drv->post_reset)
3112                                        (drv->post_reset)(cintf);
3113                        }
3114                        if (cintf != iface)
3115                                up(&cintf->dev.sem);
```

```
3116                    }
3117            }
3118
3119            usb_autosuspend_device(udev);
3120            return ret;
3121 }
```

usb_autoresume_device()增加 device 的引用计数,禁止设备 autosuspend 的发生.

后面的那个 usb_autosuspend_device()则刚好相反,减少设备的引用计数,并且使得设备可以被 autosuspend.

3068 行,在设备处于 USB_STATE_NOTATTACHED 或者 USB_STATE_SUSPENDED 的状态时,reset 是不被允许的.

3078 行,别忘了我们传递给 usb_reset_composite_device 的有两个参数,一个是设备,一个是 interface.而 USB_INTERFACE_BINDING 是一个宏,来自 include/linux/usb.h:

```
83 enum usb_interface_condition {
84          USB_INTERFACE_UNBOUND = 0,
85          USB_INTERFACE_BINDING,
86          USB_INTERFACE_BOUND,
87          USB_INTERFACE_UNBINDING,
88 };
```

这是表征 interface 的状态的,BINDING 就表示正在和 driver 绑定,有些事情不是我故意瞒着你,而是我的确不想多说,确切的说,我也是一言难尽哪.最开始,在 usb core 发现初始化设备的时候,但是在 hub_probe 被调用之前,INTERFACE 是处于 USB_INTERFACE_BINDING 状态的,直到 hub_probe 结束了之后,INTERFACE 则是处于 USB_INTERFACE_BOUND 状态,即所谓的绑定好了,而如果 hub_probe 出了错,那么 INTERFACE 就将处于 USB_INTERFACE_UNBOUND 状态.我们这里 config 是 struct usb_host_config 结构体指针,被赋为 udev->actconfig,对于一个设备来说,它使用的是什么配置,这个在初始化的时候就设好了的.对于 Root Hub 我们先不管它究竟设置为什么了,对于普通的 Hub 我们以后会看到的.struct usb_host_config 结构体中有一个结构体 struct usb_config_descriptor desc,表示 configuration 描述符,还有一个 struct usb_interface *interface[USB_MAXINTERFACES]数组,USB_MAXINTERFACES 定义为 32,这些我们当年在 usb-storage 里面也都见过.所以 config->desc.bNumInterfaces 以及 config->interface[]数组的意思就很明确了,3086 行开始的这个 for 循环就是说,这个 device 有几个 interface 就一个一个遍历,cintf 作为临时变量来表征每一个 struct usb_interface.首先我们要明白, usb_reset_composite_device()这个函数我们可是既指定了设备又指定了 interface 的,那么 3088 行判断 cintf 等不等于 iface 是什么意思呢?回到刚才那个 3078 行,我们设置了,如果 iface 不处在 BINDING 的情况下,我们将 iface 设置为 NULL 了,而这里 cintf 是从 config->interface[]数组里得出来的值,它肯定不为 NULL,那么这里如果 cintf 不等于 iface,那就说明 iface 之前是不处于 BINDING 的状态,对于这种情况我们需要执行 3089 行,down(&cintf->dev.sem),这样做的原因是为了等待,dev 是 struct device 结构体,是

struct usb_interface 结构体的一个成员,而 sem 是 struct device 结构体的一个信号量,struct semaphore sem,这个信号量专门用于同步,之所以这里需要信号量,原因如下:既然我们现在针对的是一个 device 多个 interface 的情况,那么势必就有这么一种可能,一个设备多个 interface,每个 interface 对应一个 driver,那么当设备 fail 的时候,有可能每个 driver 都希望能够 reset,那么对于这种情况,我们当然需要保证这个过程不要出现混乱,于是设置一个信号量就 ok 了,你 reset 的时候我就不能 reset,我 reset 的时候你就不能 reset.那么为什么要单独把USB_INTERFACE_BINDING列出来呢?注释里说得很清楚了,当一个interface的状态处于BINDING的时候,其实就是这个interface对应的driver的probe()函数正在执行,这个时候实际上是已经获得了锁的.你问为什么?看代码去,probe 函数是咱们提供给 usb core 调用的,咱们自己不会调用它,usb core 中,调用 probe()的前后也有这么一对 down()/up().因为 probe()这个操作也忌讳被别人影响.所以说这里对于正处于 BINDING 状态的 interface,就不需要再获得锁了,或者说不需要获得信号量了.否则就将是一道经典的死锁问题,我第一次遇到内核 Bug 就是一次死锁问题,在 8250 串口驱动中,明明已经有锁了,还要再次去获取锁,结果系统就挂了.而这正是锁的哲学,即走别人的路,让别人无路可走.

另一个问题需要清楚的是,usb_reset_composite_device()这个函数的诞生是因为目前越来越多的设备都是复合设备,即一个设备里面有多个 interface 的那种.内核中引入这个函数是在2006 年夏天,在德国世界杯开幕前不久,在我离开 Intel 的那一天,确切的说是,2006 年 5 月 26 日,Alan Stern 大侠又一次向社区里提出一个新的理念.即,原本,对于每个设备来说,都可以调用函数 usb_reset_device 来执行 reset 操作,而当今天下发展趋势是让一个设备包含多个接口,即一个 device 包含多个 interface,而我们知道一个 driver 对应一个 interface,于是就出现了多个 drivers 对应一个设备的情况,那么一个 usb_reset_device()函数就有可能对所有的 interface 都造成影响,于是,Alan 利用这样两个函数,struct usb_driver 结构体中两个成员函数 pre_reset 和 post_reset,我们知道每个 struct usb_driver 都有两个成员 pre_reset 和 post_reset,而 Alan 的理念是每个 driver 定义自己的 pre_reset 和 post_reset,当我们在调用 usb_reset_device 之前,先遍历调用每个 driver 的 pre_reset,Alan 称这些个 pre_reset 为给每个绑定在该设备上的 drivers 一个警告,告诉它们,要 reset 了.在执行完 usb_reset_device 之后,再遍历调用每个 driver 的 post_reset,post_reset 的作用是让每个 driver 知道,reset 完成了,另一方面,post_reset 还有一个作用,因为 reset 会把设备原来的状态都给清除掉,所以 post_reset 就担负了这么一个使命,即重新初始化设备.但是你得明白,并不是每一个设备驱动都定义了 pre_reset 和 post_reset,大家有没有必要执行这两个操作那都是自己决定,你要是无所谓,觉得别人 reset 整个 device 对你这个 interface 没什么影响,那就不为这两个指针赋值,那也 ok.usb core 为你提供了这么一个机制,你用不用自己看着办.这就相当于 Intel 允许报销医药费,但是可能我没有买药的需求,那么我就不用报销.这也就是为什么在 3093 行和 3111 行这两处要判断,判断这两个指针是否被赋了值,只有赋了值才去执行相应的函数,否则就没有必要瞎起哄.

继续看,device_is_registered()是一个内联函数,就是判断 struct device 结构体指针的一个成员 is_registered 是否为 1,这个值对于 Root Hub 来说,在 Host Controller 驱动程序中初始化时把它设置为 1,对于普通的设备来说,以后我们会看到,在 Hub 驱动为其作初始化的时候也会设置为 1.而 dev.driver 也是在初始化的时候会赋好值,特别的,对于咱们的 hub,这个 dev.driver 就是与之对应的 struct device_driver 结构体,而 3092 行这个 to_usb_driver()是一个宏,它得到的就是与之对应的 struct usb_driver 结构体,而对于 hub 来说,这就是 struct usb_driver hub_driver,所以,我们就不难知道,3094 行以及后面 3112 行所做的就是调用 hub_driver 里面的两个成员函数,它们是 hub_pre_reset()和 hub_post_reset().

Ok,暂时打住,越来越偏离主线了.总之,3094 行,3099 行,3012 行,这三个函数的调用,就是真正的完成了一次 hub 的 reset.就相当于计算机的一次重起,重起的原因是因为我们遇见了 hub->error.这三个函数的细节我们先暂时不看,以后再看.需要强调的一点是,3101 到 3117 行这一段,和 3081 到 3097 行这一段,基本上是对称的.

3120 行,return ret 返回了.返回值就是 usb_reset_device 的返回值.

回到 hub_events()之后,立刻把 hub->nerrors 和 hub->error 给复位了,设置为 0,想开心就要舍得伤心,想忘记就要一切归零.其中 nerrors 是记录发生错误的次数的,nerrors 就是 number of errors,要是连续发生错误就每次让 nerrors 加一.

从下面开始就进入 hub 驱动中真正干正经事的代码了.可以说 hub_events()中,此前的每一行代码都显得非常的枯燥,让人根本看不明白 hub 驱动究竟是干嘛的,直到下面这些代码才真正诠释了一个 hub 驱动应该做的事情.我们需要明白,Hub 的存在不是为了它自己,我们不是为了用 Hub 而买 Hub,我们是为了让 Hub 连接我们真正想用的设备.设备是 Hub 生命中的过客,而 Hub 点缀了设备的人生.

一个都不能少

烟波江声里,何处是江南.一晃神,一转眼,我们就这样垂垂老去.可是我们直到现在还根本就没有看明白 hub 驱动究竟是怎么工作的.但我相信,红莲即将绽放,双星终会汇聚,命运的轮转已经开始,我们只需耐心的等待.

2686 行,苦苦追寻之后,终于发现从这里开始针对端口进行分析了,有几个端口就对几个端口进行分析,分析每一个端口的状态变化,一个都不能少,很显然,这就是我们期待看到的代码,马上我们就可以知道,当我们把一个 usb 设备插入 usb 端口之后究竟会发生什么,知道 usb 设备提供的那些接口函数究竟是如何被调用的,特别是那个 probe 函数.这一刻,红领巾迎着太阳,阳光洒在海面上,水中鱼儿望着我们,悄悄的听我们愉快的歌唱,小船儿轻轻飘荡在水中,迎面吹来了凉爽的风!bNbrports 是前面我们获得的 hub descriptor 的一个成员,表征这个 hub 有几个端口.很显然,月可无光,星可无语,usb 设备却不可以没有描述符.这里就是遍历每一个端口.busy_bits 是 struct usb_hub 的一个成员,unsigned long busy_bits[1],接下来的 event_bits 也是 ,change_bits 也是 ,unsigned long event_bits[1],unsigned long change_bits[1],test_bit()我们太熟悉了,在 usb-storage 里面到处都是,只不过当时我们测试的是 us->flags 里面的某个 flag 是否设置了,而这里我们要测试的有三个东东,首先测试 busy_bits,这个 flag 实际上只有在 reset 和 resume 的函数内部才会设置,所以这里我们先不用管,而这里的意思是,如果眼下这个端口正在执行 reset 或者 resume 操作,那么咱们就跳过去,不予理睬.

2689 行,测试 change_bits.结合 2690,2691,2692 行一起看.如果这个端口对应的 change_bits 没有设置,event_bits 没有设置过,hub->activating 也为 0,那么这里就执行 continue,不过我们想都不用想,因为我们就是从 hub_activate 进来的.我们来的时候 activating 就是设置成了 1 的,所以这里的 continue 我们是不用执行的.换言之,我们继续往下走.

2694 行,hub_port_status(),portstatus 和 portchange 是我们在 hub_events()伊始定义的两个变量,u16 portstatus,u16 portchange,即两个都是 16 位.尽管说了 N 遍了,但是我还是得说第 N+1 遍,这个函数仍然是来自 drivers/usb/core/hub.c:

```
1413 static int hub_port_status(struct usb_hub *hub, int port1,
1414                             u16 *status, u16 *change)
1415 {
1416         int ret;
1417
1418         mutex_lock(&hub->status_mutex);
1419         ret = get_port_status(hub->hdev, port1, &hub->status->port);
1420         if (ret < 4) {
1421                 dev_err (hub->intfdev,
1422                         "%s failed (err = %d)\n", __FUNCTION__, ret);
1423                 if (ret >= 0)
1424                         ret = -EIO;
1425         } else {
1426                 *status = le16_to_cpu(hub->status->port.wPortStatus);
1427                 *change = le16_to_cpu(hub->status->port.wPortChange);
1428                 ret = 0;
1429         }
1430         mutex_unlock(&hub->status_mutex);
1431         return ret;
1432 }
```

重要的是其中的那个 get_port_status()函数.

```
300 /*
301  * USB 2.0 spec Section 11.24.2.7
302  */
303 static int get_port_status(struct usb_device *hdev, int port1,
304                 struct usb_port_status *data)
305 {
306         int i, status = -ETIMEDOUT;
307
308         for (i = 0; i < USB_STS_RETRIES && status == -ETIMEDOUT; i++) {
309                 status = usb_control_msg(hdev, usb_rcvctrlpipe(hdev, 0),
310                                 USB_REQ_GET_STATUS, USB_DIR_IN |
USB_RT_PORT, 0, port1,
311                         data, sizeof(*data), USB_STS_TIMEOUT);
312         }
313         return status;
314 }
```

一路从泥泞走到美景,我们再也不会对 usb_control_msg()函数陌生了,这个函数做什么勾当我们完全是一目了然.Get Port Status 是 Hub 的一个标准请求,对我们来说就是一次控制传输就可以搞定.这个请求的格式如下图所示:

bmRequestType	bRequest	wValue	wIndex	wLength	Data
10100011B	GET_STATUS	Zero	Port	Four	Port Status and Change Status

其中这个 GET_STATUS 的对应具体的数值可以在下面这张表中对比得到,

Table 11-16. Hub Class Request Codes

bRequest	Value
GET_ STATUS	0
CLEAR_ FEATURE	1
RESERVED (used in previous specifications for GET_STATE)	2
SET_ FEATURE	3
Reserved for future use	*4-5*
GET_DESCRIPTOR	6
SET_DESCRIPTOR	7
CLEAR_TT_BUFFER	8
RESET_TT	9
GET_TT_STATE	10
STOP_TT	11

而关于各种请求,咱们在 include/linux/usb/ch9.h 中也定义了相应的宏,

```
72 /*
73  * Standard requests, for the bRequest field of a SETUP packet.
74  *
75  * These are qualified by the bRequestType field, so that for example
```

```
76   * TYPE_CLASS or TYPE_VENDOR specific feature flags could be retrieved
77   * by a GET_STATUS request.
78   */
79 #define USB_REQ_GET_STATUS              0x00
80 #define USB_REQ_CLEAR_FEATURE           0x01
81 #define USB_REQ_SET_FEATURE             0x03
82 #define USB_REQ_SET_ADDRESS             0x05
83 #define USB_REQ_GET_DESCRIPTOR          0x06
84 #define USB_REQ_SET_DESCRIPTOR          0x07
85 #define USB_REQ_GET_CONFIGURATION       0x08
86 #define USB_REQ_SET_CONFIGURATION       0x09
87 #define USB_REQ_GET_INTERFACE           0x0A
88 #define USB_REQ_SET_INTERFACE           0x0B
89 #define USB_REQ_SYNCH_FRAME             0x0C
90
91 #define USB_REQ_SET_ENCRYPTION          0x0D    /* Wireless USB */
92 #define USB_REQ_GET_ENCRYPTION          0x0E
93 #define USB_REQ_RPIPE_ABORT             0x0E
94 #define USB_REQ_SET_HANDSHAKE           0x0F
95 #define USB_REQ_RPIPE_RESET             0x0F
96 #define USB_REQ_GET_HANDSHAKE           0x10
97 #define USB_REQ_SET_CONNECTION          0x11
98 #define USB_REQ_SET_SECURITY_DATA       0x12
99 #define USB_REQ_GET_SECURITY_DATA       0x13
100 #define USB_REQ_SET_WUSB_DATA           0x14
101 #define USB_REQ_LOOPBACK_DATA_WRITE     0x15
102 #define USB_REQ_LOOPBACK_DATA_READ      0x16
103 #define USB_REQ_SET_INTERFACE_DS        0x17
```

比如这里咱们传递给 usb_control_msg 的 request 就是 USB_REQ_GET_STATUS,它的值为0,和usb spec中定义的GET_STATUS的值是对应的.这个请求返回两个咚咚,一个称为Port Status,一个称为 Port Change Status.usb_control_msg()的调用注定了返回值将保存在 struct usb_port_status *data 里面,这个结构体定义与 drivers/usb/core/hub.h 中:

```
58 /*
59   * Hub Status and Hub Change results
60   * See USB 2.0 spec Table 11-19 and Table 11-20
61   */
62 struct usb_port_status {
63         __le16 wPortStatus;
64         __le16 wPortChange;
65 } __attribute__ ((packed));
```

很显然这个格式是和实际的 spec 规范对应的,我们给 get_port_status()传递的实参是 &hub->status->port,port 也是一个 struct usb_port_status 结构体变量,所以在

hub_port_status()里面,1426 和 1427 两行我们就得到了 Status Bits 和 Status Change Bits.get_port_status()返回值就是 GET PORT STATUS 请求的返回数据的长度,它至少应该能够保存 wPortStatus 和 wPortChange,所以至少不能小于 4,所以 1420 行有这么一个错误判断.这样,hub_port_status()就返回了,而 status 和 change 这两个指针也算是满载而归了,正常的话返回值就是 0.

继续往下走,2699 行,children[i-1],这么一个冬冬我们从没有见过,但是我想白痴都知道,正是像 parent 和 children 这样的指针才能把 USB 树给建立起来,而我们才刚上路,肯定还没有设置 children,所以对我们来说,至少目前 children 数组肯定为空,而我们又知道 hub->activating 这时候肯定为 1,所以就看第三个条件了,portstatus&USB_PORT_STAT_CONNECTION,这是啥意思?稍有悟性的人就能看出来,这表明这个端口连了设备,没错,USB_PORT_STAT_CONNECTION 这个宏定义于 drivers/usb/core/hub.h 中:

```
67 /*
68  * wPortStatus bit field
69  * See USB 2.0 spec Table 11-21
70  */
71 #define USB_PORT_STAT_CONNECTION        0x0001
72 #define USB_PORT_STAT_ENABLE            0x0002
73 #define USB_PORT_STAT_SUSPEND           0x0004
74 #define USB_PORT_STAT_OVERCURRENT       0x0008
75 #define USB_PORT_STAT_RESET             0x0010
76 /* bits 5 to 7 are reserved */
77 #define USB_PORT_STAT_POWER             0x0100
78 #define USB_PORT_STAT_LOW_SPEED         0x0200
79 #define USB_PORT_STAT_HIGH_SPEED        0x0400
80 #define USB_PORT_STAT_TEST              0x0800
81 #define USB_PORT_STAT_INDICATOR         0x1000
82 /* bits 13 to 15 are reserved */
83
84 /*
85  * wPortChange bit field
86  * See USB 2.0 spec Table 11-22
87  * Bits 0 to 4 shown, bits 5 to 15 are reserved
88  */
89 #define USB_PORT_STAT_C_CONNECTION      0x0001
90 #define USB_PORT_STAT_C_ENABLE          0x0002
91 #define USB_PORT_STAT_C_SUSPEND         0x0004
92 #define USB_PORT_STAT_C_OVERCURRENT     0x0008
93 #define USB_PORT_STAT_C_RESET           0x0010
```

这都是这两个变量对应的宏,usb 2.0 的 spec 里面对这些宏的意义说得很清楚,USB_PORT_STAT_CONNECTION 的意思的确是表征是否有设备连接在这个端口上,我们不妨假设有,那么 portstatus 和它相与的结果就是 1,在 usb spec 里面,这一位叫做 Current Connect Status 位,于是这里我们会看到 connect_change 被设置成了 1.而接下

来,USB_PORT_STAT_C_CONNECTION 则是表征这个端口的 Current Connect Status 位是否有变化,如果有变化,那么 portchange 和 USB_PORT_STAT_C_CONNECTION 相与的结果就是 1,对于这种情况,我们需要发送另一个请求以清除这个 flag,并且将 connect_change 也设置为 1.这个请求叫做 Clear Port Feature.这个请求也是 Hub 的标准请求,

bmRequestType	bRequest	wValue	wIndex		wLength	Data
00100011B	CLEAR_ FEATURE	Feature Selector	Selector	Port	Zero	None

它的作用就是 reset hub 端口的某种 feature.clear_port_feature() 定义于 drivers/usb/core/hub.c:

```
162 /*
163  * USB 2.0 spec Section 11.24.2.2
164  */
165 static int clear_port_feature(struct usb_device *hdev, int port1, int feature)
166 {
167         return usb_control_msg(hdev, usb_sndctrlpipe(hdev, 0),
168                 USB_REQ_CLEAR_FEATURE, USB_RT_PORT, feature, port1,
169                 NULL, 0, 1000);
170 }
```

对比上面贴出来的定义可知,USB_REQ_CLEAR_FEATURE 和 usb spec 中的 CLEAR_FEATURE 这个请求是对应的,那么一共有些什么 Feature 呢?在 drivers/usb/core/hub.h 中是这样定义的,

```
38 /*
39  * Port feature numbers
40  * See USB 2.0 spec Table 11-17
41  */
42 #define USB_PORT_FEAT_CONNECTION        0
43 #define USB_PORT_FEAT_ENABLE            1
44 #define USB_PORT_FEAT_SUSPEND           2
45 #define USB_PORT_FEAT_OVER_CURRENT      3
46 #define USB_PORT_FEAT_RESET             4
47 #define USB_PORT_FEAT_POWER             8
48 #define USB_PORT_FEAT_LOWSPEED          9
49 #define USB_PORT_FEAT_HIGHSPEED         10
50 #define USB_PORT_FEAT_C_CONNECTION      16
51 #define USB_PORT_FEAT_C_ENABLE          17
52 #define USB_PORT_FEAT_C_SUSPEND         18
53 #define USB_PORT_FEAT_C_OVER_CURRENT    19
54 #define USB_PORT_FEAT_C_RESET           20
```

```
55 #define USB_PORT_FEAT_TEST              21
56 #define USB_PORT_FEAT_INDICATOR         22
```

而在 usb spec 中,有一张与之相对应的表格,定义了许多的 feature,如图所示:

Table 11-17. Hub Class Feature Selectors

	Recipient	Value
C_HUB_LOCAL_POWER	Hub	0
C_HUB_OVER_CURRENT	Hub	1
PORT_CONNECTION	Port	0
PORT_ENABLE	Port	1
PORT_SUSPEND	Port	2
PORT_OVER_CURRENT	Port	3
PORT_RESET	Port	4

Table 11-17. Hub Class Feature Selectors (continued)

	Recipient	Value
PORT_POWER	Port	8
PORT_LOW_SPEED	Port	9
C_PORT_CONNECTION	Port	16
C_PORT_ENABLE	Port	17
C_PORT_SUSPEND	Port	18
C_PORT_OVER_CURRENT	Port	19
C_PORT_RESET	Port	20
PORT_TEST	Port	21
PORT_INDICATOR	Port	22

所以,我们清除的是 C_PORT_CONNECTION 这一个 feature.spec 里面说了,清除一个状态改变的 feature 就等于承认这么一个 feature.(clearing that status change acknowledges the change)理由很简单,每次你检测到一个 flag 被设置之后,你都应该清除掉它,以便下次人家一设你就知道是人家设了,否则你不清你下次判断你就不知道是不是又有人设了.同理,接下来的每个与 portchange 相关的判断语句都要这么做.所以如果 portchange 与上

USB_PORT_STAT_C_CONNECTION 确实为 1,那么我们就要清除这个 feature.同时我们当然也要记录 connect_change 为 1.

继续,每个端口都有一个开关,这叫做 enable 或者 disable 一个端口.portchange 和 USB_PORT_STAT_C_ENABLE 相与如果为 1 的话,说明端口开关有变化.和刚才一样,首先我们要做的是,清除掉这个变化的 feature.但是这里需要注意,spec 里对这个 feature 是这样规定的,如果 portchange 和 USB_PORT_STAT_C_ENABLE 为 1,说明这个 port 是从 enable 状态进入了 disable 状态.为什么呢?因为在 spec 规定了,Hub 的端口是不可以直接设置成 enable 的.通常让 Hub 端口 enable 的方法是 reset hub port.用 spec 的话说,这叫做发送另一个 request,名为 SET_FEATURE.SET_FEATURE 和 CLEAR_FEATURE 是对应的,一个设置一个清除. 对于 PORT_ENABLE 这一位,用 spec 里的话说,This bit may be set only as a result of a SetPortFeature(PORT_RESET) request,PORT_RESET 是为 hub 定义的众多 feature 中的一种.最后提醒一点,2711 至 2715 这段 if 语句仅仅是为了打印调试信息的,就是说如果 port enable 改变了,但是端口连接没有改变,那么打印出信息来通知调试者,不要把 clear_port_feature 这一行也纳入到 if 语句里去了.因为 port enable 的改变有多种可能,其中一种可能就是由于检测到了 disconnection,但是对于这种情况,我们下面要处理的,所以甭急.

下面这段代码就比较帅了,电磁干扰都给扯进来了.EMI,就是电磁干扰.就是说有的时候 hub port 的 enable 变成 disable 有可能是由于电磁干扰造成的,这个 if 条件判断的是,端口被 disable 了,但是连接没有变化,并且 hdev->children[i]还有值,这就说明明明有子设备连在端口上,可是端口却被 disable 了,基本上这种情况就是电磁干扰造成的,否则 hub 端口不会有这种抽风的举动.那么这种情况就设置 connect_change 为 1.因为接下来我们会看到对于 connect_change 为 1 的情况,我们会专门进行处理,而更犀利更一针见血的说法就是,hub_events()其实最重要的任务就是对于端口连接有变化的情况进行处理,或者说进行响应.

再往下,portchange 和 USB_PORT_STAT_C_SUSPEND 相与如果为 1,表明连在该端口的设备的 suspend 状态有变化,并且是从 suspended 状态出来,也就是说 resume 完成.(别问我为什么,spec 就这么规定的,没什么理由,一定要问理由那你问郑源去,他不是唱那什么如果你真的需要什么理由,一万个够不够吗.)那么首先我们就调用 clear_port_feature 清掉这个 feature.接下来这个 if 牵扯到的东西比较高深,涉及到电源管理中很华丽的部分,我们只能先跳过.否则深陷其中难免会走火入魔欲罢不能.总之这里做的就是对于该端口连了子设备的情况就把子设备唤醒,否则如果端口没有连子设备,那么就把端口 disable 掉.

2754 行,portchange 如果和 USB_PORT_STAT_C_OVERCURRENT 相与结果为 1 的话,说明这个端口可能曾经存在电流过大的情况,而现在这种情况不存在了,或者本来不存在而现在存在了.对此我们能做的就是首先清除这个 feature.有一种比较特别的情况是,如果其它的端口电流过大,那么将会导致本端口断电,即 hub 上一个端口出现 over-current 条件将有可能引起 hub 上其它端口陷入 powered off 的状态.不管怎么说,对于 over-current 的情况我们都把 hub 重新上电,执行 hub_power_on().

2763 行,portchange 如果和 USB_PORT_STAT_C_RESET 相与为 1 的话,这叫做一个端口从 Resetting 状态进入到 Enabled 状态.

2771 行,connect_change 如果为 1,就执行 hub_port_connect_change(),啥也不说了,这是每一个看 hub 驱动的人最期待的函数,因为这正是我们的原始动机,即当一个 usb 设备插入 usb 接口之后究竟会发生什么,usb 设备驱动程序提供那个 probe 函数究竟是如何被调用的.这些疑问统统会在这个函数里得到答案.来自 drivers/usb/core/hub.c:

```
2404 /* Handle physical or logical connection change events.
2405  * This routine is called when:
2406  *      a port connection-change occurs;
2407  *      a port enable-change occurs (often caused by EMI);
2408  *      usb_reset_device() encounters changed descriptors (as from
2409  *              a firmware download)
2410  * caller already locked the hub
2411  */
2412 static void hub_port_connect_change(struct usb_hub *hub, int port1,
2413                                     u16 portstatus, u16 portchange)
2414 {
2415         struct usb_device *hdev = hub->hdev;
2416         struct device *hub_dev = hub->intfdev;
2417                                      u16     wHubCharacteristics     =
le16_to_cpu(hub->descriptor->wHubCharacteristics);
2418         int status, i;
2419
2420         dev_dbg (hub_dev,
2421                 "port %d, status %04x, change %04x, %s\n",
2422                 port1, portstatus, portchange, portspeed (portstatus));
2423
2424         if (hub->has_indicators) {
2425                 set_port_led(hub, port1, HUB_LED_AUTO);
2426                 hub->indicator[port1-1] = INDICATOR_AUTO;
2427         }
2428
2429         /* Disconnect any existing devices under this port */
2430         if (hdev->children[port1-1])
2431                 usb_disconnect(&hdev->children[port1-1]);
2432         clear_bit(port1, hub->change_bits);
2433
2434 #ifdef  CONFIG_USB_OTG
2435         /* during HNP, don't repeat the debounce */
2436         if (hdev->bus->is_b_host)
2437                 portchange &= ~USB_PORT_STAT_C_CONNECTION;
2438 #endif
2439
2440         if (portchange & USB_PORT_STAT_C_CONNECTION) {
2441                 status = hub_port_debounce(hub, port1);
```

```
2442                if (status < 0) {
2443                        if (printk_ratelimit())
2444                                dev_err (hub_dev, "connect-debounce failed,
"
2445                                                "port %d disabled\n",
port1);
2446                        goto done;
2447                }
2448                portstatus = status;
2449        }
2450
2451        /* Return now if nothing is connected */
2452        if (!(portstatus & USB_PORT_STAT_CONNECTION)) {
2453
2454                /* maybe switch power back on (e.g. root hub was reset) */
2455                if ((wHubCharacteristics & HUB_CHAR_LPSM) < 2
2456                                                && !(portstatus & (1 <<
USB_PORT_FEAT_POWER)))
2457                                        set_port_feature(hdev, port1,
USB_PORT_FEAT_POWER);
2458
2459                if (portstatus & USB_PORT_STAT_ENABLE)
2460                        goto done;
2461                return;
2462        }
2463
2464 #ifdef  CONFIG_USB_SUSPEND
2465        /* If something is connected, but the port is suspended, wake it up. */
2466        if (portstatus & USB_PORT_STAT_SUSPEND) {
2467                status = hub_port_resume(hub, port1, NULL);
2468                if (status < 0) {
2469                        dev_dbg(hub_dev,
2470                                "can't clear suspend on port %d; %d\n",
2471                                port1, status);
2472                        goto done;
2473                }
2474        }
2475 #endif
2476
2477        for (i = 0; i < SET_CONFIG_TRIES; i++) {
2478                struct usb_device *udev;
2479
2480                /* reallocate for each attempt, since references
2481                 * to the previous one can escape in various ways
```

```
2482                 */
2483                udev = usb_alloc_dev(hdev, hdev->bus, port1);
2484                if (!udev) {
2485                        dev_err (hub_dev,
2486                                "couldn't allocate port %d usb_device\n",
2487                                port1);
2488                        goto done;
2489                }
2490
2491                usb_set_device_state(udev, USB_STATE_POWERED);
2492                udev->speed = USB_SPEED_UNKNOWN;
2493                udev->bus_mA = hub->mA_per_port;
2494                udev->level = hdev->level + 1;
2495
2496                /* set the address */
2497                choose_address(udev);
2498                if (udev->devnum <= 0) {
2499                        status = -ENOTCONN;      /* Don't retry */
2500                        goto loop;
2501                }
2502
2503                /* reset and get descriptor */
2504                status = hub_port_init(hub, udev, port1, i);
2505                if (status < 0)
2506                        goto loop;
2507
2508                /* consecutive bus-powered hubs aren't reliable; they can
2509                 * violate the voltage drop budget.  if the new child has
2510                 * a "powered" LED, users should notice we didn't enable it
2511                 * (without reading syslog), even without per-port LEDs
2512                 * on the parent.
2513                 */
2514                if (udev->descriptor.bDeviceClass == USB_CLASS_HUB
2515                                && udev->bus_mA <= 100) {
2516                        u16      devstat;
2517
2518                        status = usb_get_status(udev, USB_RECIP_DEVICE,
0,
2519                                        &devstat);
2520                        if (status < 2) {
2521                                dev_dbg(&udev->dev, "get status %d ?\n",
status);
2522                                goto loop_disable;
2523                        }
```

```
2524                    le16_to_cpus(&devstat);
2525                    if ((devstat & (1 << USB_DEVICE_SELF_POWERED))
== 0) {
2526                            dev_err(&udev->dev,
2527                                    "can't connect bus-powered hub "
2528                                    "to this port\n");
2529                            if (hub->has_indicators) {
2530                                    hub->indicator[port1-1] =
2531                                            INDICATOR_AMBER_BLINK;
2532                                        schedule_delayed_work
(&hub->leds, 0);
2533                            }
2534                            status = -ENOTCONN;     /* Don't retry */
2535                            goto loop_disable;
2536                    }
2537            }
2538
2539            /* check for devices running slower than they could */
2540            if (le16_to_cpu(udev->descriptor.bcdUSB) >= 0x0200
2541                            && udev->speed == USB_SPEED_FULL
2542                            && highspeed_hubs != 0)
2543                    check_highspeed (hub, udev, port1);
2544
2545            /* Store the parent's children[] pointer.  At this point
2546             * udev becomes globally accessible, although presumably
2547             * no one will look at it until hdev is unlocked.
2548             */
2549            status = 0;
2550
2551            /* We mustn't add new devices if the parent hub has
2552             * been disconnected; we would race with the
2553             * recursively_mark_NOTATTACHED() routine.
2554             */
2555            spin_lock_irq(&device_state_lock);
2556            if (hdev->state == USB_STATE_NOTATTACHED)
2557                    status = -ENOTCONN;
2558            else
2559                    hdev->children[port1-1] = udev;
2560            spin_unlock_irq(&device_state_lock);
2561
2562            /* Run it through the hoops (find a driver, etc) */
2563            if (!status) {
2564                    status = usb_new_device(udev);
2565                    if (status) {
```

```
2566                                    spin_lock_irq(&device_state_lock);
2567                                    hdev->children[port1-1] = NULL;
2568                                    spin_unlock_irq(&device_state_lock);
2569                            }
2570                    }
2571
2572                    if (status)
2573                            goto loop_disable;
2574
2575                    status = hub_power_remaining(hub);
2576                    if (status)
2577                             dev_dbg(hub_dev, "%dmA power budget left\n",
status);
2578
2579                    return;
2580
2581 loop_disable:
2582                    hub_port_disable(hub, port1, 1);
2583 loop:
2584                    ep0_reinit(udev);
2585                    release_address(udev);
2586                    usb_put_dev(udev);
2587                    if (status == -ENOTCONN)
2588                            break;
2589            }
2590
2591 done:
2592            hub_port_disable(hub, port1, 1);
2593 }
```

到今天我算是看明白了,内核里面这些函数,没有最变态只有更变态,变态哪都有,可是开源社区尤其多!你们他妈的不是我的冤家派来故意玩我的吧?面对这个函数,我真的想吐血!我打算不像过去那样一行一行讲了,我必须先来个提纲挈领,必须先开门见山把这个函数的哲学思想讲清楚,否则一行一行往下讲肯定晕菜.

盖茨家对 Linux 代码的影响

hub_port_connect_change,顾名思义,当 hub 端口上有连接变化时调用这个函数,这种变化既可以是物理变化也可以是逻辑变化.注释里说得也很清楚.有三种情况会调用这个函数,一个是连接有变化,一个是端口本身重新使能,即所谓的 enable,这种情况通常就是为了对付电磁干扰的,正如我们前面的判断中所说的那样,第三种情况就是在复位一个设备的时候发现其描述符变了,这通常对应的是硬件本身有了升级.很显然,第一种情况是真正的物理变化,后两者就算是逻辑变化.

前面几行的赋值不用多说,2422 行这个 portspeed()函数就是获得这个端口所接的设备的 speed,来自 drivers/usb/core/hub.c:

```
121 static inline char *portspeed(int portstatus)
122 {
123         if (portstatus & (1 << USB_PORT_FEAT_HIGHSPEED))
124                 return "480 Mb/s";
125         else if (portstatus & (1 << USB_PORT_FEAT_LOWSPEED))
126                 return "1.5 Mb/s";
127         else
128                 return "12 Mb/s";
129 }
```

意图太太太明显了,确定是高速/低速/还是全速的设备.这些信息都包含在 portstatus 的 bit9 和 bit10 里面.

2424 行这一小段,不多说,就是那个指示灯的事,有指示灯就设置一下,指示灯可以设置成琥珀色可以设置成绿色可以关闭,也可以设置成自动,这里设置为自动.所谓自动设置就是 hub 自己根据端口的状态来设置,比如 hub 挂起了那么指示灯当然就应该熄灭.

2430 行,如果咱们这个 hub 的子设备还有值,那么什么也别说,先把它给断了,为嘛?不为嘛.(注:这两句对白得用天津话说)

我们说了,端口连接有变化,可是变化可以是两个方向啊.一个是原来没有设备现在有了,一个是恰恰相反,原来有设备而现在没有.对于前者,hdev->children[port1-1]肯定为空,而对于后者这个指针应该就不为空,对于前者,我们接下来要做的事情就是对新连进来的设备进行初始化进行配置分配地址然后为该设备寻找相应的设备驱动程序,如果找到合适的了就调用该设备驱动程序提供的指针函数来再进行更细致更深入更对口的初始化.但是对于后者,就没必要那么麻烦了,直接调用 usb_disconnect()函数执行一些清扫工作,并且把 hub 的 change_bits 清掉.然后再确定一下端口上确实没有连接什么设备,那就可以返回了.对于 usb_disconnect()我们暂时先不看.我们先关心的是如果有设备连接进来该怎么处理.

2434 至 2438 这一段我们不管,因为我们早已经作了一个无耻的假设,假设我们关闭掉了 CONFIG_USB_OTG 这个编译开关.

2440 行,别忘了,对于连接从有到无的变化,刚才我们可是清掉了 hub 的 change_bits,所以这里再次判断 portchange&USB_PORT_STAT_C_CONNECTION 的意思就是对于连接从无到有的情况,我们还必须做一件事情,那就是判断反弹.啥叫反弹啊?换一种说法,叫做去抖动.比如你在网上聊 QQ,关键时刻,你深情表白,按键盘,你按一下键,正常情况下,在按下和松开的过程中触片可能快速的接触和分离好多次,而此时此刻你自己还无比的激动无比的紧张,那么按键肯定是多次抖动,而驱动程序不可能响应你很多次吧,因为你毕竟只是按一下键.同理,这样的去抖动技术在 hub 中也是需要的.所以原则上,spec 规定,只有持续了 100ms 的插入才算真正的插入,或者说才算稳定的插入.hub_port_debounce 就是干这件事情的,这个函数会让你至少等待 100ms,如果设备依然在,那么说明稳定了,这种情况下函数返回值就是端口的状态,即 2448 行那样把 status 赋给 portstatus.而如果 100ms 不到设备又被拔走了,那么返回负的错误码,然后打印信

息告诉你去抖动挂了.printk_ratelimit 是一个新鲜的函数,其实就是 printk 的变种,printk_ratelimit 的用途就是当你某条消息可能会重复的被多次打印的,甚至极限情况下,打印个成千上万条,直接导致日志文件溢出,把别的信息都冲掉了,所以这样是不好的,于是进化出来一个 printk_ratelimit(),它会控制打印消息的频率,如果短期内连续出现打印消息,那么它把消息抛弃,这种情况下这个函数返回 0,所以,只有返回非 0 值的情况下才会真正打印.

2452 行,如果经历过以上折腾之后,现在端口状态已经是没有设备连接了,那么再作两个判断,一个是说如果该端口的电源被关闭了,那么就打开电源.另一个是说如果该端口还处于 ENABLE 的状态,那么 goto done.可以看到 done 那里的代码就是把这个端口给 disable 掉.否则,如果端口已经是 disable 状态了,那么就直接返回吧,这次事件就算处理完了.

2466 行,这一段就是说连接虽然变化了,并且设备也是从无到有了,但是如果之前这个端口是出于 suspend 状态的话,那么这里就要调用 hub_port_resume()把这个端口给恢复.关于电源管理的深层次代码,我们暂时先搁一边.但是作为 hub 驱动程序,电源管理部分是必不可少的,现在国家提倡建设节约型社会,我们国家做一个发展中国家,资源又比较短缺,所以每一个爱国人士都有责任有义务为节约能源而做出自己力所能及的贡献.

2477 行,SET_CONFIG_TRIES 这个宏,看了就头疼,这又要引出一段故事了.

首先我想说的是,以下这个 for 循环的目的很明确,为一个 usb 设备申请内存空间,设置它的状态,把它复位,为它设置地址,获取它的描述符,然后就向设备模型中添加这么一个设备,再然后会为这个设备寻找它的驱动程序,再然后驱动程序提供的 probe()函数就会被调用.但是从 usb 这边来说,只要调用 device_add 这么一个函数向设备模型核心层添加设备就够了,剩下的事情设备模型层会去处理,这就是设备模型的优点,所以也叫统一的设备模型.就是说不管你是 pci 还是 usb 还是 scsi,总线驱动的工作就是申请并建立总线的数据结构,而设备驱动的工作就是往这条总线上注册,调用 driver_add,而设备这边也是一样,也往该总线上注册,即调用 device_add.而 driver_add 就会在总线上寻找每一个设备,如果找到了自己支持的设备,并且该设备没有和别的驱动相绑定,那么就绑定它.反过来,设备这边的做法也一样,device_add 在总线上寻找每一个设备驱动,找到了合适的就绑定它.最后,调用 probe 函数,然后兵权就交给了设备驱动.整个这个过程就叫做 usb 设备初始化.所以说,作为设备驱动程序本身来讲,它只是参与了设备初始化的一部分,很小的一部分,真正的重要的工作,都是中央集权的,由核心来执行,对于 usb 设备来说,就是 hub 驱动来集中处理这些事情.之所以这些工作可以统一来做,是因为凡是 usb 设备,它都必须遵循一些共同的特征,就好比,我们写托福作文,有那么一些书,就提供了模版,甭管是什么作文题目,都可以套用这些模版.因为不管是写人写猴还是写猪,它们都有五官,所以如果写一篇说明文的话,都一定会写五官.以前我觉得自己还挺帅,可是后来我发现,我有的五官,猪也有.我一个在科大读书的同学更惨,科大女生少,结果时间长了,他说他看到一头母猪也觉得挺眉清目秀的.

Ok,那么这么一个过程为何要循环呢?早在中日甲午战争的时候,Linux 内核中是没有 SET_CONFIG_TRIES 这么一个宏的,后来在非典那年的圣诞节前夕,David Brownell 同志提交了一个内核补丁,出现了这么一个宏,并且把这个宏的值被固定为 2.再之后,2004 年底,这个宏的定义发生了一些变化,变得更加灵活.

第一,为什么设为 2?这个理由超级简单,我们说了,要获取设备描述符,如何获得?给设备发送请求,发送一个 GET-DEVICE-DESCRIPTOR 的请求,然后正规企业的正规设备就会返回设备描述符.看仔细了,正规,什么叫正规? Sony 算不算正规?Philip 算不算正规?我不知道.现在不负责任的厂

商越来越多,一边号称自己的正规企业,一边生产出让人大跌隐形眼镜的设备来卖.你就说我前几天去清华校内那个桃李餐厅,三层的那个小餐厅,他们也敢说自己正规,结帐的时候发票都不给开,这边收着我们的钱,那边逃着国家的税,你说这能叫正规吗?说回 usb,本来 usb spec 就规定了只要发送这么一个请求你设备就该返回你的设备描述符,可是你这么一试,它偏给你说请求失败.你说急死你不?所以,写代码的人学乖了,发这些重要的请求都按两次发,一次不成功再发一次,成功了就不发,两次还不成功,那没办法了,病入膏肓了...

第二,那么后来为何又改为别的值了?我们来看,现在这个宏被改为了什么?drivers/usb/core/hub.c 中:

```
1446 #define PORT_RESET_TRIES        5
1447 #define SET_ADDRESS_TRIES       2
1448 #define GET_DESCRIPTOR_TRIES    2
1449 #define SET_CONFIG_TRIES        (2 * (use_both_schemes + 1))
1450 #define USE_NEW_SCHEME(i)       ((i) / 2 == old_scheme_first)
```

usb_both_schemes 和 old_scheme_first.这两个参数,

```
95 /*
96  * As of 2.6.10 we introduce a new USB device initialization scheme which
97  * closely resembles the way Windows works.  Hopefully it will be compatible
98  * with a wider range of devices than the old scheme.  However some
previously
99  * working devices may start giving rise to "device not accepting address"
100  * errors; if that happens the user can try the old scheme by adjusting the
101  * following module parameters.
102  *
103  * For maximum flexibility there are two boolean parameters to control the
104  * hub driver's behavior.  On the first initialization attempt, if the
105  * "old_scheme_first" parameter is set then the old scheme will be used,
106  * otherwise the new scheme is used.  If that fails and "use_both_schemes"
107  * is set, then the driver will make another attempt, using the other scheme.
108  */
109 static int old_scheme_first = 0;
110 module_param(old_scheme_first, bool, S_IRUGO | S_IWUSR);
111 MODULE_PARM_DESC(old_scheme_first,
112                  "start with the old device initialization scheme");
113
114 static int use_both_schemes = 1;
115 module_param(use_both_schemes, bool, S_IRUGO | S_IWUSR);
116 MODULE_PARM_DESC(use_both_schemes,
117                 "try the other device initialization scheme if the "
    118                      "first one fails");
```

如果你像李玟不知道满江红的词作者岳飞是谁,这没什么,如果你像周杰伦不知道雷锋是谁,这也没什么,如果你像范冰冰不知道生得渺小死得和谐的刘胡兰是哪个时代的人,这也没什么,如果你像杨丞琳不知道抗日战争是八年,这仍然没什么,可是如果你稍有悟性,你就应该能看出,这两个是模块加载的时候的参数,在你加载一个模块的时候,在你用 modprobe 或者 insmod 加载一个模块的时候,你可以显示的指定 old_scheme_first 的值,可以指定 usb_both_schemes 的值,如果你不指定,那么这里的默认值是 old_scheme_first 为 0,而 use_both_schemes 为 1.

那么他们具体起着什么作用?scheme,盗版的金山词霸告诉我,方案,计划的意思.那么 old scheme,both scheme 这两个词组似乎已经反映出来说有两个方案,一个旧的,一个新的.这是怎么回事?什么方面的方案?一个是来自Linux的方案,一个是来自Windows的方案,目的是为了获得设备的描述符.

说来话长,首先我想说,你们这些Linux发烧友们,你们老是说微软不好,说Windows不好,可是说句良心话,没有Windows,没有微软,你们靠什么学习Linux?你们是如何知道Linux的?你不要跟我说你是先学习 Linux 然后才知道微软的,不要说你是先用 Mozilla 浏览网页后来才从网页上知道有 IE 这么一个冬冬存在的.呵呵,我相信大多数人是恰恰相反吧.

所以我一直是喜欢盖茨家的产品的,尤其是在 2001 年 10 月盖茨来中国在上海我亲眼见过他听他做了一次报告之后,更是觉得他很了不起.具体到我们目前提到的这个方案问题,这里需要一些背景知识.

第一个,endpoint 0.0 号端点是 usb spec 中一个特殊的端点.此前我一直没有一本正经的讲过端点 0,也许你早就在问了,为何当初计算端点数目的时候没有提到端点 0 呢?其实我还不是为你好,多一事不如少一事,有些概念,和爱情一样,只有该认识的时候才去认识比较好.正如电影<<2046>>里所说的那样,爱情这东西,时间很关键,认识得太早或太晚,都不行.

usb spec 中是这样规定的,所有的 USB 设备都有一个默认的控制管道.英文叫 Default Control Pipe.与这条管道对应的端点叫做 0 号端点,也就是传说中的 endpoint zero.这个端点的信息不需要纪录在配置描述符里,就是说并没有一个专门的端点描述符来描述这个 0 号端点,因为不需要,原因是 endpoint zero 基本上所有的特性都是在 spec 规定好了的,大家都一样,所以不需要每个设备另外准备一个描述符来描述它.(换言之,在接口描述符里的 bNumEndpoints 是指的该 interface 包含的端点,但是这其中并不包含 Endpoint zero,如果一个设备除了这个 Endpoint zero 以外没有别的端点了,那么它的接口描述符里的 bNumEndpoints 就应该是 0,而不是 1.)然而,别忘了我说的是"基本上",有一个特性则是不一样的,这叫做 maximum packet size,每个端点都有这么一个特性,即告诉你该端点能够发送或者接收的包的最大值.对于通常的端点来说,这个值被保存在该端点描述符中的 wMaxPacketSize 这一个 field,而对于端点 0 就不一样了,由于它自己没有一个描述符,而每个设备又都有这么一个端点,所以这个信息被保存在了设备描述符里,所以我们在设备描述符里可以看到这么一项,bMaxPacketSize0,而且 spec 还规定了,这个值只能是 8,16,32 或者 64 这四者之一,而且,如果一个设备工作在高速模式,这个值还只能是 64,取别的值都不行.

而我们知道,我们所做的很多工作都是通过与 endpoint 0 打交道来完成的,那么 endpoint 0 的 max packet size 自然是我们必须要知道的,否则肯定没法正确的进行控制传输.于是问题就出现了.我不知道 max packet size 就没法进行正常的传输,可是 max packet size 又在设备描述符里,我不进行传输我就不知道 max packet size 啊?我晕!这一刻,我想到了美国作家约瑟夫 海

勒的代表作<<第22条军规>>.说第二次世界大战末期,飞行大队的一个上校不断给飞行员们增加飞行任务,远远超出一般规定,飞行员们都得了恐惧症,很多人惶惶不可终日,其中投弹手尤塞恩找到一个军医帮忙,想让他证明自己疯了.军医告诉他,虽然按照所谓的"第 22 条军规",疯子可以免于飞行,但同时又规定必须由本人提出申请,而如果本人一旦提出申请,便证明你并未变疯,因为"对自身安全表示关注,乃是头脑理性活动的结果".这样,这条表面讲究人道的军规就成了耍弄人的圈套.难道这里 usb spec 也是一个温柔的陷阱?

我喘着气对自己说:冷静,冷静,妈的,冷静,冷静,oh,shit,冷静冷静冷静.

后来我终于明白了.

可是星爷说过:"以你的智商我很难跟你解释."

我刚才说了,max packet size 只能是 8,16,32 或者 64,这也就是说,至少你得是 8 吧,而我们惊人的发现,设备描述符公共是 18 个 bytes,其中,第 byte7 恰恰就是 bMaxPacketSize0,byte0 到 byte7 算起来就是 8 个字节,那也就是说,我先读 8 个字节,然后设备返回 device descriptor 的前 8 个字节,于是我就知道它真正的 max packet size 了,于是我再读一次,这次才把整个描述符 18 个字节都给读出来,不就 ok 了吗?我靠,写代码的你们他妈的太有才了.

但事情往往不是这么简单,正所谓人算不如天算,生活不是电影,生活比电影苦.写代码的以为自己的算法天衣无缝.可是实践下来却遇到了问题.马克思主义哲学认为,实践是检验人品的唯一标准,这是由人品的本性和实践的特点所决定的.马克思主义哲学把实践的观点引入认识论,把辩证法和唯物主义有机地结合起来,在人类认识史上真正科学地解决了人品标准问题.

在 2004 年 10 月,即我大四开始艰难找工作的那段日子里,开源社区的同志们发现一个怪事,Sony 家的一个摄像机没法在 Linux 下正常工作.问题就出在获取设备描述符上,当你把一个 8 个字节的 GET-DEVICE-DESCRIPTOR 的请求发送给它们家的设备时,你得不到一个 8 个字节的不完整的描述符,相反,你会遇到溢出的错误,因为 Sony 它们家的设备只想一口气把 18 个字节的整个设备描述符全都给你返回,结果导致了错误,而且实践证明这样的错误还有可能会毁坏设备.

咦,奇怪了,这怎么办?有人爆料说这款设备在 Windows 下工作是完全正常的.这可不得了了.如何是好?后来又有人爆料,说 Windows 下面人家采取的是另一种策略,或者说方案,人家就是直接发送 64 个字节的请求过去,即要求你设备返回 64 个字节过来,如果你设备端点 0 的 max packet size 是 32 或者 64,那么你反正只要把 18 个字节的设备描述传递过来就可以了,但是如果你设备端点 0 的 max packet size 就是 8 或者 16,而设备描述符是 18 个字节,一次肯定传递不完,那么你必然是传递了一次以后还等待着继续传递,但是我从驱动角度来说,我只要获得了 8 个字节就够了,而对于设备,你不是等着继续传吗,我直接对你做一次 reset,让你复位,这样不就清掉了你剩下的想传的数据了么?然后我获得了前 8 个字节我就可以知道你真正的 max packet size,然后我就按这个真正的最大值来进行下面的传输,首先就是获得你那个 18 个字节的真正的完整的设备描述符.这样子,也就达到了目的了.这就是 Windows 下面的处理方法.

于是开源社区的兄弟们发现,很多厂商都是按着 Windows 的这种策略来测试自己的设备的,他们压根儿就没有测试过请求 8 个字节的设备描述符,于是,也就没人能保证当你发送请求要它返回 8 个字节的设备描述符的时候它能够正确的响应.所以,Linux 开发者们委曲求全,把这种 Windows

下的策略给加了进来,其实,Linux 的那种策略才是 usb spec 提供的策略,而现实是,Windows 没有遵守这种策略,然而厂商们出厂的时候就只是测试了能在 Windows 下工作,他们认为遵守 Windows 就是遵守了 usb spec.而事实呢,却并非如此.严格意义来说,这是 Windows 这边的 bug,不过这种做法却引导了潮流.呵呵,这不禁让人想起另一桩趣事,毛阿敏刚出道的时候,第一次在中央电视台录像,战战兢兢的,那时候姜昆给他主持节目,姜昆那时候已经是腕儿,她还没成腕儿呢,她向人家姜昆请教自己问人家自己有什么毛病,姜昆说别的都行,就是走路显着不太成熟,毛阿敏感激不尽,而且沉痛的说,姜老师,我就是上台这路这不会走.结果后来,一曲你从哪里来使毛阿敏名声大振,又过了些日子,姜昆发现,所有的小歌星们都开始学毛阿敏那两步走,姜昆这个气哟,向毛阿敏控诉,毛阿敏自己也乐得不行.当了名人了,就是毛病也有人学,现实就是这么可乐.(参考文献,姜昆<<笑面人生>>,1996 年出版,我妈 1997 年来北京的时候买的)

所以,就这样,如今的代码里实现了这两种策略,每种试两次,原来的那种策略叫做 old scheme,现在的做法就是具体使用那种策略你作为用户你可以在加载模块的时候自己设置,但是如果你不设置,那么默认的方法就是先使用新的这种机制,试两次,然后如果不行,就使用旧的那种,我们看到前面我贴出来的那个宏, USE_NEW_SCHEME,她就是用来判断是不是使用新的 scheme 的.这个宏会在 hub_port_init()函数中用到,如果它为真,那么就用新的策略,即发送那个期望 64 个字节的请求.如果 fail 了,那才发送那个 8 个字节的请求.这些我们将会在 hub_port_init()中看到,到时候再说.

至此我们总算可以理解 SET_CONFIG_TRIES 这么一个宏了,它被定义为(2 * (use_both_schemes + 1)),而 use_both_schemes 就是说两种策略都用,这个参数也是可以自己在加载模块的时候设置,默认值为 1,即默认的情况时先用新的策略,不行就用旧的那个.而 usb_both_schemes 为 1 就意味着 SET_CONFIG_TRIES 等于 4,即老的策略试两次,新的策略试两次,当然,成功了就不用多试了,多试是为失败而准备的.

看明白了这个宏,我们可以进入到这个 for 循环来仔细看个究竟了.

八大重量级函数闪亮登场(一)

还有人记得 1996 年那部史诗般的电视剧<<英雄无悔>>吗?那年我初二.这部戏让我认识了濮存昕,也是这部戏确立了濮存昕少妇杀手的地位,后来大肆那年濮存昕去过复旦,宣传艾滋病方面的知识,尽管那时候我正在求职的路上遭遇种种挫折,但还是抽出了时间去五教看了听了他的讲座,完了之后他还即兴了一段朗诵.我觉得他身上那种健康的形象是我喜欢的,因为这种积极向上的东西我太缺了.

<<英雄无悔>>里面濮存昕扮演的公安局长高天有一句最经典的话:世上注定会有撑船人和坐船人,而他应该是那个执著的撑船仔.其实 hub 在 usb 世界里扮演的又何尝不是这种角色呢?我们来看这个循环,这是 Hub 驱动中最核心的代码,然而这段代码却自始至终是在为别的设备服务.

在这个循环中,主要涉及这么八个重量级函数,先点明它们的角色分工.

第一个函数,usb_alloc_dev(),为一个 struct usb_device 结构体指针,申请内存,这个结构体指针可不是为 hub 准备的,它正是为了 hub 这个端口所接的设备而申请的,别忘了我们此时此刻的

上下文,之所以进入到了这个循环,是因为我们的 Hub 检测到某个端口有设备连接了进来,所以我们作为 Hub 驱动当然就义不容辞的要为该设备做点什么.

第二个函数,usb_set_device_state(),这个函数用来设置设备的状态,struct usb_device 结构体中,有一个成员,enum usb_device_state state,这一刻,会把这个设备的状态设置为 USB_STATE_POWERED,即上电状态.

第三个函数,choose_address(),为设备选择一个地址.一会咱们会用实例来看看效果.

第四个函数,hub_port_init(),不多说了,这个就是端口初始化,主要就是前面说的获取设备的描述符.

第五个函数,usb_get_status(),这个函数是专门为 hub 准备的,不是为当前的这个 hub,而是说当前 hub 的这个端口上连接的如果又是 hub,那么和连接普通设备当然不一样.

第六个函数,check_highspeed(),不同速度的设备当然待遇不一样.

第七个函数,usb_new_device().寻找驱动程序,调用驱动程序的 probe,跟踪这个函数就能一直跟踪到设备驱动程序的 probe()函数的调用.

第八个函数,hub_power_remaining(),别忘了,电源管理将是 hub 驱动永恒的话题.

Ok,下面就让我们来认真的逐个看一看这八大函数.

usb_alloc_dev(),drivers/usb/core/usb.c:

```
226 /**
227  * usb_alloc_dev - usb device constructor (usbcore-internal)
228  * @parent: hub to which device is connected; null to allocate a root hub
229  * @bus: bus used to access the device
230  * @port1: one-based index of port; ignored for root hubs
231  * Context: !in_interrupt()
232  *
233  * Only hub drivers (including virtual root hub drivers for host
234  * controllers) should ever call this.
235  *
236  * This call may not be used in a non-sleeping context.
237  */
238 struct usb_device *
239 usb_alloc_dev(struct usb_device *parent, struct usb_bus *bus, unsigned
port1)
240 {
241         struct usb_device *dev;
242
243         dev = kzalloc(sizeof(*dev), GFP_KERNEL);
```

```
244         if (!dev)
245                 return NULL;
246
247         if (!usb_get_hcd(bus_to_hcd(bus))) {
248                 kfree(dev);
249                 return NULL;
250         }
251
252         device_initialize(&dev->dev);
253         dev->dev.bus = &usb_bus_type;
254         dev->dev.type = &usb_device_type;
255         dev->dev.dma_mask = bus->controller->dma_mask;
256         dev->state = USB_STATE_ATTACHED;
257
258         INIT_LIST_HEAD(&dev->ep0.urb_list);
259         dev->ep0.desc.bLength = USB_DT_ENDPOINT_SIZE;
260         dev->ep0.desc.bDescriptorType = USB_DT_ENDPOINT;
261         /* ep0 maxpacket comes later, from device descriptor */
262         dev->ep_in[0] = dev->ep_out[0] = &dev->ep0;
263
264         /* Save readable and stable topology id, distinguishing devices
265          * by location for diagnostics, tools, driver model, etc.  The
266          * string is a path along hub ports, from the root.  Each device's
267          * dev->devpath will be stable until USB is re-cabled, and hubs
268          * are often labeled with these port numbers.  The bus_id isn't
269          * as stable:  bus->busnum changes easily from modprobe order,
270          * cardbus or pci hotplugging, and so on.
271          */
272         if (unlikely(!parent)) {
273                 dev->devpath[0] = '0';
274
275                 dev->dev.parent = bus->controller;
276                 sprintf(&dev->dev.bus_id[0], "usb%d", bus->busnum);
277         } else {
278                 /* match any labeling on the hubs; it's one-based */
279                 if (parent->devpath[0] == '0')
280                         snprintf(dev->devpath, sizeof dev->devpath,
281                                 "%d", port1);
282                 else
283                         snprintf(dev->devpath, sizeof dev->devpath,
284                                 "%s.%d", parent->devpath, port1);
285
286                 dev->dev.parent = &parent->dev;
287                 sprintf(&dev->dev.bus_id[0], "%d-%s",
```

```
    288                         bus->busnum, dev->devpath);
    289
    290                 /* hub driver sets up TT records */
    291         }
    292
    293         dev->portnum = port1;
    294         dev->bus = bus;
    295         dev->parent = parent;
    296         INIT_LIST_HEAD(&dev->filelist);
    297
    298 #ifdef  CONFIG_PM
    299         mutex_init(&dev->pm_mutex);
    300                                 INIT_DELAYED_WORK(&dev->autosuspend,
usb_autosuspend_work);
    301         dev->autosuspend_delay = usb_autosuspend_delay * HZ;
    302 #endif
    303         return dev;
    304 }
```

首先是申请内存,申请一个 struct usb_device 结构体指针的内存,这个指针就是 dev,bus 就是总线,这里传递进来的两个实参我们看到,一个是 hdev,一个是 hdve->bus,hdev 自然很明确,就是 hub 所对应的那个 struct usb_device 指针,它的 bus 当然更加明确,一个主机控制器就对应一条总线,那么不管我们在哪里说,都是那条总线.bus 是 struct usb_bus 结构体指针,而 bus_to_hcd()得到的就是该总线对应的主机控制器,江湖上的人都知道,主机控制器就是代表一条总线.主机控制器由一个结构体 struct usb_hcd 表示,struct usb_hcd 定义于 drivers/usb/core/hcd.c,它有一个成员,struct usb_bus self.那么这里 bus_to_hcd()就是得到该 bus 所属于的那个 struct usb_hcd 所对应的指针,usb_get_hcd()是增加引用计数,具体怎么实现咱们在设备模型里面有过交待.这里很显然,因为这个主机控制器的总线上多了一个设备,当然得为主机控制器增加引用计数,只要有设备在,主机控制器的数据结构就得在,否则系统肯定挂了.这里如果 usb_get_hcd()失败了那就没啥好说的了,释放内存吧,哪凉快去哪待着.

然后 device_initialize(&dev->dev),初始化设备,第一个 dev 是 struct usb_device 结构体指针,而第二个 dev 是 struct device 结构体,这是设备模型里面一个最最基本的结构体,使用它必然要先初始化,device_initialize 就是内核提供给咱们的初始化函数.

接下来,关于 struct device 结构体,我们迫不得已必须讲两句了,熟悉 2.6 设备模型的兄弟们一定知道 struct device.我们其实已经见过很多次了,在 usb-storage 的故事里见过,在 hub 的故事里此前也见过,只是一直没有把它的定义贴出来,因为那时候总觉得没有必要,但现在,关于这个结构体的成员出镜率越来越高,我不得不把它的定义贴出来.正如<<无间道>>里所说的那样,从来只有事情改变人,人不可能改变事情.这个结构体定义于 include/linux/device.h:

```
    410 struct device {
    411         struct klist            klist_children;
    412           struct klist_node       knode_parent;           /* node in
sibling list */
```

```
413         struct klist_node       knode_driver;
414         struct klist_node       knode_bus;
415         struct device           *parent;
416
417         struct kobject kobj;
418         char    bus_id[BUS_ID_SIZE];    /* position on parent bus */
419         struct device_type      *type;
420         unsigned                is_registered:1;
421         unsigned                uevent_suppress:1;
422         struct device_attribute uevent_attr;
423         struct device_attribute *devt_attr;
424
425         struct semaphore        sem;    /* semaphore to synchronize
calls to
426                                         * its driver.
427                                         */
428
429         struct bus_type * bus;          /* type of bus device is on */
430         struct device_driver *driver;   /* which driver has allocated this
431                                            device */
432         void            *driver_data;   /* data private to the driver */
433         void            *platform_data; /* Platform specific data, device
434                                            core doesn't touch it */
435         struct dev_pm_info      power;
436
437 #ifdef CONFIG_NUMA
438         int             numa_node;      /* NUMA node this device is
close to */
439 #endif
440         u64             *dma_mask;      /* dma mask (if dma'able
device) */
441         u64             coherent_dma_mask;/* Like dma_mask, but for
442                                              alloc_coherent mappings as
443                                              not all hardware supports
444                                                      64 bit addresses for
consistent
445                                              allocations such descriptors.
*/
446
447         struct list_head        dma_pools;      /* dma pools (if dma'ble)
*/
448
449         struct dma_coherent_mem *dma_mem; /* internal for coherent
mem
```

```
450                                             override */
451         /* arch specific additions */
452         struct dev_archdata     archdata;
453
454         spinlock_t              devres_lock;
455         struct list_head        devres_head;
456
457         /* class_device migration path */
458         struct list_head        node;
459         struct class            *class;
460         dev_t                   devt;           /* dev_t, creates the
sysfs "dev" */
461         struct attribute_group  **groups;       /* optional groups */
462
463         void    (*release)(struct device * dev);
464 };
```

又是一个暴长的结构体,所以说其实当初我不贴出来也是为了你好.

对着这个结构体来解释代码,dev.bus 就是 struct bus_type 的指针,bus_type 顾名思义就是总线类型,对于 usb 设备来说,你可以搜索整个 drivers/usb/目录下的代码,你会发现,其实总共就是定义过两个类型,一个是定义于 drivers/usb/core/driver.c 中的 struct bus_type usb_bus_type,另一个是定义于 drivers/usb/serial/usb-serial.c 中的 struct bus_type usb_serial_bus_type.后者干嘛用的不用我说了吧,整个drivers/usb/serial/目录就是专门为 usb-to-serial 而准备的,串口方面的,咱们不必理睬.所以在咱们的故事中,不管你在哪里看到 struct bus_type 的指针,它的指向都一样,就是 usb_bus_type. 到 drivers/usb/core/driver.c 里面看一下这个变量的定义.

```
1523 struct bus_type usb_bus_type = {
1524         .name =         "usb",
1525         .match =        usb_device_match,
1526         .uevent =       usb_uevent,
1527         .suspend =      usb_suspend,
1528         .resume =       usb_resume,
1529 };
```

实际上在 usb 系统的一开始的初始化代码就有这么一句,bus_register(&usb_bus_type),就这句让系统知道有这么一个类型的总线.不过需要强调的是,一个总线类型和一条总线可别混淆了,从硬件上来讲,一个 host controller 就会连出一条 usb 总线,而从软件上来讲,不管你有多少个 host controller,或者说有多少条总线,它们通通属于 usb_bus_type 这么一个类型,只是每一条总线对应一个 struct usb_bus 结构体变量,也正是前面我们刚刚说过的那个 struct usb_hcd 中那个叫做 self 的成员(struct usb_hcd 中 struct usb_bus self),这个变量在主机控制器的驱动程序中来申请.总线类型和具体的总线这种关系正如 C++中的类和对象,不要跟我说你不知道类和对象的关系.那我只能解释说,我身边有两类人,一类是有对象的,一类是没有对象的...

Ok,确定了 dev.bus 指向 usb_bus_type 之后,下一个 dev.type,struct device_type 结构体指针,这个就真的不用再解释了,其作用和 struct bus_type 一样,即总线有总线的类型,设备也有设备的类型,正所谓物以类聚.这里等号右边的 usb_device_type 来自 drivers/usb/core/usb.c,

```
195 struct device_type usb_device_type = {
196         .name =         "usb_device",
197         .release =      usb_release_dev,
198 };
```

现在我们还不需要去认识这个结构体内部,只是知道我们的所有的 usb 设备都会属于 usb_device_type 这么一类就可以了,这个只是为了让我们从软件的角度来说方便管理而已.

然后 dma_mask,这个就是与 DMA 传输相关的了,设备能不能进行 dma 传输,得看主机控制器的脸色,主机控制器不支持的话设备自作多情那也没有用.所以这里 dma_mask 被设置为 host controller 的 dma_mask.注意到我们在 struct usb_hcd 结构体里有一个 struct usb_bus self,而在 struct usb_bus 结构体里又有一个 struct device * controller,实际上 struct usb_hcd 和 struct usb_bus 都是为主机控制器而准备的数据结构,只是分工不一样,你要是看它们不爽也可以考虑把它们合并为一个结构体,不过我估计开源社区的那帮家伙不会同意你这样乱来.

256 行,dev->state,这里 dev 就是 struct usb_device 的意思了,而关于 state,我们必须很清楚,这是调试 usb 设备驱动的基础.你如果连 usb 设备有哪些状态都不知道的话,你不要跟人说你会调试 usb 设备驱动程序,真的,丢不起那人.struct usb_device 里面有一个成员,enum usb_device_state state,这是一个枚举的数据类型.

在 include/linux/usb/ch9.h 中有定义,

```
557 enum usb_device_state {
558         /* NOTATTACHED isn't in the USB spec, and this state acts
559          * the same as ATTACHED ... but it's clearer this way.
560          */
561         USB_STATE_NOTATTACHED = 0,
562
563         /* chapter 9 and authentication (wireless) device states */
564         USB_STATE_ATTACHED,
565         USB_STATE_POWERED,                  /* wired */
566         USB_STATE_UNAUTHENTICATED,          /* auth */
567         USB_STATE_RECONNECTING,             /* auth */
568         USB_STATE_DEFAULT,                  /* limited function */
569         USB_STATE_ADDRESS,
570         USB_STATE_CONFIGURED,               /* most functions
*/
571
572         USB_STATE_SUSPENDED
```

```
573
574             /* NOTE:   there are actually four different SUSPENDED
575              * states, returning to POWERED, DEFAULT, ADDRESS, or
576              * CONFIGURED respectively when SOF tokens flow again.
577              */
578 };
```

这些都是 usb 设备的可能状态,在 usb spec 中有定义,但是有些在 usb spec 里面没有,只是这里设置的,从软件的角度来说对 usb 设备的状态进行更细的划分.在 usb spec 2.0 中一共定义了 6 种状态,分别是 Attached,Powered,Default,Address,Configured,Suspended.那么咱们这里所谓的 USB_STATE_NOTATTACHED 就是表征设备并没有 Attached,而咱们此时此刻设置的是 USB_STATE_ATTACHED 就是做为当检测到设备时的初始状态.

端点 0 的特殊地位决定了她必将受到特殊的待遇.在 struct usb_device 结构体中就有一个成员 struct usb_host_endpoint ep0,而它的 urb_list 也是在这里就要先初始化了.

ep0.desc 就是该端点的描述符,端点描述符的长度总是 7 个字节,这是 usb spec 规定的,即 endpoint 的描述符中 bLength 为 7.宏 USB_DT_ENDPOINT_SIZE 就是被定义为 7.bDescriptorType 就是描述符类型,描述符既然有设备描述符,有接口描述符,有端点描述符,等等,当然就有一个类别的说法,要不怎么区分呢?如下图所示,usb spec 里定义了端点描述符的类别就是 5,

Table 9-5. Descriptor Types

Descriptor Types	Value
DEVICE	1
CONFIGURATION	2
STRING	3
INTERFACE	4
ENDPOINT	5
DEVICE_QUALIFIER	6
OTHER_SPEED_CONFIGURATION	7
INTERFACE_POWER[1]	8

而咱们的代码里定义了 USB_DT_ENDPOINT 这个宏就是 0x05,所以这里没什么好说的.一切按规矩办事.

ep_in 和 ep_out 就是两个数组,是 struct usb_device 中的两个成员,struct usb_host_endpoint *ep_in[16]和 struct usb_host_endpoint *ep_out[16],一个设备就是最多有 32 个管道,16 个进,16 个出,基本上一个端点对应一个管道.考你一下,知道我为什么不直接说一个设备最多有 32 个端点?理由很简单,端点 0 太太太太太特殊了,别的端点不是进就是出,但是端点 0 是双向的,既可以收也可以发.所以对它,我们需要以 ep_in[0]和 ep_out[0]来记录.

接下来的几个变量,parent 自不必说,没有它就没法形成一棵 usb 设备树,USB 设备树上的每一个设备都得有 parent,只有 Root Hub 没有,正所谓树欲静而风不止,子欲孝而亲不在.当我刚看到这段 if 语句的时候,觉得它真是荒唐,咱们现在不是在 hub 驱动程序中,在判断端口有连接设备了才执行这段代码的吗,hub 端口里面怎么会连接有 Root Hub 呢?但转念一想,不对,这种想法太幼稚了点,有一位伟人的话说,我这叫 too simple, sometimes naïve.的确我们是在这种情景下调用 usb_alloc_dev 的,可是这个函数并非只有我们调用啊,别人也会调用,在 host controller 的驱动中就会调用,它就要为 Root hub 申请内存啊,所以对于 usb_alloc_dev 函数来说当然就应该判断这个设备是不是 Root Hub 了.在 Host Controller 驱动程序中,调用这个函数的时候传递的 parent 就是 NULL,所以这里这个 if 就是用来区分这两种情况的.我们来仔细看一下这个 if 内部的赋值.

首先是 devpath,然后是 dev.bus_id,最后是 busnum.重点来看 devpath, struct usb_device 结构体中 char devpath[16],很显然这将会用来记录一个字符串,这个字符串啥意思?给你看个直观的东西,

localhost:~ # ls /sys/bus/usb/devices/

1-0:1.0 2-1 2-1:1.1 4-0:1.0 4-5:1.0 usb2 usb4

2-0:1.0 2-1:1.0 3-0:1.0 4-5 usb1 usb3

Sysfs 文件系统下,我们看到这些乱七八糟的东西,它们都是啥?usb1/usb2/usb3/usb4 表示哥们的计算机上接了 4 条 usb 总线,即 4 个 usb 主机控制器,事物多了自然就要编号,就跟我们中学或大学里面的学号一样,就是用于区分多个个体,而 4-0:1.0 表示什么?4 表示是 4 号总线,或者说 4 号 Root Hub,0 就是这里我们说的 devpath,1 表示配置为 1 号,0 表示接口号为 0.也即是说,4 号总线的 0 号端口的设备,使用的是 1 号配置,接口号为 0.那么 devpath 是否就是端口号呢?显然不是,这里我列出来的这个例子是只有 Root Hub 没有级联 Hub 的情况,如果在 Root Hub 上又接了别的 Hub,然后一级一级连下去,子又生孙,孙又生子,子又有子,子又有孙.子子孙孙,无穷匮也.那么如何在 sysfs 里面来表征这整个大家族呢?这就是 devpath 的作用,顶级的设备其 devpath 就是其连在 Root Hub 上的端口号,而次级的设备就是其父 Hub 的 devpath 后面加上其端口号,即如果 4-0:1.0 如果是一个 Hub,那么它下面的 1 号端口的设备就可以是 4-0.1:1.0,2 号端口的设备就可以是 4-0.2:1.0,3 号端口就可以是 4-0.3:1.0.总的来说,就是端口号一级一级往下加.这个思想是很简单的,也是很朴实的.

至此,我们可以来一次性看一下 272 行到 291 行这一段代码了.首先判断如果是 Root Hub,那么直接赋值 dev->devpath[0] 为 '0',以示特殊,然后父设备设为 controller,同时把 dev->bus_id[]设置为像 usb1/usb2/usb3/usb4 这样的字符串.如果不是 Root Hub,那么两种情况,第一种,如果是连接在 Root Hub 上,dev->path 就是等于端口号,第二种,如果不是连接

在 Root Hub 上,那么就是在父设备的 devpath 基础上加上一个端口号,在父设备的 devpath 和端口号之间用一个点”.”连接起来.最后把 bus_id[]设置成 usb1/usb2/usb3/usb4 这样的字符串后面连接上 devpath.

接下来 293 到 295 这三行就不用多说了,白痴都知道.

296 行,初始化一个队列.这个队列干嘛用的?usbfs 会用到,只要你够帅,你就能看到在/proc/bus/usb/下面有很多 usb 相关的信息.每个 usb 设备会在这下面对应一个文件,这个文件系统称为 USB 设备文件系统.你可以用下面这条命令挂载这个文件系统:

mount –t usbfs none /proc/bus/usb

然后你 cat /proc/bus/usb/devices 就能看到那些系统里 usb 设备的信息.对 usbfs 的实现在 drivers/usb/core/inode.c 以及 drivers/usb/core/devices.c 这两个文件中.现在把这个队列初始化为空,不久的将来,当您打开设备在 usbfs 中对应的文件时,这个队列就会加入东西.

298 行,又涉及到电源管理的部分了,首先是初始化一个互斥锁,pm_mutex,struct usb_device 中的成员,struct mutex pm_mutex.然后是调用 INIT_DELAYED_WORK(),这个咱们不陌生吧,前面专门讲过的.不过需要解释的是,autosuspend,自动挂起,在 struct usb_device 中也有一个成员,struct delayed_work autosuspend, 而 usb_autosuspend_work()是定义于 drivers/usb/core/driver.c 中的一个函数.先不管这个函数干嘛的,至少咱们通过这句话可以知道,在明天的明天的明天,在未来的某年某月某一天的某个时刻,usb_autosuspend_work()函数会被调用.具体是什么时刻,咱们先甭管,到时候再说.而 autosuspend_delay 就很简单了,就是延时,一样的,等到用的时候再来说,不过需要注意的是,这个冬冬被赋值为 usb_autosuspend_delay*HZ,其中 usb_autosuspend_delay 又是一个模块加载参数,

```
54 #ifdef  CONFIG_USB_SUSPEND
55 static int usb_autosuspend_delay = 2;             /* Default delay value,
56                                                   * in seconds */
57 module_param_named(autosuspend, usb_autosuspend_delay, int, 0644);
58 MODULE_PARM_DESC(autosuspend, "default autosuspend delay");
59
60 #else
61 #define usb_autosuspend_delay             0
62 #endif
```

如果您打开了 CONFIG_USB_SUSPEND,那么这个值默认就是 2,您当然可以自己修改,在您加载模块 usbcore 模块的时候作为一个参数传递进来就可以了.毕竟每个设备的具体情况不一样,可能 2 秒钟对大多数设备都适用,但也许对你的设备就不合适.你可以自己设置.这个得具体情况具体对待.即使同样是2秒钟,在不同的情景中可能意义不一样.比如你问一分钟有多长?这就先要看你是蹲在厕所里面,还是等在厕所外面.如果您没打开CONFIG_USB_SUSPEND,那么不必说,直接就是定义为 0,也没有意义.

Ok,到这里,我们走完了八大函数的第一个.路漫漫其修远兮,让我们荡起双桨上下而求索.

八大重量级函数闪亮登场(二)

第二个函数,usb_set_device_state(),鉴于网友”潜水潜到二零零八”提出 drivers/usb/core/hub.c 出镜频率过于的高,为避免被人成为新时期祥林嫂,经支部开会决定,从此以后凡是出自 drivers/usb/core/hub.c 这个文件的函数将不再做介绍其来源,这个就当是默认的位置.

```
1041 /**
1042  * usb_set_device_state - change a device's current state (usbcore, hcds)
1043  * @udev: pointer to device whose state should be changed
1044  * @new_state: new state value to be stored
1045  *
1046  * udev->state is _not_ fully protected by the device lock.  Although
1047  * most transitions are made only while holding the lock, the state can
1048  * can change to USB_STATE_NOTATTACHED at almost any time.  This
1049  * is so that devices can be marked as disconnected as soon as possible,
1050  * without having to wait for any semaphores to be released.  As a result,
1051  * all changes to any device's state must be protected by the
1052  * device_state_lock spinlock.
1053  *
1054  * Once a device has been added to the device tree, all changes to its state
1055  * should be made using this routine.  The state should _not_ be set
directly.
1056  *
1057  * If udev->state is already USB_STATE_NOTATTACHED then no change
is made.
1058  * Otherwise udev->state is set to new_state, and if new_state is
1059  * USB_STATE_NOTATTACHED then all of udev's descendants' states are
also set
1060  * to USB_STATE_NOTATTACHED.
1061  */
1062 void usb_set_device_state(struct usb_device *udev,
1063                 enum usb_device_state new_state)
1064 {
1065         unsigned long flags;
```

```
1066
1067        spin_lock_irqsave(&device_state_lock, flags);
1068        if (udev->state == USB_STATE_NOTATTACHED)
1069                ;       /* do nothing */
1070        else if (new_state != USB_STATE_NOTATTACHED) {
1071
1072                /* root hub wakeup capabilities are managed out-of-band
1073                 * and may involve silicon errata ... ignore them here.
1074                 */
1075                if (udev->parent) {
1076                        if (udev->state == USB_STATE_SUSPENDED
1077                                                || new_state ==
USB_STATE_SUSPENDED)
1078                                ;       /* No change to wakeup settings
*/
1079                        else if (new_state == USB_STATE_CONFIGURED)
1080                                device_init_wakeup(&udev->dev,
1081
(udev->actconfig->desc.bmAttributes
1082                                                                &
USB_CONFIG_ATT_WAKEUP));
1083                        else
1084                                device_init_wakeup(&udev->dev, 0);
1085                }
1086                udev->state = new_state;
1087        } else
1088                recursively_mark_NOTATTACHED(udev);
1089        spin_unlock_irqrestore(&device_state_lock, flags);
1090 }
```

天可怜见,这个函数不是很长,问题是,这个函数里面又调用了别的函数.

USB_STATE_NOTATTACHED,就是啥也没有,基本上就是说设备已经断开了,这种情况当然啥也不用做.

咱们刚才在 usb_alloc_dev 设置了等于 USB_STATE_ATTACHED,所以继续,new_state,结合实参看一下,传递的是 USB_STATE_POWERED,Root Hub 另有管理方式,我们这里首先就处理非 Root Hub 的情况,如果原来就是 USB_STATE_SUSPENDED,现还设置 USB_STATE_SUSPENDED,那么当然什么也不用做.如果新的状态要被设置为 USB_STATE_CONFIGURED,那么调用 device_init_wakeup(),初始化唤醒方面的东西,您要是和我一样,对电源管理不感兴趣,那么估计这里您不会被唤醒,您会进入睡眠.不过人在江湖,身不由己,如果能够退出江湖,我们都想退出,然而,任我行说过,有人的地方就有江湖,人就是江湖,你怎么退出?我们又如何退出呢?既然不能退出,那么只好硬着头皮前进.

要认识 device_init_wakeup()首先需要知道两个概念,can_wakeup 和 should_wakeup.这两个家伙从哪里钻出来的?看 struct device 结构体,里面有这么一个成员, struct dev_pm_info power,来看看 struct dev_pm_info,来自 include/linux/pm.h:

```
265 struct dev_pm_info {
266         pm_message_t            power_state;
267         unsigned                can_wakeup:1;
268 #ifdef  CONFIG_PM
269         unsigned                should_wakeup:1;
270         pm_message_t            prev_state;
271         void                    * saved_state;
272         struct device           * pm_parent;
273         struct list_head        entry;
274 #endif
275 };
```

这些都是电源管理部分的核心数据结构,显然我们没有必要深入研究,只是需要知道,can_wakeup 为 1 表明一个设备可以被唤醒,设备驱动为了支持 Linux 中的电源管理,有责任调用 device_init_wakeup()来初始化 can_wakeup.而 should_wakeup 则是在设备的电源状态发生变化的时候被 device_may_wakeup()用来测试,测试它该不该变化.因此 can_wakeup 表明的是一种能力,should_wakeup 表明的是有了这种能力以后去不去做某件事,就好比我们吵架的时候经常说,不是我打不过你,而是我不想打你.打得过是一种能力,但是有这种能力不一定就会去真的打,还得衡量该不该打.

我们给 device_init_wakeup()传递的参数是这个设备以及配置描述符中的 bmAttributes&USB_CONFIG_ATT_WAKEUP, 这是因为 ,USB spec 中规定了,bmAttributes,的 D5 表明的就是一个 USB 设备是否具有被唤醒的能力.如下图所示:

Table 9-10. Standard Configuration Descriptor (Continued)

Offset	Field	Size	Value	Description
7	*bmAttributes*	1	Bitmap	Configuration characteristics D7: Reserved (set to one) D6: Self-powered D5: Remote Wakeup D4...0: Reserved (reset to zero) D7 is reserved and must be set to one for historical reasons. A device configuration that uses power from the bus and a local source reports a non-zero value in *bMaxPower* to indicate the amount of bus power required and sets D6. The actual power source at runtime may be determined using the GetStatus(DEVICE) request (see Section 9.4.5). If a device configuration supports remote wakeup, D5 is set to one.

而 USB_CONFIG_ATT_WAKEUP 被定义为 1<<5,所以这里比较的就是 D5 是否为 1,为 1 就是说:"我能".

1083 这个 else 就是说,如果设备将要被设置的新状态又不是 USB_STATE_CONFIGUERD,那么就执行这里的 device_init_wakeup,这里第二个参数传递的是 0,就是说先不打开这个设备的 wakeup 能力.咱们这个上下文就是这种情况,咱们刚刚才说到,咱们的新状态就是 USB_STATE_POWERED.

直到 1086 行,才正式把咱们的状态设置为新的这个状态,对于我们这个上下文,那就是 USB_STATE_POWERED.

1087 行这里又是一个 else,那么很显然这个 else 的意思就是原来的状态不是 USB_STATE_NOTATTACHED 而现在要设置成 USB_STATE_NOTATTACHED.这又是一个递归函数.其作用就是其名字中所说的那样,递归的把各设备都设置成 NOTATTACHED 状态.

```
1028 static void recursively_mark_NOTATTACHED(struct usb_device *udev)
1029 {
1030         int i;
1031
1032         for (i = 0; i < udev->maxchild; ++i) {
1033                 if (udev->children[i])
1034
recursively_mark_NOTATTACHED(udev->children[i]);
1035         }
1036         if (udev->state == USB_STATE_SUSPENDED)
1037                 udev->discon_suspended = 1;
1038         udev->state = USB_STATE_NOTATTACHED;
1039 }
```

这段代码真的是太简单了,属于教科书般的递归函数的经典例子.就是每一个设备遍历自己的子节点,一个个调用 recursively_mark_NOTATTACHED() 函数把其 state 设置为 USB_STATE_NOTATTACHED.如果设备的状态处于 USB_STATE_SUSPENDED,那么设置 udev->discon_suspended 为 1,struct usb_device 中的一个成员,unsigned discon_suspended,含义很明显,表征 Disconnected while suspended.这里这么一设置,暂时我们还不知道有什么用,不过到时候我们就会在电源管理部分的代码里看到判断这个 flag 了,很显然设置了这个 flag 就会阻止 suspend 相关的代码被调用,咱们走着瞧.

Ok,第二个函数就这么轻轻松松搞定!

八大重量级函数闪亮登场(三)

在开始第三个函数前,2492 行至 2494 行还有三行代码,对 udev 中的 speed,bus_mA,level 进行赋值.

先说一下,bus_mA,struct usb_device 中的成员,unsigned short bus_mA,记录的是能够从总线上获得的电流,毫无疑问就是咱们前面算出来的 hub 上的那个 mA_per_port.上头能给多少咱们就要多少.

再说 level,级别,表征 usb 设备树的级连关系.Root Hub 当然其 level 就是 0,其下面一层就是 level 1,再下面一层就是 level 2,依此类推.

然后说 speed,include/linux/usb/ch9.h 中定义了这么一个枚举类型的变量:

```
548 /* USB 2.0 defines three speeds, here's how Linux identifies them */
549
550 enum usb_device_speed {
551         USB_SPEED_UNKNOWN = 0,                  /* enumerating */
552         USB_SPEED_LOW, USB_SPEED_FULL,          /* usb 1.1 */
553         USB_SPEED_HIGH,                         /* usb 2.0 */
554         USB_SPEED_VARIABLE,                     /* wireless (usb 2.5)
*/
555 };
```

很明显的含义,用来标志设备的速度.众所周知,USB 设备有三种速度,低速,全速,高速.USB1.1 那会儿只有低速,全速,后来才出现了高速,高速就是所谓的 480Mbps,不过细心的你或许注意到这里还有一个 USB_SPEED_VARIABLE.两千零五年那个五月,Intel 等公司推出了 Wireless USB spec 1.0 版,即所谓的无线 USB 技术,江湖上也把这个 usb 技术称为 usb 2.5.无线技术的推出必然会让设备的速度不再稳定,当年这个标准推出的时候是号称在 3 米范围内,能够提供 480Mbps 的理论传输速度,而在 10 米范围左右出现递减,据说是 10 米内 110Mbps.那时正值英特尔中国 20 周年,所以中国这边的员工每人发了一个无线 USB 鼠标.其实就是一个 USB 接头,接在电脑的 usb 端口上,而鼠标这边没有线,鼠标和接头之间的通信是无线的,使用传说中蓝牙技术.我的那个无线鼠标基本上四五米之外就不能用了.总之,这里的变量 usb_device_speed 就是用来表征设备速度的,现阶段还不知道这个设备究竟是什么速度的,所以先设置为 UNKNOWN.等到知道了以后再进行真正的设置.

第三个函数,choose_address().

这个函数的目的就是为设备选择一个地址.很显然,要通信就要有地址,你要给人写情书表白,你首先得知道人家的通信地址,或者电子邮箱地址.

```
1132 static void choose_address(struct usb_device *udev)
1133 {
1134         int             devnum;
1135         struct usb_bus  *bus = udev->bus;
1136
1137         /* If khubd ever becomes multithreaded, this will need a lock */
1138
1139                 /* Try to allocate the next devnum beginning at
bus->devnum_next. */
```

```
1140        devnum = find_next_zero_bit(bus->devmap.devicemap, 128,
1141                        bus->devnum_next);
1142        if (devnum >= 128)
1143                devnum = find_next_zero_bit(bus->devmap.devicemap,
128, 1);
1144
1145        bus->devnum_next = ( devnum >= 127 ? 1 : devnum + 1);
1146
1147        if (devnum < 128) {
1148                set_bit(devnum, bus->devmap.devicemap);
1149                udev->devnum = devnum;
1150        }
1151 }
```

那么现在是时候让我们来认识一下 usb 子系统里面关于地址的游戏规则了.在悠悠岁月中,一个神话就是浪花一朵,一滴苦酒就是史书一册,而在usb世界里,一条总线就是大树一棵,一个设备就是叶子一片.为了记录这棵树上的每一个叶子节点,每条总线设有一个地址映射表,即 struct usb_bus 结构体里有一个成员 struct usb_devmap devmap,

```
268 /* USB device number allocation bitmap */
269 struct usb_devmap {
270        unsigned long devicemap[128 / (8*sizeof(unsigned long))];
271 };
```

同时 struct usb_bus 结构体里面还有一个成员,int devnum_next,在总线初始化的时候,其 devnum_next 被设置为 1,而在 struct usb_device 中有一个 int devnum,咱们这个 choose_address 函数的基本思想就是一个轮询的算法.

我们来介绍一下这段代码背后的哲学.首先,bus 上面不是有这么一张表嘛,假设 unsigned long=4bytes,那么 unsigned long devicemap[128/(8*sizeof(unsigned long))]就等价于 unsigned long devicemap[128/(8*4)],进而等价于 unsigned long devicemap[4],而 4bytes 就是 32 个 bits,因此这个数组最终表示的就是 128 个 bits.而这也对应于一条总线可以连接 128 个 usb 设备.之所以这里使用 sizeof(unsigned long),就是为了跨平台应用,不管 unsigned long 到底是几,总之这个 devicemap 数组最终可以表示 128 位.

128 个 bits 里,每当加入一个设备,就先找到下一位为 0 的 bit,然后把该 bit 设置为 1,同时把 struct usb_device 中的 devnum 设置为该数字,比如我们找到第 19 位为 0,那么就把 devnum 设置为 19,同时把 bit 19 设置为 1,而 struct usb_bus 中的 devnum_next 就设置为 20.

那么所谓轮询,即如果这个编号超过了 128,那么就从 1 开始继续搜索,因为也许开始那段的号码原来分配给某个设备但后来这个设备撤掉了,所以这个号码将被设置为 0,于是再次可用.

弄清楚了这些基本思想后,我们再来看代码就很简单了.这时候相信你可以自豪的和杨振宁先生一样,高呼:"我能!"当然,我不会像翁帆女士一样喊:"你不能,我能让你能!"

find_next_zero_bit()的意思很明显,名字解释了一切.不同的体系结构提供了自己不同的函数实现,比如 i386,这个函数就定义于 arch/i386/lib/bitops.c 中,而 x8664 则对应于 arch/x86_64/lib/bitops.c中,利用这个函数我们就可以找到这128位中下一个为0的那一位.这个函数的第三个参数表征从哪里开始寻找,我们注意到第一次我们是从devnum_next开始找,如果最终返回值暴掉了(大于或者等于 128),那么就从 1 开始再找一次.而 bus->devnum_next 也是按我们说的那样设置,正常就是 devnum+1,但如果 devnum 已经达到 127 了,那么就从头再来,设置为 1.

如果 devnum 正常,那么就把 bus 中的 device map 中的那一位设置为 1.同时把 udev->devnum 设置为 devnum.然后这个函数就可以返回了.如果 128 个 bits 都被占用了,devnum 就将是零或者负的错误码,于是 choose_address 返回之后我们就要进行判断,如果真的是满掉了,那么咱们有心杀贼无力回天,也就不用往下走了,不过你要真的能连满 128 个设备,那你也蛮狠的.不得不把陈小春的那首<<算你狠>>送给你!

八大重量级函数闪亮登场(四)

接下来我们来到了第四个函数,hub_port_init().这个函数和接下来要遇到的usb_new_device()是最重要的两个函数,也是相对复杂的函数.

```
2096 /* Reset device, (re)assign address, get device descriptor.
2097  * Device connection must be stable, no more debouncing needed.
2098  * Returns device in USB_STATE_ADDRESS, except on error.
2099  *
2100  * If this is called for an already-existing device (as part of
2101  * usb_reset_device), the caller must own the device lock.  For a
2102  * newly detected device that is not accessible through any global
2103  * pointers, it's not necessary to lock the device.
2104  */
2105 static int
2106 hub_port_init (struct usb_hub *hub, struct usb_device *udev, int port1,
2107                int retry_counter)
2108 {
2109         static DEFINE_MUTEX(usb_address0_mutex);
2110
2111         struct usb_device       *hdev = hub->hdev;
2112         int                     i, j, retval;
2113         unsigned                delay = HUB_SHORT_RESET_TIME;
2114         enum usb_device_speed   oldspeed = udev->speed;
2115         char                    *speed, *type;
2116
2117         /* root hub ports have a slightly longer reset period
2118          * (from USB 2.0 spec, section 7.1.7.5)
2119          */
2120         if (!hdev->parent) {
```

```
2121                delay = HUB_ROOT_RESET_TIME;
2122                if (port1 == hdev->bus->otg_port)
2123                        hdev->bus->b_hnp_enable = 0;
2124        }
2125
2126        /* Some low speed devices have problems with the quick delay, so
*/
2127        /*  be a bit pessimistic with those devices. RHbug #23670 */
2128        if (oldspeed == USB_SPEED_LOW)
2129                delay = HUB_LONG_RESET_TIME;
2130
2131        mutex_lock(&usb_address0_mutex);
2132
2133        /* Reset the device; full speed may morph to high speed */
2134        retval = hub_port_reset(hub, port1, udev, delay);
2135        if (retval < 0)         /* error or disconnect */
2136                goto fail;
2138        retval = -ENODEV;
2139
2140             if (oldspeed != USB_SPEED_UNKNOWN && oldspeed !=
udev->speed) {
2141                dev_dbg(&udev->dev, "device reset changed speed!\n");
2142                goto fail;
2143        }
2144        oldspeed = udev->speed;
2145
2146        /* USB 2.0 section 5.5.3 talks about ep0 maxpacket ...
2147         * it's fixed size except for full speed devices.
2148         * For Wireless USB devices, ep0 max packet is always 512 (tho
2149         * reported as 0xff in the device descriptor). WUSB1.0[4.8.1].
2150         */
2151        switch (udev->speed) {
2152        case USB_SPEED_VARIABLE:         /* fixed at 512 */
2153                                udev->ep0.desc.wMaxPacketSize   =
__constant_cpu_to_le16(512);
2154                break;
2155        case USB_SPEED_HIGH:             /* fixed at 64 */
2156                                udev->ep0.desc.wMaxPacketSize   =
__constant_cpu_to_le16(64);
2157                break;
2158        case USB_SPEED_FULL:             /* 8, 16, 32, or 64 */
2159                /* to determine the ep0 maxpacket size, try to read
2160                 * the device descriptor to get bMaxPacketSize0 and
2161                 * then correct our initial guess.
```

```
2162                 */
2163                                udev->ep0.desc.wMaxPacketSize   =
__constant_cpu_to_le16(64);
2164                 break;
2165         case USB_SPEED_LOW:             /* fixed at 8 */
2166                                udev->ep0.desc.wMaxPacketSize   =
__constant_cpu_to_le16(8);
2167                 break;
2168         default:
2169                 goto fail;
2170         }
2171
2172         type = "";
2173         switch (udev->speed) {
2174         case USB_SPEED_LOW:      speed = "low";   break;
2175         case USB_SPEED_FULL:     speed = "full"; break;
2176         case USB_SPEED_HIGH:     speed = "high"; break;
2177         case USB_SPEED_VARIABLE:
2178                                 speed = "variable";
2179                                 type = "Wireless ";
2180                                 break;
2181         default:                speed = "?";     break;
2182         }
2183         dev_info (&udev->dev,
2184                  "%s %s speed %sUSB device using %s and address
%d\n",
2185                 (udev->config) ? "reset" : "new", speed, type,
2186                                  udev->bus->controller->driver->name,
udev->devnum);
2187
2188         /* Set up TT records, if needed  */
2189         if (hdev->tt) {
2190                 udev->tt = hdev->tt;
2191                 udev->ttport = hdev->ttport;
2192         } else if (udev->speed != USB_SPEED_HIGH
2193                         && hdev->speed == USB_SPEED_HIGH) {
2194                 udev->tt = &hub->tt;
2195                 udev->ttport = port1;
2196         }
2197
2198          /* Why interleave GET_DESCRIPTOR and SET_ADDRESS this
way?
2199          * Because device hardware and firmware is sometimes buggy in
2200          * this area, and this is how Linux has done it for ages.
```

```
2201          * Change it cautiously.
2202          *
2203          * NOTE:   If USE_NEW_SCHEME() is true we will start by issuing
2204          * a 64-byte GET_DESCRIPTOR request.  This is what Windows
does,
2205          * so it may help with some non-standards-compliant devices.
2206          * Otherwise we start with SET_ADDRESS and then try to read the
2207          * first 8 bytes of the device descriptor to get the ep0 maxpacket
2208          * value.
2209          */
2210         for (i = 0; i < GET_DESCRIPTOR_TRIES; (++i, msleep(100))) {
2211                 if (USE_NEW_SCHEME(retry_counter)) {
2212                         struct usb_device_descriptor *buf;
2213                         int r = 0;
2214
2215 #define GET_DESCRIPTOR_BUFSIZE   64
2216                             buf = kmalloc(GET_DESCRIPTOR_BUFSIZE,
GFP_NOIO);
2217                         if (!buf) {
2218                                 retval = -ENOMEM;
2219                                 continue;
2220                         }
2221
2222                         /* Retry on all errors; some devices are flakey.
2223                          * 255 is for WUSB devices, we actually need to
use
2224                          * 512 (WUSB1.0[4.8.1]).
2225                          */
2226                         for (j = 0; j < 3; ++j) {
2227                                 buf->bMaxPacketSize0 = 0;
2228                                         r = usb_control_msg(udev,
usb_rcvaddr0pipe(),
2229                                             USB_REQ_GET_DESCRIPTOR,
USB_DIR_IN,
2230                                             USB_DT_DEVICE << 8, 0,
2231                                             buf, GET_DESCRIPTOR_BUFSIZE,
2232                                             USB_CTRL_GET_TIMEOUT);
2233                                 switch (buf->bMaxPacketSize0) {
2234                                 case 8: case 16: case 32: case 64: case
255:
2235                                         if (buf->bDescriptorType ==
2236                                                         USB_DT_DEVICE)
{
2237                                                 r = 0;
```

```
2238                                            break;
2239                                    }
2240                                    /* FALL THROUGH */
2241                            default:
2242                                    if (r == 0)
2243                                            r = -EPROTO;
2244                                    break;
2245                            }
2246                            if (r == 0)
2247                                    break;
2248                    }
2249                    udev->descriptor.bMaxPacketSize0 =
2250                                    buf->bMaxPacketSize0;
2251                    kfree(buf);
2252
2253                    retval = hub_port_reset(hub, port1, udev, delay);
2254                    if (retval < 0)          /* error or disconnect */
2255                            goto fail;
2256                    if (oldspeed != udev->speed) {
2257                            dev_dbg(&udev->dev,
2258                                    "device reset changed speed!\n");
2259                            retval = -ENODEV;
2260                            goto fail;
2261                    }
2262                    if (r) {
2263                            dev_err(&udev->dev, "device descriptor "
2264                                            "read/%s, error %d\n",
2265                                            "64", r);
2266                            retval = -EMSGSIZE;
2267                            continue;
2268                    }
2269 #undef GET_DESCRIPTOR_BUFSIZE
2270            }
2271
2272            for (j = 0; j < SET_ADDRESS_TRIES; ++j) {
2273                    retval = hub_set_address(udev);
2274                    if (retval >= 0)
2275                            break;
2276                    msleep(200);
2277            }
2278            if (retval < 0) {
2279                    dev_err(&udev->dev,
2280                            "device not accepting address %d, error
%d\n",
```

```
2281                            udev->devnum, retval);
2282                    goto fail;
2283            }
2284
2285            /* cope with hardware quirkiness:
2286              *  - let SET_ADDRESS settle, some device hardware wants it
2287               *  - read ep0 maxpacket even for high and low speed,
2288               */
2289            msleep(10);
2290            if (USE_NEW_SCHEME(retry_counter))
2291                    break;
2292
2293            retval = usb_get_device_descriptor(udev, 8);
2294            if (retval < 8) {
2295                    dev_err(&udev->dev, "device descriptor "
2296                                    "read/%s, error %d\n",
2297                                    "8", retval);
2298                    if (retval >= 0)
2299                            retval = -EMSGSIZE;
2300            } else {
2301                    retval = 0;
2302                    break;
2303            }
2304    }
2305    if (retval)
2306            goto fail;
2307
2308    i = udev->descriptor.bMaxPacketSize0 == 0xff?
2309        512 : udev->descriptor.bMaxPacketSize0;
2310    if (le16_to_cpu(udev->ep0.desc.wMaxPacketSize) != i) {
2311            if (udev->speed != USB_SPEED_FULL ||
2312                            !(i == 8 || i == 16 || i == 32 || i == 64)) {
2313                    dev_err(&udev->dev, "ep0 maxpacket = %d\n", i);
2314                    retval = -EMSGSIZE;
2315                    goto fail;
2316            }
2317            dev_dbg(&udev->dev, "ep0 maxpacket = %d\n", i);
2318            udev->ep0.desc.wMaxPacketSize = cpu_to_le16(i);
2319            ep0_reinit(udev);
2320    }
2321
```

```
2322                            retval = usb_get_device_descriptor(udev,
USB_DT_DEVICE_SIZE);
2323            if (retval < (signed)sizeof(udev->descriptor)) {
2324                    dev_err(&udev->dev, "device descriptor read/%s, error
%d\n",
2325                            "all", retval);
2326                    if (retval >= 0)
2327                            retval = -ENOMSG;
2328                    goto fail;
2329            }
2330
2331            retval = 0;
2332
2333 fail:
2334            if (retval)
2335                    hub_port_disable(hub, port1, 0);
2336            mutex_unlock(&usb_address0_mutex);
2337            return retval;
2338 }
```

像这种近 300 行的函数,这些年来,我竟也渐渐习惯了.我现在是以一种看文物的心态看这些变态的函数,竟也慢慢品出了一点周口店遗风.因为我明白,即便我像鲁迅先生那样呐喊,写代码的那些家伙也不会停下写这些函数的手.难怪江湖上称这些人的所作所为为行为艺术.

hub_port_init()这个函数的基本思想就是做初始化,首先是把一个设备 reset,然后是分配地址.在然后是获得设备描述符.

首先 DEFINE_MUTEX 是来自于 include/linux/mutex.h 中的一个宏,用它可以定义一把互斥锁,在 Linux 内核中,其实是在 2005 年底才建立比较系统的完善的互斥锁机制,在那年冬天,北京的最后一场雪过后,来自 RedHat 公司的 Ingo Molnar 大侠大胆的提出了他所谓的 Generic Mutex Subsystem,即通用的互斥锁机制.此前内核中很多地方使用的都是信号量,正如我们在 2.6.10 内核中 usb-storage 中所看到的那样,而当时间的箭头指向了 2005 年末的时候,区里(开源社区,下称区里)很多同志抱怨说信号量不灵,很多时候不好用,当然区里为这事展开了一场轰轰烈烈的讨论.老黑客 Ingo Molnar 受不了了,在一周之后,愤然提出要对内核进行一场大的革命,这次革命后来被称为一二一九运动.当时 Ingo 同志提出了诸多理由要求使用新的互斥锁机制,而不是过去普遍出现在内核中的信号量机制,比如新的机制占用更小的内存,代码更为紧凑,更快,更便于调试.在诸多优势的诱惑下,区里的群众将信将疑的便认可了这种做法.忽如一夜春风来,紧接着的几个月里,人民群众纷纷提交 patch,把原来用信号量的地方都改成了互斥锁.而这种改变深入到 Linux 中 usb 子系统是始于 2006 年春天,农历二月二十二,Greg 同志大旗一挥,将 usb 中的代码中绝大多数的信号量代码换成了互斥锁代码.所以到今天,您看 2.6.22.1 的代码中,整个 usb 子系统里几乎没有了 down/up 这一对函数的使用,取而代之的是 mutex_lock()和 mutex_unlock()函数对.而要初始化,只需像我们这里一样,DEFINE_MUTEX(name)即可.关于这个新的互斥锁的定义在 include/linux/mutex.h 中,而实现在 kernel/mutex.c 中.

Ok,让我们继续.hdev 被定义用来记录 hub 所对应的那个 struct usb_device,而 delay 记录延时,因为 usb 设备的 reset 工作不可能是瞬间的,通常会有一点点延时,这很容易理解,你的计算机永远不可能说你按一下 reset 键就立刻能够重起马上就能重新工作的,这里咱们首先给 delay 设置的初始值为 10ms,即 HUB_SHORT_RESET_TIME 这个宏被定义为 10ms.这个 10ms 的来源是 usb spec 2.0 中 7.1.7.5 节 Reset Signaling 里面说的,"The reset signaling must be driven for a minumum of 10ms",这个 10ms 在 usb spec 中称之为 T_{DRST}.一个 Hub 端口在 reset 之后将进入 Enabled 状态.

然后定义一个 oldspeed 用来记录设备在没有 reset 之前的速度.

2120 行,只有 root hub 没有父亲,而 usb spec 2.0 里说得很清楚,"It is required that resets from root ports have a duration of at least 50ms",这个 50ms 被称为 T_{DRSTR}.即 HUB_ROOT_RESET_TIME 这个宏被定义为 50ms.

2122 和 2123 行是 OTG 相关的,HNP 是 OTG 标准所支持的协议,HNP 即 Host Negotiation Protocol,坊间俗称主机通令协议.咱们既然不关注 OTG,那么这里也就不做解释了.省得把问题复杂化了.当断不断,反受其乱. – 史记 春申君列传.

2128 行,这两行代码来源于实践.实践表明,某些低速设备要求有比较高的延时才能完成好它们的 reset,这很简单,286 的机器重起肯定比 P4 的机器要慢.

好,调用 mutex_lock 获得互斥锁了,即表明下面这段代码一个时间只能被一个进程执行.

然后调用 hub_port_reset().

```
1509 static int hub_port_reset(struct usb_hub *hub, int port1,
1510                                 struct usb_device *udev, unsigned int
delay)
1511 {
1512         int i, status;
1513
1514         /* Reset the port */
1515         for (i = 0; i < PORT_RESET_TRIES; i++) {
1516                 status = set_port_feature(hub->hdev,
1517                                 port1, USB_PORT_FEAT_RESET);
1518                 if (status)
1519                         dev_err(hub->intfdev,
1520                                         "cannot reset port %d (err =
%d)\n",
1521                                         port1, status);
1522                 else {
1523                         status = hub_port_wait_reset(hub, port1, udev,
delay);
1524                         if (status && status != -ENOTCONN)
1525                                 dev_dbg(hub->intfdev,
```

```
1526                                                "port_wait_reset: err =
%d\n",
1527                                               status);
1528                    }
1529
1530                    /* return on disconnect or reset */
1531                    switch (status) {
1532                    case 0:
1533                            /* TRSTRCY = 10 ms; plus some extra */
1534                            msleep(10 + 40);
1535                            /* FALL THROUGH */
1536                    case -ENOTCONN:
1537                    case -ENODEV:
1538                            clear_port_feature(hub->hdev,
1539                                    port1, USB_PORT_FEAT_C_RESET);
1540                             /* FIXME need disconnect() for NOTATTACHED
device */
1541                            usb_set_device_state(udev, status
1542                                            ? USB_STATE_NOTATTACHED
1543                                            : USB_STATE_DEFAULT);
1544                            return status;
1545                    }
1546
1547                    dev_dbg (hub->intfdev,
1548                            "port %d not enabled, trying reset again...\n",
1549                            port1);
1550                    delay = HUB_LONG_RESET_TIME;
1551            }
1552
1553            dev_err (hub->intfdev,
1554                    "Cannot enable port %i.   Maybe the USB cable is bad?\n",
1555                    port1);
1556
1557            return status;
1558 }
```

事到如今,有些函数不讲也不行了.这就是 set_port_feature.其实之前我们遇见过,只是因为当时属于可讲可不讲,所以就先跳过去了.但现在不讲不行了,我们前面讲过它的搭档 clear_port_feature,所以我不讲你也应该知道 set_port_feature()干嘛用的,很显然,一个是清楚 feature,一个是设置 feature.Linux 中很多这种成对的函数,刚才讲的那个 mutex_lock()和 mutex_unlock()不也是这样么?其实这种思想是借鉴了我国的黄梅戏<<天仙配>>中所描绘的那种”你耕田来我织布,我挑水来你浇园,你我好比鸳鸯鸟,比翼双飞在人间”的纯朴的爱情观.

```
172 /*
173  * USB 2.0 spec Section 11.24.2.13
```

```
 174  */
 175 static int set_port_feature(struct usb_device *hdev, int port1, int feature)
 176 {
 177         return usb_control_msg(hdev, usb_sndctrlpipe(hdev, 0),
 178                 USB_REQ_SET_FEATURE, USB_RT_PORT, feature, port1,
 179                 NULL, 0, 1000);
 180 }
```

看明白了 clear_port_feature()的一定不会觉得这个函数看不懂.发送一个控制请求,设置一个 feature,咱们传递进来的 feature 是 USB_PORT_FEAT_RESET,即对应于 usb spec 中的 reset.发送好了之后就延时等待,调用 hub_port_wait_reset():

```
1457 static int hub_port_wait_reset(struct usb_hub *hub, int port1,
1458                                 struct usb_device *udev, unsigned int
delay)
1459 {
1460         int delay_time, ret;
1461         u16 portstatus;
1462         u16 portchange;
1463
1464         for (delay_time = 0;
1465                         delay_time < HUB_RESET_TIMEOUT;
1466                         delay_time += delay) {
1467                 /* wait to give the device a chance to reset */
1468                 msleep(delay);
1469
1470                 /* read and decode port status */
1471                         ret = hub_port_status(hub, port1, &portstatus,
&portchange);
1472                 if (ret < 0)
1473                         return ret;
1474
1475                 /* Device went away? */
1476                 if (!(portstatus & USB_PORT_STAT_CONNECTION))
1477                         return -ENOTCONN;
1478
1479                 /* bomb out completely if something weird happened */
1480                 if ((portchange & USB_PORT_STAT_C_CONNECTION))
1481                         return -EINVAL;
1482
1483                 /* if we`ve finished resetting, then break out of the loop */
1484                 if (!(portstatus & USB_PORT_STAT_RESET) &&
1485                     (portstatus & USB_PORT_STAT_ENABLE)) {
1486                         if (hub_is_wusb(hub))
1487                                 udev->speed = USB_SPEED_VARIABLE;
```

```
1488                                            else if (portstatus &
USB_PORT_STAT_HIGH_SPEED)
1489                                    udev->speed = USB_SPEED_HIGH;
1490                                            else if (portstatus &
USB_PORT_STAT_LOW_SPEED)
1491                                    udev->speed = USB_SPEED_LOW;
1492                            else
1493                                    udev->speed = USB_SPEED_FULL;
1494                            return 0;
1495                    }
1496
1497                    /* switch to the long delay after two short delay failures */
1498                    if (delay_time >= 2 * HUB_SHORT_RESET_TIME)
1499                            delay = HUB_LONG_RESET_TIME;
1500
1501                    dev_dbg (hub->intfdev,
1502                            "port %d not reset yet, waiting %dms\n",
1503                            port1, delay);
1504            }
1505
1506            return -EBUSY;
1507 }
```

这里 HUB_RESET_TIMEOUT 是设置的一个超时,这个宏的值为 500 毫秒,即如果 reset 了 500 毫秒还没好那么就返回错误值,朽木不可雕也.而循环的步长正是我们前面设置的那个 delay.

msleep(delay)就是休眠 delay 毫秒.

休眠完了就读取端口的状态.hub_port_status()不用说了,咱们前面讲过了.获得端口状态.错误就返回错误码,正确就把信息记录在 portstatus 和 portchange 里.

1476 行判断,如果在 reset 期间设备都被撤掉了,那就返回吧,甭浪费感情了.

1480 行判断,如果又一次汇报说有设备插入,那就是见鬼了.返回错误立刻向上级汇报说人鬼情未了.

正如刚才说过的,reset 真正完成以后,status 就应该是 enabled,所以 1484 和 1485 行的 if 如果满足就说明 reset 好了.

1486 行这个 if 是判断这是否是一个无线 Root hub,即既是 Root Hub,又是无线 hub,因为在 struct usb_hcd 中有一个成员 unsigned wireless,这个 flag 标志了该主机控制器是 Wireless 的.我们刚才说过了,Wireless 的话,speed 就是那个 USB_SPEED_VARIABLE.

否则如果 portstatus 和 USB_PORT_STAT_HIGH_SPEED 相与为 1,则是高速设备,如果与 USB_PORT_STAT_LOW_SPEED 相与为 1 则是低速设备,剩下来的可能就是全速设备.到这里,

这个 hub_port_wait_reset 就可以返回了,正常的返回 0.总之,注意,经过这里 udev 的 speed 就被设置好了.

1497 和 1498 行,走到这里就说明还没有 reset 好.如果已经过了 2 个 HUB_SHORT_RESET_TIME,就把步长设置为 HUB_LONG_RESET_TIME,即 200ms,然后打印一条警告信息,继续循环,如果循环完全结束还不行,那就说明超时了,返回-EBUSY.

回到 hub_port_reset 中来,下面走到了 1531 行,一个 switch,根据刚才的返回值做一次选择,如果是 0,说明正常, 安全起见,索性再等 50ms.如果是错误码,并且错误码表明设备不在了,则首先清掉 reset 这个 feature,然后把这个为刚才这个设备申请的 struct usb_device 结构体的状态设置为 USB_STATE_NOTATTACHED. 并且返回 status. 如果你问那个 USB_STATE_DEFAULT 是怎么回事?那么说明你没有好好学谭浩强的那本经典教材.在 switch 里面,如果 status 为 0,那么由于 case 0 那一部分后面 break 语句,所以 case –ENOTCONN 和 case –ENODEV 下面的那几句代码都会执行,即 clear_port_feature 是总会执行的,usb_set_device_state()也会执行,而对于 status 为 0 的情况,属于正常情况,从这时候开始,struct usb_device 结构体的状态就将记录为 USB_STATE_DEFAULT,即所谓的默认状态,然后返回值就是 0.而对于端口 reset,我们设置的重复次数是 PORT_RESET_TRIES,它等于 5,你当然可以把它改为 1,没人拦住你.只要你对自己的设备够自信,一次 reset 就肯定成功.信自己,金莱克!

如果 1553 行还会执行,那么说明肯定出了大问题了,reset 都没法进行,于是返回错误状态吧.

回到 hub_init_port()中来,如果刚才失败了,就 goto fail,否则也暂时将 retval 这个临时变量设置为-ENODEV,而 2140 行,如果 oldspeed 不是 USB_SPEED_UNKNOWN,并且也不等于刚刚设置的这个 speed,那么说明 reset 之前设备已经在工作了,而这次 reset 把设备原来的速度状态也给改变了.这是不合理的,必须结束函数,并且 disable 这个端口.于是 goto fail,而在 fail 那边可以看到,由于 retval 不为 0,所以调用 hub_port_disable()关掉这个端口.然后释放互斥锁,并且返回错误代码.

2144 行,如果不是刚才这种情况,那么令 oldspeed 等于现在这个 udev->speed.

2151 行开始,又是一个 switch,感觉这一段代码的选择出现得太多了点,代码的选择倒是简单,可是人,作为微小而孤独的个体,在人生的选择题前,则总会无可避免地徘徊起来.在一个又一个渡口上,在一次又一次险象中,我们究竟能选择什么,该选择什么?

其实这里是设置一个初始值,我们曾经介绍过 ep0.ep0.desc.wMaxPacketSizep0.desc.wMaxPacketSize 用来记录端点 0 的单个包的最大传输 size.对于无线设备,无线 usb spec 规定了,端点 0 的最大包 size 就是 512,而对于高速设备,这个 size 也是 usb spec 规定好了,64bytes,而低速设备同样是 usb spec 规定好了,8bytes.唯一存在变数的是全速设备,它可能是 8,可能是 16,可能是 32,也可能是 64,对于这种设备,没有办法,只能通过读取设备描述符来获得了.也正是这个全速设备引发了我们前面讨论的那个两种策略的问题.不过那时候我们没有点明说是这场 PK 是因为全速设备引起的,因为那时候说还太早了,说了您也忘了,而现在看到了代码就不会忘了.正如我们说过的,Linux 向 Windows 妥协了,这里 Full Speed 的设备,默认先把 wMaxPacketSize 设置为 64,而不是以前的那种 8.

2172 至 2186 这些行,仅仅是为了打印一行调试信息,对于写代码的人来说,可能很有用,而对读代码的人来说,也许就没有任何意义了,正如对聋子而言,正版唱片 CD 的作用只是拿来当照脸的镜子.

Sep 9 11:32:49 localhost kernel: usb 4-5: new high speed USB device using ehci_hcd and address 3

Sep 9 11:32:52 localhost kernel: usb 2-1: new low speed USB device using uhci_hcd and address 3

比如在我的计算机里,就可以在/var/log/messages 日志文件里看到上面这样的信息.ehci_hcd 和 uhci_hcd 就是主机控制器的驱动程序.这个 3 就是设备的 devnum.

2189 行,不是 switch 又是 if,除了判断还是判断,只不过刚才是选择,现在是如果,如果明天是世界末日,我就会去抢银行,可是既然明天是世界末日了,那要很多钱又能做什么呢?这里如果 hdev->tt 为真,则如何如何,否则,如果设备本身不是高速的,而 hub 是高速的,那么如何如何.tt 我们介绍过,transaction translator,而 ttport 是 struct usb_device 中的一个成员,int ttport,这个 ttport 以后会被用到,tt 也将在以后会被用到,暂时先不细说,到时候再回来看.

GET_DESCRIPTOR_TRIES 等于 2,这段代码的思想我们以前就讲过,USB_NEW_SCHEME(retry_counter),retry_counter 就是 hub_port_init()传递进来的最后一个参数,而我们给它的实参正是那个从 0 到 SET_CONFIG_TRIES-1 的那个 i.假设我们什么也没有设置,都是使用默认值,那么 use_both_schemes 默认值为 1,而 old_scheme_first 默认值为 0,于是 SET_CONFIG_TRIES 为 4,即 i 将从 0 变到 3,而 USB_NEW_SCHEME(i)将在 i 为 0 和 1 的时候为 1,在 i 为 2 和 3 的时候为 0.所以也就是说,先进行两次新的策略,如果不行就再进行两次旧的策略.所有这一切只有一个目的,就是为了获得设备的描述符.由于思想已经非常清楚,代码我们就不再一行一行讲了.尤其是那些错误判断的句子.

只是介绍一下其中调用的几个函数.对于新策略,首先定义一个 struct usb_device_descriptor 的指针 buf,然后申请 64 个字节的空间,发送一个控制传输的请求,然后结束之后,察看 buf->bMaxPackSize0,合理值只有 8/16/32/64/512,这里 255 实际上是 WUSB 协议规定的,毕竟只有 8 位,最大就是 255 了,所以就用这个值来代表 WUSB 设备.实际上 WUSB 的大小是 512.循环三次是保险起见.因为实践表明这类请求通常成功率很难达到 100%.

然后 2249 行用 udev->descriptor.bMaxPacketSize0 来记录这个临时获得的值.然后 buf 的使命结束了,释放它的内存.

然后正如我们曾经分析的那样,把设备 reset.

然后是设置地址.hub_set_address()

```
2076 static int hub_set_address(struct usb_device *udev)
2077 {
2078         int retval;
2079
```

```
2080        if (udev->devnum == 0)
2081                return -EINVAL;
2082        if (udev->state == USB_STATE_ADDRESS)
2083                return 0;
2084        if (udev->state != USB_STATE_DEFAULT)
2085                return -EINVAL;
2086        retval = usb_control_msg(udev, usb_sndaddr0pipe(),
2087                USB_REQ_SET_ADDRESS, 0, udev->devnum, 0,
2088                NULL, 0, USB_CTRL_SET_TIMEOUT);
2089        if (retval == 0) {
2090                usb_set_device_state(udev, USB_STATE_ADDRESS);
2091                ep0_reinit(udev);
2092        }
2093        return retval;
2094 }
```

和前面那个 choose_address 不同,choose_address 是从软件意义上挑选一个地址.而这里要发送真正的请求,因为设置设备地址本身就是 usb spec 2.0 规定的标准的请求之一,这里我们用宏 USB_REQ_SET_ADDRESS 来代替,只有真正发送了请求之后硬件上才能真正通过这个地址进行通信.这里最关键的就是传递了 udev->devnum,这正是我们前面选择的地址,这里赋给了 wIndex,来自 usb spec 2.0 中图表说明了一切:

9.4.6 Set Address

This request sets the device address for all future device accesses.

bmRequestType	bRequest	wValue	wIndex	wLength	Data
00000000B	SET_ADDRESS	Device Address	Zero	Zero	None

The *wValue* field specifies the device address to use for all subsequent accesses.

从此以后这个设备站起来了,并一举确立了它在 usb 江湖中的地位.

设好了地址之后,把设备的状态从开始的那个 USB_STATE_DEFAULT 变成了 USB_STATE_ADDRESS,从此以后他就算是有户口的主了.在 usb spec 中,这一状态被称为 Address state,我叫它有地址的状态.

然后调用了 ep0_reinit().

```
2066 static void ep0_reinit(struct usb_device *udev)
2067 {
2068        usb_disable_endpoint(udev, 0 + USB_DIR_IN);
2069        usb_disable_endpoint(udev, 0 + USB_DIR_OUT);
2070        udev->ep_in[0] = udev->ep_out[0] = &udev->ep0;
2071 }
```

usb_disable_endpoint 是 usbcore 提供的一个函数,具体到咱们这个端点,它会让 ep_out[0]和 ep_in[0]置为 NULL.然后会取消掉任何挂在 endpoint 上的 urb.再然后这里 2070 行再次把 ep_in[0]和 ep_out[0]指向 udev->ep0.须知 ep0 是真正占内存的数据结构,而 ep_in[0]和 ep_out[0]只是指针,而在 host controller 驱动程序里将被用到的正是这两个指针数组.你也许会问,这里为何要调用 ep0_reinit?在别的地方也有看见调用 ep0_reinit()的,为什么需要调用这个函数呢?听好了,我只说一遍,而且一般人我还不告诉他.首先,主机控制器通常会记录着每一个端点的状态,(否则它怎么知道如何跟每个端点通信?)而这个东西在每次设备状态发生变化了之后就要相应的作出变化,或者说 needs to be cleared out.具体来说,ep0 是一个 struct usb_host_endpoint 的结构体,这个结构体的变量都是为 host controller driver 来使用的,struct usb_host_endpoint 里有一个成员 struct list_head urb_list,也就是说,所有针对该 endpoint 的 urb 请求被排列成一个队列,在我们这个上下文里,也许我们的设备还没有任何 urb 请求,但是要知道 hub_set_address()这个函数是被 hub_port_init 调用,而 hub_port_init()并不是只有在我们这个上下文里被调用,也许下一次调用的时候,咱们这个 ep0 对应的 urb_list 里面已经有东西了,那么你这里重新设置一下地址,那么毫无疑问,你需要把原来的那些 urb 请求给清除掉.打个不太恰当的比方,自从上海市大学生电影节徐静蕾来了一趟复旦在相辉堂我近距离见到她之后,我就成为了一名徐静蕾的粉丝.可是来到北京之后,我一个清华的同学知道我特别喜欢徐静蕾,他偏跟我说,北京人都知道,徐静蕾换男朋友和她换卫生巾的频率差不多,这句话严重破坏了徐静蕾在我心中的玉女形象.当然,我们这里暂且不管这一说法是否属实,假设徐静蕾换了男朋友,那么她可能会把原来的那些肉麻短信都给删掉,哪怕有的她都还没有读过,每一次换都会如此.而这,正是 ep0_reinit()所做的,它会通知 host controller,由 host controller 来具体实施.

当一个设备没有被设置地址时,它使用默认地址,即地址 0,而当我们设置地址之后,这个地址发生了变化,所以主机控制器必须知道这件事情,否则他没法控制.就好比外企里面大家经常相互转些垃圾邮件,有一天我离开了 Intel,但我还希望同学和老同事继续给我转这些垃圾邮件,那么我至少得告诉人家我换邮箱了,然后对方就会相应的更新一下地址簿.同样,这里 usb_disable_endpoint()最终会调用 host controller driver 提供的函数,让 host controller driver 作出相应的反应.

回到 hub_port_init 中来,设置好了地址,然后 2289 行,先睡 10ms,然后如果还是新策略,那么就可以结束了,因为该做的都做完了.如果是旧策略,那么执行到这里还刚上路呢,我们说过了,思路就是先获得设备描述符的前 8 个字节,然后从中得知 udev->descriptor 的 bMaxPacketSize0,然后再次调用 usb_get_device_descriptor 完整的获得一次设备描述符.然后就 ok 了,到 2336 行,释放互斥锁,所谓的解铃还须系铃人.

至此,hub_port_init()就可以返回了.

声明一下,usb_get_device_descriptor()是来自 drivers/usb/core/message.c 中的函数,由 usbcore 提供,我们这里就不细讲了.其作用是很明显的,获取设备描述符.

再啰嗦一句,这里 2319 行又调用了一次 ep0_reinit,理由很简单,这是针对 Full Speed 而言的,因为别的 speed 的设备的 ep0 的最大包 size 一开始就设置好了,不会改变,而 Full Speed 的则是刚刚从设备描述福利读出来,而且 2310 行的意思就是说读出来这个值和最初我们设想的那个值不同,当然以读出来的为准,所以 ep0.desc.wMaxpacketSize 又有变化,有了变化就得调用

ep0_reinit 让 host controller driver 知道,和前面那个换地址的情况类似,只不过这次是换包的大小,自然也需要把 urb 给清掉,重新洗牌.

八大重量级函数闪亮登场(五)

Go go go, fire in the hole!

八大函数已经看了一半,剩下一半,Linux 十六岁了,十六岁的季节,一半是诗,一半是梦,一首浸透着生命的诗,一个温馨的少年梦.十六岁的天空,一半绚丽,一半深沉,一种缀满五彩缤纷云霞般的绚丽,一种风也洒脱,雨也豪迈的深沉.

让我们继续下一半,我知道你和我一样,也感觉到了一些疲倦,这个时候,正是体现我们作为社会主义有志青年的关键时刻,让我们以黄健翔为榜样,一边高喊张靓颖万岁,一边像男人_婆_一样去战斗!

2514 行至 2536 行,整个这一块代码是专门为了处理 Hub 的,网友"洞房不败"质疑我,这不废话么,现在讲的就是 hub 驱动,不是处理 hub 难道还是为了处理显示器的?错,我的意思是说,2514 行这个 if 语句判断的是,接在当前 Hub 端口的设备不是别的普通设备,恰恰也正是另一个 Hub,即所谓的级联.udev->bus_mA 刚刚在 2493 行那里设置的,即 hub 那边能够提供的每个端口的电流.在当年那个 hub_configure 函数里面我们曾经设置了 hub->mA_per_port,如果它小于等于 100mA,说明设备供电是存在问题的,电力不足.那么这种情况我们首先要判断接入的这个Hub是不是也得靠总线供电,如果是,那就麻烦了.所以这里再次调用usb_get_status(),虽说我们把这个函数列入八大重量级函数之一,但是实际上我们前面已经讲过这个函数了,所以这里不必进入函数内部,只是需要知道 usb_get_status 这么一执行,正常的话,这个子 hub 的状态就被记录在devstat里面了,而2525行的意思就是如果这个设备不能自力更生,那么我们就打印一条错误信息,然后 goto loop_disable,关闭这个端口,结束这次循环.

2529 行至 2533 行又是指示灯相关的代码,当初我们在讲蝴蝶效应的时候就已经说得很清楚了,此刻 schedule_delayed_work(&hub->leds,0)函数这么一执行,就意味着当初注册的 led_work()函数将会立刻被调用.这个函数其实挺简单的,代码虽然不短,但是都是幼稚的代码.考虑到 usb_get_status()我们不用再贴出来了,所以这里就干脆顺便看一下这个幼稚函数吧.

```
208 #define LED_CYCLE_PERIOD        ((2*HZ)/3)
209
210 static void led_work (struct work_struct *work)
211 {
212         struct usb_hub          *hub =
213                 container_of(work, struct usb_hub, leds.work);
214         struct usb_device       *hdev = hub->hdev;
215         unsigned                i;
216         unsigned                changed = 0;
217         int                     cursor = -1;
218
219         if (hdev->state != USB_STATE_CONFIGURED || hub->quiescing)
```

```
220                 return;
221
222         for (i = 0; i < hub->descriptor->bNbrPorts; i++) {
223                 unsigned        selector, mode;
224
225                 /* 30%-50% duty cycle */
226
227                 switch (hub->indicator[i]) {
228                 /* cycle marker */
229                 case INDICATOR_CYCLE:
230                         cursor = i;
231                         selector = HUB_LED_AUTO;
232                         mode = INDICATOR_AUTO;
233                         break;
234                 /* blinking green = sw attention */
235                 case INDICATOR_GREEN_BLINK:
236                         selector = HUB_LED_GREEN;
237                         mode = INDICATOR_GREEN_BLINK_OFF;
238                         break;
239                 case INDICATOR_GREEN_BLINK_OFF:
240                         selector = HUB_LED_OFF;
241                         mode = INDICATOR_GREEN_BLINK;
242                         break;
243                 /* blinking amber = hw attention */
244                 case INDICATOR_AMBER_BLINK:
245                         selector = HUB_LED_AMBER;
246                         mode = INDICATOR_AMBER_BLINK_OFF;
247                         break;
248                 case INDICATOR_AMBER_BLINK_OFF:
249                         selector = HUB_LED_OFF;
250                         mode = INDICATOR_AMBER_BLINK;
251                         break;
252                 /* blink green/amber = reserved */
253                 case INDICATOR_ALT_BLINK:
254                         selector = HUB_LED_GREEN;
255                         mode = INDICATOR_ALT_BLINK_OFF;
256                         break;
257                 case INDICATOR_ALT_BLINK_OFF:
258                         selector = HUB_LED_AMBER;
259                         mode = INDICATOR_ALT_BLINK;
260                         break;
261                 default:
262                         continue;
263                 }
```

```
264                 if (selector != HUB_LED_AUTO)
265                         changed = 1;
266                 set_port_led(hub, i + 1, selector);
267                 hub->indicator[i] = mode;
268         }
269         if (!changed && blinkenlights) {
270                 cursor++;
271                 cursor %= hub->descriptor->bNbrPorts;
272                 set_port_led(hub, cursor + 1, HUB_LED_GREEN);
273                 hub->indicator[cursor] = INDICATOR_CYCLE;
274                 changed++;
275         }
276         if (changed)
277                                         schedule_delayed_work(&hub->leds,
LED_CYCLE_PERIOD);
278 }
```

注意了,刚才我们进入这个函数之前,我们设置了 hub->indicator[port1-1] 为 INDICATOR_AMBER_BLINK,而眼下这个函数从 222 行开始主循环,有多少个端口就循环多少次,即遍历端口.227 行就判断了,对于咱们的这个情形,很显然,selector 被设置为 HUB_LED_AMBER,这个宏的值为 1.而 mode 被设置为 INDICATOR_AMBER_BLINK_OFF.

然后 266 行,set_port_led(),

```
182 /*
183  * USB 2.0 spec Section 11.24.2.7.1.10 and table 11-7
184  * for info about using port indicators
185  */
186 static void set_port_led(
187         struct usb_hub *hub,
188         int port1,
189         int selector
190 )
191 {
192         int status = set_port_feature(hub->hdev, (selector << 8) | port1,
193                         USB_PORT_FEAT_INDICATOR);
194         if (status < 0)
195                 dev_dbg (hub->intfdev,
196                         "port %d indicator %s status %d\n",
197                         port1,
198                         ({ char *s; switch (selector) {
199                         case HUB_LED_AMBER: s = "amber"; break;
200                         case HUB_LED_GREEN: s = "green"; break;
201                         case HUB_LED_OFF: s = "off"; break;
202                         case HUB_LED_AUTO: s = "auto"; break;
```

```
203                             default: s = "??"; break;
204                             }; s; }),
205                             status);
206 }
```

看到调用 set_port_feature 我们就熟悉了,USB_PORT_FEAT_INDICATOR 对应 usb spec 中的 PORT_INDICATOR 这个 feature,

Table 11-25. Port Indicator Selector Codes

Value	Port Indicator Color	Port Indicator Mode
0	Color set automatically, as defined in Table 11-6	Automatic
1	Amber	Manual
2	Green	
3	Off	
4-FFH	Reserved	Reserved

咱们传递进来的是 Amber,即 Selector 为 1.于是指示灯会亮 Amber,即琥珀色.这里我们看到有两个 Mode,一个是 Automatic,一个是 Manual,Automatic 就是灯自动闪,自动变化,而 Manual 基本上就是说我们需要用软件来控制灯的闪烁.我们选择的是后者,所以 265 行我们就设置 changed 为 1.这样,我们将走到 276 行, schedule_delayed_work()再次执行,但这次执行的时候,第二个参数不再是 0,而是 LED_CYCLE_PERIOD,即 0.66HZ.而我们现在的 hub->indicator[port1-1]和刚才进来的时候相反,是 INDICATOR_AMBER_BLINK_OFF,于是你会发现下次咱们进来又会变成 INDICATOR_AMBER_BLINK,如此反复.这就意味着,这个函数接下来将以这个频率被调用,也就是说指示灯将以 0.66HZ 的频率亮了又灭灭了又亮,一闪一闪亮晶晶.不过我需要说的是,注意我们刚才是如何进入到这个函数的,是因为我们遇见了电力不足的情况进来的,所以这并不是什么好事,通常琥珀色的灯亮了话,说明硬件方面有问题.就比如我们这里的电的问题.按 spec 规定,琥珀色亮而不闪,表明是错误环境,琥珀色又亮又闪,表明硬件有问题,只有绿色才是工作状态,如果绿色闪烁,那说明软件有问题.

接下来我们把第六个函数也讲掉.check_highspeed(),看了名字基本就知道是干嘛的了.usb spec 里面规定,一个设备如果能够进行高速传输,那么它就应该在设备描述符里的 bcdUSB 这一项写上 0200H.所以这里的意思就是说如果一个设备可以进行高速传输,但它现在却处于全速传输的状态,并且,highspeed_hubs,这个变量干嘛的?drivers/usb/core/hub.c 中定义的一个静态变量.static unsigned highspeed_hubs,不要说你是第一次见它,当年我们在 hub_probe()里面就和它有过一面之缘.让我们把思绪拉回到 hub_probe 中去,当时我们就判断了如果 hdev->speed 是 USB_SPEED_HIGH,则 highspeed_hubs++,所以现在这里的意思就很明

确了,如果有高速的 hub,而你又能进行高速传输,而你偏偏还要进行全速传输,那么你就是生得贱!这种情况下,调用 check_highspeed.

```
2340 static void
2341 check_highspeed (struct usb_hub *hub, struct usb_device *udev, int
port1)
2342 {
2343         struct usb_qualifier_descriptor *qual;
2344         int                             status;
2345
2346         qual = kmalloc (sizeof *qual, GFP_KERNEL);
2347         if (qual == NULL)
2348                 return;
2349
2350         status = usb_get_descriptor (udev, USB_DT_DEVICE_QUALIFIER,
0,
2351                         qual, sizeof *qual);
2352         if (status == sizeof *qual) {
2353                 dev_info(&udev->dev, "not running at top speed; "
2354                         "connect to a high speed hub\n");
2355                 /* hub LEDs are probably harder to miss than syslog */
2356                 if (hub->has_indicators) {
2357                                                 hub->indicator[port1-1] =
INDICATOR_GREEN_BLINK;
2358                         schedule_delayed_work (&hub->leds, 0);
2359                 }
2360         }
2361         kfree(qual);
2362 }
```

还是那句话,有些事情知道多了才觉得太残酷.我原以为我可以瞒过去,看来是不行了, struct usb_qualifier_descriptor 这个结构体牵出了另外一个概念,Device Qualifier descriptor.首先这个结构体定义于 include/linux/usb/ch9.h 中:

```
348 /* USB_DT_DEVICE_QUALIFIER: Device Qualifier descriptor */
349 struct usb_qualifier_descriptor {
350         __u8  bLength;
351         __u8  bDescriptorType;
352
353         __le16 bcdUSB;
354         __u8  bDeviceClass;
355         __u8  bDeviceSubClass;
356         __u8  bDeviceProtocol;
357         __u8  bMaxPacketSize0;
358         __u8  bNumConfigurations;
```

```
359             __u8  bRESERVED;
360 } __attribute__ ((packed));
```

当我们设计 usb 2.0 的时候我们务必要考虑与过去的 usb 1.1 的兼容,高速设备如果接在一个旧的 hub 上面,总不能说用不了吧?所以,如今的高速设备通常是可以工作于高速也可以不工作于高速,即可以调节,接在高速 hub 上就按高速工作,如果不然,那么就按全速的方式去工作.关键得看环境,放眼古代,三字经中有"昔孟母择邻处子不学断机杼"的佳话,展望今天,也许我长在中国是一个优秀的大学生,但是也许生在日本我就是一个纯粹的变态,环境改变人嘛.那么这一点是如何实现的呢?首先,在高速和全速下有不同设备配置信息的高速设备必须具有一个 device_qualifier 描述符,怎么称呼呢,你可以叫它设备限定符描述符,不过这个叫法过于别扭,所以我们直接用英文,就叫 device qualifier 描述符.它干嘛用的呢?它描述了一个高速设备在进行速度切换时所需改变的信息.比如,一个设备当前工作于全速状态,那么 device qualifier 中就保存着信息记录这个设备工作在高速状态的信息,反之如果一个设备当前工作于高速状态,那么 device qualifier 中就包含着这个设备工作于全速状态的信息.

Ok,这里我们看到,首先定义一个 device qualifier 描述符的指针 qual,然后为其申请内存空间,然后 usb_get_descriptor 去获得这个描述符,这个函数的返回值就是设备返回了多少个 bytes.如果的确是 device qualifier 描述符的大小,那么说明这个设备的确是可以工作在高速状态的,因为全速设备是没有 device qualifier 的,只有具有高速工作能力的设备才具有 device qualifier 描述符,而对于全速设备,在收到这么一个请求之后,返回的只是错误码.所以这里的意思就是说如果你这个设备确实是能够工作在高速的,然而你却偏偏工作于全速,并且我们刚才调用 check_highspeed()之前也看到了,我们已经判断出设备是工作于全速而系统里有高速的 hub,那么至少这说明你这个设备不正常,所以剩下的代码就和刚才那个闪琥珀色的道理一样,只不过这次是闪绿灯,通常闪绿灯的意思是软件问题.

好了,又讲完一个函数.至此我们已经过五关斩六将,八个函数讲完了六个.你也许觉得这些函数很枯燥,觉得读代码很辛苦,不过很不幸,我得告诉你,其实真正最重要的函数是第七个.usb_new_device.这个函数一结束你就可以用 lsusb 命令看到你的设备了.正如<<这个杀手不太冷>>中的对白,人生本就是苦还是只有童年苦?生命就是如此.

八大重量级函数闪亮登场(六)

在调用 usb_new_device 之前,2555 至 2560 这一小段,如果说 hub 已经被撤掉了,那么老规矩,别浪费感情了.否则,把 udev 赋值给 hdev->children 数组中的对应元素,也正是从此以后,这个设备才算是真正挂上了这棵大树.

Ok,如果 status 确实为 0,(注意,2549 刚刚把 status 赋为了 0.)正式调用 usb_new_device.

```
1275 /**
1276  * usb_new_device - perform initial device setup (usbcore-internal)
1277  * @udev: newly addressed device (in ADDRESS state)
1278  *
1279  * This is called with devices which have been enumerated, but not yet
1280  * configured.  The device descriptor is available, but not descriptors
```

```
1281  * for any device configuration.  The caller must have locked either
1282  * the parent hub (if udev is a normal device) or else the
1283  * usb_bus_list_lock (if udev is a root hub).  The parent's pointer to
1284  * udev has already been installed, but udev is not yet visible through
1285  * sysfs or other filesystem code.
1286  *
1287  * It will return if the device is configured properly or not.  Zero if
1288  * the interface was registered with the driver core; else a negative
1289  * errno value.
1290  *
1291  * This call is synchronous, and may not be used in an interrupt context.
1292  *
1293  * Only the hub driver or root-hub registrar should ever call this.
1294  */
1295 int usb_new_device(struct usb_device *udev)
1296 {
1297         int err;
1298
1299         /* Determine quirks */
1300         usb_detect_quirks(udev);
1301
1302         err = usb_get_configuration(udev);
1303         if (err < 0) {
1304                 dev_err(&udev->dev, "can't read configurations, error
%d\n",
1305                         err);
1306                 goto fail;
1307         }
1308
1309         /* read the standard strings and cache them if present */
1310                         udev->product = usb_cache_string(udev,
udev->descriptor.iProduct);
1311         udev->manufacturer = usb_cache_string(udev,
1312                         udev->descriptor.iManufacturer);
1313                             udev->serial = usb_cache_string(udev,
udev->descriptor.iSerialNumber);
1314
1315         /* Tell the world! */
1316                 dev_dbg(&udev->dev, "new device strings: Mfr=%d,
Product=%d, "
1317                         "SerialNumber=%d\n",
1318                         udev->descriptor.iManufacturer,
1319                         udev->descriptor.iProduct,
1320                         udev->descriptor.iSerialNumber);
```

```
1321          show_string(udev, "Product", udev->product);
1322          show_string(udev, "Manufacturer", udev->manufacturer);
1323          show_string(udev, "SerialNumber", udev->serial);
1324
1325 #ifdef  CONFIG_USB_OTG
1326          /*
1327           * OTG-aware devices on OTG-capable root hubs may be able to
use SRP,
1328           * to wake us after we've powered off VBUS; and HNP, switching
roles
1329           * "host" to "peripheral".  The OTG descriptor helps figure this
out.
1330           */
1331          if (!udev->bus->is_b_host
1332                          && udev->config
1333                          && udev->parent == udev->bus->root_hub) {
1334                  struct usb_otg_descriptor       *desc = 0;
1335                  struct usb_bus                  *bus = udev->bus;
1336
1337                  /* descriptor may appear anywhere in config */
1338                                          if  (__usb_get_extra_descriptor
(udev->rawdescriptors[0],
1339
le16_to_cpu(udev->config[0].desc.wTotalLength),
1340                                           USB_DT_OTG, (void **) &desc)
== 0) {
1341                          if (desc->bmAttributes & USB_OTG_HNP) {
1342                                  unsigned                port1 =
udev->portnum;
1343
1344                                  dev_info(&udev->dev,
1345                                          "Dual-Role OTG device on %sHNP
port\n",
1346                                          (port1 == bus->otg_port)
1347                                                  ? "" : "non-");
1348
1349                                  /* enable HNP before suspend, it's simpler
*/
1350                                  if (port1 == bus->otg_port)
1351                                          bus->b_hnp_enable = 1;
1352                                  err = usb_control_msg(udev,
1353                                          usb_sndctrlpipe(udev, 0),
1354                                          USB_REQ_SET_FEATURE, 0,
1355                                          bus->b_hnp_enable
```

```
1356                                                                    ?
USB_DEVICE_B_HNP_ENABLE
1357                                                                    :
USB_DEVICE_A_ALT_HNP_SUPPORT,
1358                                                            0, NULL, 0,
USB_CTRL_SET_TIMEOUT);
1359                                if (err < 0) {
1360                                        /* OTG MESSAGE: report errors
here,
1361                                             * customize to match your
product.
1362                                         */
1363                                        dev_info(&udev->dev,
1364                                                 "can't set HNP mode;
%d\n",
1365                                                 err);
1366                                        bus->b_hnp_enable = 0;
1367                                }
1368                        }
1369                }
1370        }
1371
1372        if (!is_targeted(udev)) {
1373
1374                /* Maybe it can talk to us, though we can't talk to it.
1375                 * (Includes HNP test device.)
1376                 */
1377                if (udev->bus->b_hnp_enable || udev->bus->is_b_host)
{
1378                                        err = __usb_port_suspend(udev,
udev->bus->otg_port);
1379                        if (err < 0)
1380                                dev_dbg(&udev->dev, "HNP fail, %d\n",
err);
1381                }
1382                err = -ENODEV;
1383                goto fail;
1384        }
1385 #endif
1386
1387        /* export the usbdev device-node for libusb */
1388        udev->dev.devt = MKDEV(USB_DEVICE_MAJOR,
1389                               (((udev->bus->busnum-1) * 128) +
(udev->devnum-1)));
```

```
1390
1391        /* Register the device.  The device driver is responsible
1392         * for adding the device files to sysfs and for configuring
1393         * the device.
1394         */
1395        err = device_add(&udev->dev);
1396        if (err) {
1397                dev_err(&udev->dev, "can't device_add, error %d\n",
err);
1398                goto fail;
1399        }
1400
1401        /* Increment the parent's count of unsuspended children */
1402        if (udev->parent)
1403                usb_autoresume_device(udev->parent);
1404
1405 exit:
1406        return err;
1407
1408 fail:
1409        usb_set_device_state(udev, USB_STATE_NOTATTACHED);
1410        goto exit;
1411 }
```

这个函数看似很长,实则不然.幸亏咱们前面作了一个厚颜无耻的假设,即假设不打开支持OTG的代码.在这里1325至1385行就这么被我们华丽丽的飘过了.而剩下的代码就相对来说简单多了,主要就是调用了几个函数.一个一个来看.

usb_detect_quirks().如果不是因为我们讲过了usb-storage,不是因为在usb-storage里面见过那个unusual_devs.h的故事,也许这里我会耐心的给您讲一讲这个关于quirks的故事.实际上这是两个相似的故事,它们共同印证着列夫托尔斯泰在安娜卡列尼娜中的开篇第一句,幸福的家庭都是相似的,不幸的家庭各有各的不幸.好的USB设备都是相似的,大家遵守同样的游戏规则,而不好的USB设备却各有各的毛病,在usb-storage里面我们使用了unusual_devs.h,而在整个usb子系统范围内,我们使用另外两个文件,drivers/usb/core/quirks.c以及include/linux/usb/quirks.h. quirk,金山词霸说,怪癖的意思.说白了就是说白里透红,与众不同.

在include/linux/usb/quirks.h中,我们看到这个文件超级的短,只有两行有意义,其余几行是注释,

```
1 /*
2  * This file holds the definitions of quirks found in USB devices.
3  * Only quirks that affect the whole device, not an interface,
4  * belong here.
5  */
```

```
 6
 7 /* device must not be autosuspended */
 8 #define USB_QUIRK_NO_AUTOSUSPEND          0x00000001
 9
10 /* string descriptors must not be fetched using a 255-byte read */
11 #define USB_QUIRK_STRING_FETCH_255        0x00000002
```

这个文件总共就是这么 11 行,而其中定义了两个 flag,第一个 USB_QUIRK_NO_AUTOSUSPEND 表明这个设备不能自动挂起,执行自动挂起会对设备造成伤害,确切的说是设备会被 crash.而第二个宏,USB_QUIRK_STRING_FETCH_255,是说该设备在获取字符串描述符的时候会 crash.

与此同时,在 drivers/usb/core/quirks.c 中定义了这么一张表,

```
18 /* List of quirky USB devices.  Please keep this list ordered by:
19  *       1) Vendor ID
20  *       2) Product ID
21  *       3) Class ID
22  *
23  * as we want specific devices to be overridden first, and only after that,
any
24  * class specific quirks.
25  *
26  * Right now the logic aborts if it finds a valid device in the table, we might
27  * want to change that in the future if it turns out that a whole class of
28  * devices is broken...
29  */
30 static const struct usb_device_id usb_quirk_list[] = {
31         /* HP 5300/5370C scanner */
32                 {  USB_DEVICE(0x03f0,  0x0701),  .driver_info  =
USB_QUIRK_STRING_FETCH_255 },
33         /* Seiko Epson Corp - Perfection 1670 */
34                 {  USB_DEVICE(0x04b8,  0x011f),  .driver_info  =
USB_QUIRK_NO_AUTOSUSPEND },
35         /* Elsa MicroLink 56k (V.250) */
36                 {  USB_DEVICE(0x05cc,  0x2267),  .driver_info  =
USB_QUIRK_NO_AUTOSUSPEND },
37
38         { }  /* terminating entry must be last */
39 };
```

这张表被称作 usb 黑名单.在 2.6.22.1 的内核中这张表里只记录了 3 个设备,但之所以创建这张表,目的在于将来可以扩充,比如这个夏天,Oliver 同学又往这张表里添加了几个扫描仪,比如明基的 S2W 3300U,精工爱普生的 Perfection 1200,以及另几家公司的一些产品.所以 2.6.23 的内核里将会看到这张表的内容比现在丰富.而从原理上来说,这张表和当初我们的那个

unusual_devs.h 是一样的,usb_detect_quirks()函数就是为了判断一个设备是不是在这张黑名单上,然后如果是的,就判断它具体是属于哪种问题,我们注意到,07 年之后的内核中,struct usb_device 结构体有一个元素 u32 quirks,就是用来做这个检测的,usb_detect_quirks 会为在黑名单中找得到的设备的 struct usb_device 结构体中的 quirks 赋值,然后接下来相关的代码就会判断一个设备的 quirks 中的某一位是否设置了,目前 quirks 里面只有两位可以设置,即 USB_QUIRKS_STRING_FETCH_255 所对应的 0x00000002 和 USB_QUIRK_NO_AUTOSUSPEND 所对应的 0x00000001.而今年 4 月份,Alan 同学又提交了一个 patch,增加了另一个标志位,USB_QUIRK_RESET_RESUME,值为 0x00000004,以表征一个设备不能正确的 resume,而只能通过 reset 才能让它从挂起状态恢复正常.

所以,usb_detect_quirks()所带给我们的就是这么一个故事.它反映的是这样一种现状,即商家只管赚钱,却不管他们家生产出来的产品是否真的合格,只要它的产品差不多就行了,反正是 usb 设备,能用即可,基本功能满足,用户也鉴别不出好坏了.就好比我上个月买的雕牌洗衣粉洗了几次总觉得不好,后来仔细一看,包装袋上写着"周住牌"洗衣粉.

还有一次看路边卖五粮液,觉得便宜就一次性买了好几瓶,回去一喝,感觉完全不对,仔细一看吧,人家瓶子上写的是丑粮液.说了这个我就来气,从上海来北京的时候,火车站买一小说,金庸新著,上了火车我怒了,妈的,作者叫金庸新.

1302 行,usb_get_configuration(),获得配置描述符,我想你如果清楚了如何获得设备描述符,自然就不难知道如何获得配置描述符,知道了配置描述符,自然就不难知道如何获得接口描述符,然后是端点描述符.usb_get_configuration 来自 drivers/usb/core/config.c,我们不打算深入去讲这个函数,如果你有兴趣自己去看,那我做一点点解释,我们知道一个手机可以有多种配置,比如可以摄像,可以接在电脑里当做一个 U 盘,那么这两种情况就属于不同的配置,在手机里面有相应的选择菜单,你选择了哪种它就按哪种配置进行工作,供你选择的这个就叫做配置.很显然,当

你摄像的时候你不可以访问这块 U 盘,当你访问这块 U 盘的时候你不可以摄像,因为你做了选择.第二,既然一个配置代表一种不同的功能,那么很显然,不同的配置可能需要的接口就不一样,我假设你的手机里从硬件上来说一共有 5 个接口,那么可能当你配置成 U 盘的时候它只需要用到某一个接口,当你配置成摄像的时候,它可能只需要用到另外两个接口,可能你还有别的配置,然后你可能就会用到剩下那两个接口,那么当你选择好一种配置之后,你给设备发送请求,请求去获得配置描述符的时候,设备返回给你的就绝不仅仅是一个配置描述符,它还必须返回更多的信息.按 usb spec 的说法就是,设备将返回的是除了配置描述符以外,与这种配置相关的接口描述符,以及与这些接口相关的端点描述符,都会一次性返回给你.也正是因为如此,你才会知道足够的信息,从此以后你就可以为所欲为了.

另外一点我需要提示的是,一个接口可以有多种 setting,即所谓的 alternatesetting,比如在打印机驱动程序里,不同的 setting 可以表明使用不同的通信协议,又比如在声音设备驱动中 setting 可以决定不同的音频格式.那么我作为 usb 设备驱动程序我如何知道这些呢?首先,对于任何一个 interface 来说,usb spec 规定了默认的 setting 是 setting zero,即 0 号设置是默认设置,而如果一个 interface 可以有多种 setting,那么每一个 setting 将对应一个 interface 描述符,换言之,即便你只有一个 interface,但是由于你可能有两种 setting,那么你就有两个 interface 描述符,而它们对应于同一个 interface 编号,或者说我们知道接口描述符里面有一个成员,bInterfaceNumber 和一个 bAlternateSetting,就是对于这种情况,两个 interface 描述符将具有相同的 bInterfaceNumber,而不相同的是 bAlternateSetting,另一方面,因为不同的 setting 完全有可能导致需要不同的端点,所以也将有不同的端点描述符.

而总的来说,在我们的 usb 设备驱动程序可以正常工作之前,我们需要知道的信息是,接口描述符,Setting,以及端点描述符,从软件的角度来说,我们记得当初我们在 usb-storage 中的 probe 函数,我们传递给它的一个重要参数就是 struct usb_interface 指针,同时所有关于端点的信息也必须在调用 storage_probe 之前知道,而这一切的一切,都在设备里,我们所需要做的就是发送请求,然后设备就把相关信息返回给我们,然后我们就记录下来,填充好我们自己的数据结构,而这些数据结构,对所有的 usb 设备都是一样的,因为这些都是 usb spec 里面规定的,也正是因为如此,写代码的兄弟们才把这部分工作交给 usb core 来完成,而不是纷纷下放给每一个设备单独去执行,因为那样就太浪费了,大家都得干一些重复的工作,显然是没有必要的.

Ok,关于 usb_get_configuration()函数我们就说这么多,我们传递的是 struct usb_device 结构体指针,即我们这个故事中的 udev,从此以后你就会发现 udev 中的各个成员就有值了,这些值从哪来的?正是从设备里面来.

回到 usb_new_device 中来,1310 行 1323 行,还记得我们说过那个字符串描述符吧,这里就是去获得字符串描述符,并且保存下来,知道为什么你用 lsusb 命令可以看到诸如下面的内容了吧,

```
localhost:/usr/src/linux-2.6.22.1/drivers/usb/core # lsusb
Bus 001 Device 001: ID 0000:0000
Bus 002 Device 003: ID 0624:0294 Avocent Corp.
Bus 002 Device 001: ID 0000:0000
Bus 003 Device 001: ID 0000:0000
Bus 004 Device 003: ID 04b4:6560 Cypress Semiconductor Corp. CY7C65640 USB-2.0 "TetraHub"
Bus 004 Device 001: ID 0000:0000
```

其中那些字符串,就是这里保存起来的.试想如果不保存起来,那么每次你执行 lsusb,都要去向设备发送一次请求,那设备还不被你烦死?usb_cache_string()就是干这个的,它来自 drivers/usb/core/message.c, 从此 udev->product,udev->manufacturer,udev->serial 里面就有值了.而下面那几行就是打印出来,show_string 其实就是变相的 printk 语句.1315 那句注释尤其搞笑,说什么"Tell the world!"告诉世界自己是一个什么样的设备,我有点怀疑写代码的兄弟是不是活得太压抑了一点,要知道在我们国家,跟一个人说隐私那叫倾诉,跟一群人说隐私那叫变态,跟全国人民说隐私那就叫<<艺术人生>>.

接下来,我们已经说过了,OTG 的代码我们只能飘过,然后就到了 1388 行,这里就是传统理论中的那两个主设备号和次设备号了.记得去年在恒隆广场面试赛门铁克的时候就被问到 Linux 中的设备号是怎么回事,分为主设备号和次设备号.按传统理论来说,主设备号表明了一类设备,一般对应着确定的驱动程序,而次设备号通常是因为一个驱动程序要支持多个设备而为了让驱动程序区分它们而设置的.比如如下,偶的硬盘:

```
localhost:/usr/src/linux-2.6.22.1/drivers/usb/core # ls -l /dev/sd*
brw-r----- 1 root disk 8,  0 Aug  6 18:19 /dev/sda
brw-r----- 1 root disk 8,  1 Aug  6 18:19 /dev/sda1
brw-r----- 1 root disk 8,  2 Aug  6 18:19 /dev/sda2
brw-r----- 1 root disk 8,  3 Aug  6 18:19 /dev/sda3
brw-r----- 1 root disk 8,  4 Aug  6 18:19 /dev/sda4
brw-r----- 1 root disk 8, 16 Aug  6 18:19 /dev/sdb
brw-r----- 1 root disk 8, 32 Aug  6 18:19 /dev/sdc
brw-r----- 1 root disk 8, 48 Aug  6 18:19 /dev/sdd
brw-r----- 1 root disk 8, 64 Aug  6 18:19 /dev/sde
```

scsi 硬盘主设备号都是 8,而不同的盘或者不同的分区都有不同的次设备号.次设备号具体为多少并不重要,不过最大不能超过 255.usb 子系统里使用以下公式安排次设备号的,即 minor=((dev->bus->busnum-1)*128)+(dev->devnum-1);而 USB_DEVICE_MAJOR 被定义为 189,我们看:

```
localhost:/usr/src/linux-2.6.22.1/drivers/usb/core # cat /proc/devices
Character devices:
  1 mem
  2 pty
  3 ttyp
  4 /dev/vc/0
  4 tty
  4 ttyS
  5 /dev/tty
  5 /dev/console
  5 /dev/ptmx
  7 vcs
 10 misc
 13 input
```

```
 21 sg
 29 fb
128 ptm
136 pts
162 raw
180 usb
189 usb_device
254 megaraid_sas_ioctl
```

189 被称为 usb_device,这两行代码是用于与 usbfs 文件系统相交户的.dev_t 记录下了设备的主设备号和次设备号.即,dev_t 包含两部分,主设备号部分和次设备号部分.高 12 位表征主设备号,低 20 位表示次设备号.

1395 行,device_add(),Linux 2.6 设备模型中最基础的函数之一,这个函数非常了不起.要深入追踪这个函数,足以写一篇专题文章了.这个函数来自 drivers/base/core.c 中,是设备模型那边提供的函数,从作用上来说,这个函数这么一执行,系统里就真正有了咱们这个设备,/sysfs 下面也能看到了,而且将会去遍历注册到 usb 总线上的所有的驱动程序,如果找到合适的,就去调用该驱动的 probe 函数,对于 U 盘来说,最终将调用 storage_probe()函数,对于 hub 来说,最终将调用 hub_probe()函数,而传递给它们的参数,正是我们此前获得的 struct usb_interface 指针和一个 struct usb_device_id 指针.后者我们在 usb-storage 里面已经非常熟悉了,它正是我们在 usb 总线上寻找驱动程序的依据,换句话说,每个驱动程序都会 usb-storage 那样,把自己支持的设备定义在一张表里,表中的每一项就是一个 struct usb_device_id,然后当我们获得了一个具体设备,我们就把该设备的实际的信息与这张表去比较,如果找到匹配的了,就认为该驱动支持该设备,从而最终会调用该驱动的 probe()函数.而从此,这个设备就被传递到了设备驱动.而 hub 驱动也完成了它最重要的一项工作.

再次回到 usb_new_device(),1402 行,如果该设备不是 Root Hub,则调用 usb_autoresume_device().这个函数来自 drivers/usb/core/driver.c,也是 usb core 提供的,是电源管理方面的函数,如果设备这时候处于 suspended 状态,那么这个函数将把它唤醒,因为我们已经要开始用它了,怎么会任凭它睡眠呢.

最后 1406 行,函数终于返回了.正确的话返回值为 0.耶!

返回之后首先判断返回值,如果不为 0 说明出错了,那么就把 hdev->children 相应的那位设置为空.要知道我们在调用 usb_new_device 之前可是把它设置成了 udev,怎奈它不思进取,只好放弃把它拉入 usb 组织的想法.

八大函数只剩下最后一个 hub_power_remaining(),这个函数相对来说比较小巧玲珑.这是与电源管理相关的.虽然说刚接入的设备可能已经被设备驱动认领了,但是作为 hub 驱动,自身的工作还是要处理干净的.

```
2364 static unsigned
2365 hub_power_remaining (struct usb_hub *hub)
2366 {
2367         struct usb_device *hdev = hub->hdev;
```

```
2368        int remaining;
2369        int port1;
2370
2371        if (!hub->limited_power)
2372                return 0;
2373
2374                                remaining = hdev->bus_mA -
hub->descriptor->bHubContrCurrent;
2375        for (port1 = 1; port1 <= hdev->maxchild; ++port1) {
2376                struct usb_device       *udev = hdev->children[port1 -
1];
2377                int                     delta;
2378
2379                if (!udev)
2380                        continue;
2381
2382                /* Unconfigured devices may not use more than 100mA,
2383                 * or 8mA for OTG ports */
2384                if (udev->actconfig)
2385                        delta = udev->actconfig->desc.bMaxPower * 2;
2386                else if (port1 != udev->bus->otg_port || hdev->parent)
2387                        delta = 100;
2388                else
2389                        delta = 8;
2390                if (delta > hub->mA_per_port)
2391                         dev_warn(&udev->dev, "%dmA is over %umA
budget "
2392                                          "for port %d!\n",
2393                                                delta, hub->mA_per_port,
port1);
2394                remaining -= delta;
2395        }
2396        if (remaining < 0) {
2397                dev_warn(hub->intfdev, "%dmA over power budget!\n",
2398                        - remaining);
2399                remaining = 0;
2400        }
2401        return remaining;
2402 }
```

limited_power 是咱们当初在 hub_configure()中设置的.设置了它说明能源是有限的,希望大家珍惜.所以如果这个变量不为 0,我们就要对电源精打细算.要计算出现在还能提供多大电流.即把当前 bus_mA 减去 hub 自己需要的电流以及现在连在 hub 端口上的设备所消耗的电流,求出剩余值来,然后打印出来,告诉世界我们还有多少电流 budget.bHubContrCurrent 咱们前面在

讲 hub_configure 的时候就已经说过了,是 Hub 控制器本身最大的电流需求,单位是 mA,来自 hub 描述符.

2375 至 2395 行这段循环,就是上面说的这个思想的具体实现.遍历每个端口进行循环.每个设备的配置描述符中 bMaxPower 就是该设备从 usb 总线上消耗的最大的电流.其单位是 2mA,所以这里要乘以 2.没有配置过的设备是不可能获得超过 100mA 电流的.

如果 delta 比 hub 为每个端口提供的平均电流要大,那么至少要警告一下.

然后循环完了,remaining 就是如其字面意义一样,还剩下多少电流可供新的设备再接进来.注意到我们从软件的角度来说,是不会强行对设备采取什么措施,我们最多是打印出调试信息,警告警告,而设备如果真的遇到了供电问题,它自然会出现异常,它也许不能工作,这些当然由具体的设备驱动程序去关注,从 Hub 这一层来说,没有必要去干涉人家内部的事情,做到这一步已经是仁至义尽了.

终于我们讲完了这八个函数,可以说到这里为止,我们已经知道了 hub 驱动是在端口连接有变化的时候如何工作的,并且更重要的是我们知道了 hub 驱动是如何为子设备驱动服务的.回到 hub_port_connect_change 之后,一切正常的话我们将会从 2579 行返回.

剩下的一些行就是错误处理代码.我们就不必再看了.因此我们将返回到 hub_events()中来.这个函数还剩下几行,我们下节再看.不过你千万别以为看到这里你就完全明白 hub 驱动程序了,因为那样无异于你把自己家里的铁锅背在背上就以为自己是忍者神鬼.

别着急,慢慢来,其实懂与不懂,就像黑夜和白天,相隔一瞬间.

是月亮惹的祸还是 spec 的错

来北京九个月了,和同学去五道口的光合作用酒吧玩了一夜,其间和两个乌拉圭的女留学生搭讪,其间学会了吸水烟,其间和我一个北邮的同学聊到生活的艰难,他说,人们为了求生而来到大都市,但是依我看,他们是为了求死而来.出来之后,独自叹息,工作两年,没有省下一分钱,生活压力大得不得了,买房买车这些事看起来和我简直有一万光年的距离.而你不得不承认,这种距离,就像稀盐酸,腐蚀的是我的人生观,腐蚀的是我对生活的信心.可恶的是那些专家还说,买不起房子可以租房子,那么我想问,我娶不起媳妇,可不可以也租一个来?怎么没见哪个专家提出愿意出租他的女儿来给我做一个月媳妇呢?难怪大家都说,杀掉一批专家有利于更好的建设和谐社会.

想到这些烦心事心里就郁闷,在站台等 375 路公交车回家,一边等一边看了看代码,九月的天气,下起大雨,淋湿我的思绪,可笑的是,却没能遮住我发现 Bug 的双眼.雨后的花瓣,散落一地,把它做成书签,藏在日记,记录下这个 Linux 内核中的 Bug.

让我们用代码来说话.走在 hub_events()的小路上,2777 行,把 hub->event_bits 给清掉,然后读一次 hub 的状态,HUB_STATUS_LOCAL_POWER 我们以前在 hub_configure 中见过,用来标志这个 hub 是有专门的外接电源的还是从 usb 总线上获取电源,而 C_HUB_LOACL_POWER用来标志这一位有变化,这种情况下,先把C_HUB_LOCAL_POWER

清掉,同时判断,如果是原来没有电源现在有了电源,那么可以取消 limited_power 了,把它设置为 0,如果是反之,原来是有电源的,而现在没了,那么没什么说的,把 limited_power 设置为 1.

这种解释,用孙俪的新歌<<爱如空气>>中的那句歌词来说,看似很美丽.但是,也正如孙俪唱的那样,看似很美丽,却无法触及,因为真理和错误有时候只有一步之遥.我们来对照一下 usb spec.如下图所示:

Table 11-19. Hub Status Field, *wHubStatus*

Bit	Description
0	**Local Power Source:** This is the source of the local power supply. This field indicates whether hub power (for other than the SIE) is being provided by an external source or from the USB. This field allows the USB System Software to determine the amount of power available from a hub to downstream devices. 0 = Local power supply good 1 = Local power supply lost (inactive)
1	**Over-current:** If the hub supports over-current reporting on a hub basis, this field indicates that the sum of all the ports' current has exceeded the specified maximum and all ports have been placed in the Powered-off state. If the hub reports over-current on a per-port basis or has no over-current detection capabilities, this field is always zero. For more details on over-current protection, see Section 7.2.1.2.1. 0 = No over-current condition currently exists. 1 = A hub over-current condition exists.
2-15	**Reserved** These bits return 0 when read.

看出问题来了吗?代码和我的解释刚好相反.代码的意思是 HUB_STATUS_LOCAL_POWER 为 1,就设置 limited_power 为 0,反之则设置 limited_power 为 1.你说你应该相信代码还是应该相信我?网友"硬把红杏拽出墙"提醒大家,<<偷天陷阱>>里有一句话,不要相信漂亮女人,尤其是不穿衣服的裸体女人.不过还好,我四处张望了以后发现,这里没有女人.但我必须郑重声明,我要像新闻联播里声明北京电视台的纸包子报道是假新闻一样郑重声明:我从不说假话.这句除外.

让我们多说两句,众所周知,一个 hub 可以用两种供电方式,一种是自带电源,即 hub 上面自己有根电源线,插到插座上,就 ok 了.另一种是没有自带电源,由总线来供电.具体这个 hub 是使用的哪种方式供电,就是从这个状态位里面可以读出来,即上面这个 Local power source,Spec 的意思是,Local Power Source 如果为 0,表明本地供电正常,即说明是自带电源,如果 Local Power Source 为 1,则说明本地供电挂了,或者根本就没有本地供电.而我们 Hub 设备驱动中之所以引入一个叫做 limited_power 的变量,就是为了记录这一现象,如果你这个 Hub 有自己的电源,那么你就可以为所欲为,可以无法无天,因为你有后台撑腰,你有 Power,但是如果你是一个普通老百姓,你要依赖政府,你要依靠总线来给你解决电源问题,那么驱动程序就要记录下来,因为总的资源是有限的,你占用了这么多,分给别人的就少了这么多,所以这种情况下设置 limited_power 为 1,也算是记下这么一件事.

所以说,正确的赋值应该是 HUB_STATUS_LOCAL_POWER 为 1,设置 limited_power 为 1,HUB_STATUS_LOCAL_POWER 为 0,设置 limited_power 为 0.所以这就是传说中的 Bug.有趣的是这个 Bug 自 2005 年平安夜由三剑客之一的 Alan 提出,并于 2006 年一月由 Greg 正式引入 Linux 内核,在连续几个稳定版的内核中隐藏了近两年,终于在这个早上,就在我在酒吧一夜未曾合眼然后出来坐上 375 路公共汽车之后发现了.你说我容易么?

不过我想问,这究竟是月亮惹的祸还是 spec 的错?其实 spec 算不上错,但是 spec 中这种定义是不合理的,它这一位就不该叫做 Local Power Source,因为这样一叫别人就会误以为这位为 1 的时候表示有电源,为 0 表示没有电源,所以才导致了这个 Bug,更合理的叫法应该是叫 Local Power Lost. 现在好了,既然被我发现了,那么 2.6.23 的正式版内核里不会有这个 Bug 了.不过你别以为这样的 Bug 很幼稚,Alan 同志说过,对一个人显然的事情未必会对另一个人显然,很多 Bug 都存在这样的情况,当你事后去看的话,你会觉得它很不可思议,太低级了,但是有时候低级的错误你却未必能够在短时间内发现.

Ok,我们继续看代码,2791 行,对于有过流的改变也是同样的处理,因为过流可能导致端口关闭,所以重新给它上电.hub_power_on().其实这个函数我们以前见过,只是当时出于情节考虑,先飘过了,现在我们对 hub 有了这么多认识了之后再来看这个函数就好比一个大学生去看小学数学题一样简单.

```
475 static void hub_power_on(struct usb_hub *hub)
476 {
477         int port1;
478         unsigned pgood_delay = hub->descriptor->bPwrOn2PwrGood *
2;
479         u16 wHubCharacteristics =
480
le16_to_cpu(hub->descriptor->wHubCharacteristics);
481
482         /* Enable power on each port.  Some hubs have reserved values
483          * of LPSM (> 2) in their descriptors, even though they are
484            * USB 2.0 hubs.  Some hubs do not implement port-power
switching
485          * but only emulate it.  In all cases, the ports won't work
486          * unless we send these messages to the hub.
487          */
488         if ((wHubCharacteristics & HUB_CHAR_LPSM) < 2)
489                 dev_dbg(hub->intfdev, "enabling power on all ports\n");
490         else
491                 dev_dbg(hub->intfdev, "trying to enable port power on "
492                                 "non-switchable hub\n");
493         for (port1 = 1; port1 <= hub->descriptor->bNbrPorts; port1++)
494                                 set_port_feature(hub->hdev,  port1,
USB_PORT_FEAT_POWER);
495
496         /* Wait at least 100 msec for power to become stable */
497         msleep(max(pgood_delay, (unsigned) 100));
498 }
```

关键的代码就是一行,494 行,set_port_feature,USB_PORT_FEAT_POWER,这一位如果为 0,表示该端口处于 Powered-off 状态.同样,任何事情引发该端口进入 Powered-off 状态的话都会使得这一位为 0.而 set_port_feature 就会把这一位设置为 1,这叫做使得端口 power on.关

于对 HUB_CHAR_LPSM 的判断,本来是没有必要的,这里又一次出了怪事了,关于这件事情的解释完全在 482 到 487 这段注释里面,我就不用中文重复了,总之最终的做法就是对每个端口都执行一次 set_port_feature.然后最后 497 行,睡眠,经验值是 100ms,而 Hub 描述符里有一位 bPwrOn2PwrGood,全称就是 b-Power On to Power Good,即从打开电源到电源稳定的时间,显然我们应该在电源稳定了之后再去访问每一个端口所以这里睡眠时间就取这两个中的较大的那一个.

2799 行,设置 hub->activating 为 0,也就是说以上这一段被称为 activating,而这个变量本身,就是一个标志而已.别忘了我们是从 hub_actiavte 调用 kick_khubd()从而进入到这个 hub_events()的.而在 hub_activate()中我们设置了 hub->activating 为 1.而那个函数也是唯一一个设置这个变量为 1 的地方.

2803 行,如果是 Root Hub,并且 hub->busy_bits[0]为 0,hub->busy_bits 只有在一个端口为 reset 或者 resume 的时候才会被设置成 1.我们暂时先不管.对于这种情况,即既是 Root Hub,又没有端口处于 reset/resume 状态,调用 usb_enable_root_hub_irq()函数,这个函数来自 drivers/usb/core/hcd.c,是 host controller driver 相关的,有些 host controller 的驱动程序提供了一个叫做 hub_irq_enable 的函数,这里就会去调用它,不过目前主流的 ehci/uhci/ohci 都没有提供这个函数,所以你可以认为这个函数什么也没干.这个函数的作用正如它的名字解释的那样,开启端口中断.关于这个函数,涉及到一些比较专业性的东西,有两种中断的方式,边缘触发和电平触发.只有电平触发的中断才需要这个函数,边缘触发的中断不需要这个函数.算了,不跟你多说了,想你也不感兴趣.

2808 行,判断,如果 hub 的 event_list 没有东西了,那么就调用 usb_autopm_enable(),调用了这个函数这个 hub 就可以被挂起了,强调一下,可以做某事不等于马上就做某事了,真的被挂起是有条件的,首先它的子设备必须先挂起了,而且确实一段时间内没有总线活动了,才会被挂起.

最后,释放 hdev 的锁,减少 intf 的引用计数,至此,hub_events()这个永垂不朽的函数就算结束了!对于大部分人来说,特别是对于那些不求甚解的同志们来说,你们需要学习的 hub 驱动就算学完了.因为你已经完全明白了 hub 驱动在设备插入之后会做一些什么事情,会如何为设备服务,并最终把控制权交给设备驱动.而 hub_thread()/hub_events()将永远这么循环下去.

不过我的故事可没有结束,最起码还有一个重要的函数没有讲.hub_irq.之前我们这里的故事都是基于一个事实就是我们主动去读了 hub 端口的状态,而以后正常工作的 hub 驱动是不会莫名其妙就去读的,只有发生了中断才会去读.而这个中断的服务函数就是 hub_irq,也即是说,凡是真正的有端口变化事件发生,hub_irq 就会被调用,而 hub_irq()最终会调用 kick_khubd(),触发 hub 的 event_list,于是再次调用 hub_events().

下节我们会讲 hub_irq().然后还会讲剩下的一些函数,这其中最关键的就是电源管理部分的代码.因为电源管理是眼下 Linux 内核中最热门的话题之一.在 usb 中实现电源也是最近两年才开始的,并且经常有 Bug,所以,我们有必要,我们很有必要,我们非常有必要来看看这部分代码.网友"有钱人终成眷属"问我,有 Bug 为什么还要看啊?我想,男人,最起码应该有点责任心,我们对国家,对集体,对家庭,都应该有点责任心,要有使命感,不要见困难就让,见容易就上.这方面我做得就不错,现在台海局势不太好,我就想好了,万一哪天攻打台湾,我就会勇敢的上前线,我听说,台湾那边最危险的人物叫林志玲,所以,请组织把林志玲交给我,我保证完成任务!

所谓的热插拔

你问我这世界,最远的地方在哪里,我将答案抛向蓝天之外落在你心底,你问我这世界,最后的真爱在哪里,我把线索指向大海之外直达我怀里.你问我 hub_irq()这个函数,最终是被谁调用,我却只能说我既没有答案也没有线索.当然,你要是问我芙蓉姐姐还能红多久,我倒是可以很爽快的告诉你,你知道永远有多远吗?

我们曾经在 hub_configure 中讲过中断传输,当时调用了 usb_fill_int_urb()函数,并且把 hub_irq 作为一个参数传递了进去,最终把 urb->complete 赋值为 hub_irq.然后,主机控制器会定期询问 hub,每当 hub 端口上有一个设备插入或者拔除时,它就会向主机控制器打小报告.(怎么哪都有人打小报告啊,当初在 Intel 不知道哪位哥们缺德,跟老板说我做事情只关注结果不管过程,只要问题解决了就好,根本不管问题是怎么被解决的.害得我在 Intel 差点没过试用期.)具体来说,从硬件的角度看,就是 hub 会向 host controller 返回一些信息,或者说 data,这个 Data 被称作"Hub and Port Status Change Bitmap",而从软件角度来看,host controller 的驱动程序接下来会在处理好这个过程的 urb 之后,调用该 urb 的 complete 函数,对于 hub 来说,这个函数就是 hub_irq().

```
337 /* completion function, fires on port status changes and various faults */
338 static void hub_irq(struct urb *urb)
339 {
340         struct usb_hub *hub = urb->context;
341         int status;
342         int i;
343         unsigned long bits;
344
345         switch (urb->status) {
346         case -ENOENT:             /* synchronous unlink */
347         case -ECONNRESET:          /* async unlink */
348         case -ESHUTDOWN:           /* hardware going away */
349                 return;
350
351         default:                  /* presumably an error */
352                 /* Cause a hub reset after 10 consecutive errors */
353                         dev_dbg (hub->intfdev, "transfer --> %d\n",
urb->status);
354                 if ((++hub->nerrors < 10) || hub->error)
355                         goto resubmit;
356                 hub->error = urb->status;
357                 /* FALL THROUGH */
358
359         /* let khubd handle things */
360         case 0:                   /* we got data:   port status changed */
361                 bits = 0;
362                 for (i = 0; i < urb->actual_length; ++i)
```

```
363                         bits |= ((unsigned long) ((*hub->buffer)[i]))
364                                         << (i*8);
365                 hub->event_bits[0] = bits;
366                 break;
367         }
368
369         hub->nerrors = 0;
370
371         /* Something happened, let khubd figure it out */
372         kick_khubd(hub);
373
374 resubmit:
375         if (hub->quiescing)
376                 return;
377
378         if ((status = usb_submit_urb (hub->urb, GFP_ATOMIC)) != 0
379                         && status != -ENODEV && status != -EPERM)
380                 dev_err (hub->intfdev, "resubmit --> %d\n", status);
381 }
```

你问这个参数 urb 是哪个 urb?告诉你,中断传输就是只有一个 urb,不是说像 bulk 传输那样每次开启一次传输都要有申请一个 urb,提交 urb,对于中断传输,一个 urb 就可以了,反复利用,所以我们只有一次调用 usb_fill_int_urb()函数.这正体现了中断交互的周期性.

340 行,当初我们填充 urb 的时候,urb->context 就是赋的 hub,所以现在这句话就可以获得我们的那个 hub.

345 行开始判断 urb 的状态,前三种都是出错了,直接返回.

351 的 default 和 case 0.这段代码是我认为最有技术含量的一段代码.我不知道我是该恨谭浩强该是该恨我自己,我记得我在谭浩强的书上看到的 default 总是在各个 case 之后,结果我以为这里 default 不管 case 等于 0 与否都会执行,结果半天没看懂,后来我明白了,其实当 urb->status 为 0 的时候,default 那一段是不会执行的.

所以这段代码就很好理解了.一开始 hub->error 为 0,hub->nerrors 也为 0,所以 default 这一段很明显,goto resubmit,即,我们允许年轻人犯错误,并且可以允许犯十次,错了没有关系,只要不在同一条阴沟里翻船翻十次就可以了,我相信这对大多数人来说都足够了,唯一例外的大概是中国男足了.每次阴沟里翻船,爬起来一看,咦,怎么还是上次那条阴沟啊?resubmit 那一段就是重新调用了一次 usb_submit_urb()而已.当然,还判断了 hub->quisecing.这个变量初始值为 1,但是我们前面在 hub_activate 里把它设置为了 0,有一个函数会把它设置为 1,这个函数就是 hub_quiesce(),而调用后者的只有两个函数,一个是 hub_suspend,一个是 hub_pre_reset().于是,这里的意思就很明确了,如果 hub 被挂起了,或者要被 reset 了,那么就不用重新提交 urb 了,hub_irq()函数直接返回吧.

再看 case 0,urb->status 为 0,说明这个 urb 被顺利的处理了,即 host controller 获得了他想要的数据,即那个"Hub and Port Status Change Bitmap",因为我们当初调用 usb_fill_int_urb 的时候,把*hub->buffer 传递给了 urb->transfer_buffer,所以这个数据现在就在 hub->buffer 中,我们来看这个 bitmap 是什么样子,usb spec 中给出了这样一幅图:

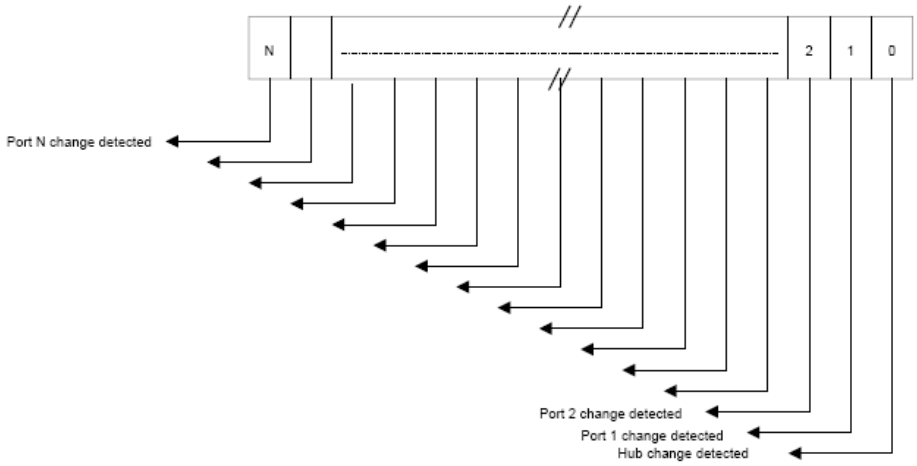


Figure 11-22. Hub and Port Status Change Bitmap

首先这幅图为我们解答了一个很大的疑惑.当时在 hub_events()中您大概还有一些地方不是很清楚.现在我们可以来弄清楚它们了.

我们回过头来看,struct usb_hub 中,unsigned long event_bits[1],首先这是一个数组,其次这个数组只有一个元素,而这一个元素恰恰就是对应这里的这个 Bitmap,即所谓的位图,每一位都有其作用.一个 unsigned long 至少 4 个字节,即 32 个 bits.所以够用了,而我们看到这张图里,bit0 和其它的 bit 是不一样的,bit 0 表示 Hub 有变化,而其它 bit 则具体表示某一个端口有没有变化,即 bit 1 表示端口 1 有变化,bit 2 表示端口 2 有变化,如果一个端口没有变化,对应的那一位就是 0.

所以我们可以回到 hub_events()函数中来,看看当时我们是如何判断 hub->event_bits 的,当时我们有这么一小段,

```
2685                /* deal with port status changes */
2686                for (i = 1; i <= hub->descriptor->bNbrPorts; i++) {
2687                        if (test_bit(i, hub->busy_bits))
2688                                continue;
2689                        connect_change = test_bit(i, hub->change_bits);
2690                        if (!test_and_clear_bit(i, hub->event_bits) &&
2691                                                        !connect_change
&& !hub->activating)
2692                                continue;
```

看到了吗?循环指数 i 从 1 开始,有多少端口就循环多少次,而对 event_bits 的测试,即 2690 判断的是 bitmap 中 bit 1,bit 2,…,bit N,而不需要判断 bit 0.

反过来,如果具体每个端口没有变化,而变化的是 hub 的整体,比如,Local Power 有变化,比如 Overcurrent 有变化,我们则需要判断的是 bit 0.即当时我们在 hub_events()中看到的下面这段代码.

```
2776                /* deal with hub status changes */
2777                if (test_and_clear_bit(0, hub->event_bits) == 0)
2778                        ;       /* do nothing */
2779                else if (hub_hub_status(hub, &hubstatus, &hubchange) <
0)
2780                        dev_err (hub_dev, "get_hub_status failed\n");
2781                else {
2782                        if (hubchange & HUB_CHANGE_LOCAL_POWER) {
2783                                dev_dbg (hub_dev, "power change\n");
2784                                        clear_hub_feature(hdev,
C_HUB_LOCAL_POWER);
2785                                        if (hubstatus &
HUB_STATUS_LOCAL_POWER)
2786                                /* FIXME: Is this always true? */
2787                                hub->limited_power = 0;
2788                                else
2789                                hub->limited_power = 1;
2790                        }
2791                        if (hubchange & HUB_CHANGE_OVERCURRENT) {
2792                                dev_dbg (hub_dev, "overcurrent
change\n");
2793                                msleep(500);    /* Cool down */
2794                                        clear_hub_feature(hdev,
C_HUB_OVER_CURRENT);
2795                                hub_power_on(hub);
2796                        }
2797                }
```

而这样我们就很清楚 hub_events()的整体思路了,判断每个端口是否有变化,如果有变化就去处理它,没有变化也没有关系,接下来判断是否 hub 整体上有变化,如果有有变化,那么也去处理它.满足了个人利益,满足了集体利益,这不正体现了我们社会主义制度的优越性么?

关于这个 actual_length,其实你不傻的话这个根本就不用我啰嗦了.因为每一个 hub 的 port 是不一样的,所以这张 bitmap 的长度就不一样,比如说你是 16 个 port,那么这个 bitmap 最多就只要 16+1 个 bit 就足够了.而 actual_length 就是 3,即 3 个 bytes.因为 3 个 bytes 等于 24 个 bits,足以容纳 16+1 个 bits 了.而 struct usb_hub 中,buffer 是这样一个成员,char (*buffer)[8],所以 3 个 bytes 就意味着这个 buffer 的前三个元素里承载着我们的希望.这样我们就不难理解这里这个 hub->event_bits 是如何变成为这张 bitmap 的了.

369 行,把 nerrors 清零吧,过去的事情就让它过去吧,让我们忘记过去,展望未来.其实这道理不用我说,小刚那首<<忘记>>就说的很清楚了:有太多往事就别喝下太少酒精,太珍惜生命就别随

便掏心,舍不得看破就别张开眼睛,想开心就要舍得伤心;有太多行李就别单独旅行,不能够离开就不要接近,舍不得结束就别开始一段感情,想忘记就要一切归零.谢谢林夕,用这么好的歌词为我们诠释了 Linux 内核代码.

最关键的当然还是 372 行,再次调用 kick_khubd()函数,于是会再一次触发 hub_events().

而 hub_irq()函数也就到这里了.这个函数不长,可是很重要,其中最重要的正是最后这句 kick_khubd().而这也就是所谓的热插拔的实现路径,要知道我们上次分析 kick_khubd 的时候是在 hub 初始化的时候,即那次针对的情况是设备一开始就插在了 hub 上,而这里再次调用 kick_khubd 才是真正的在使用过程中,突然间 hub 有了变化的情况,应该说这个后者才是真正有技术含量的冬冬.

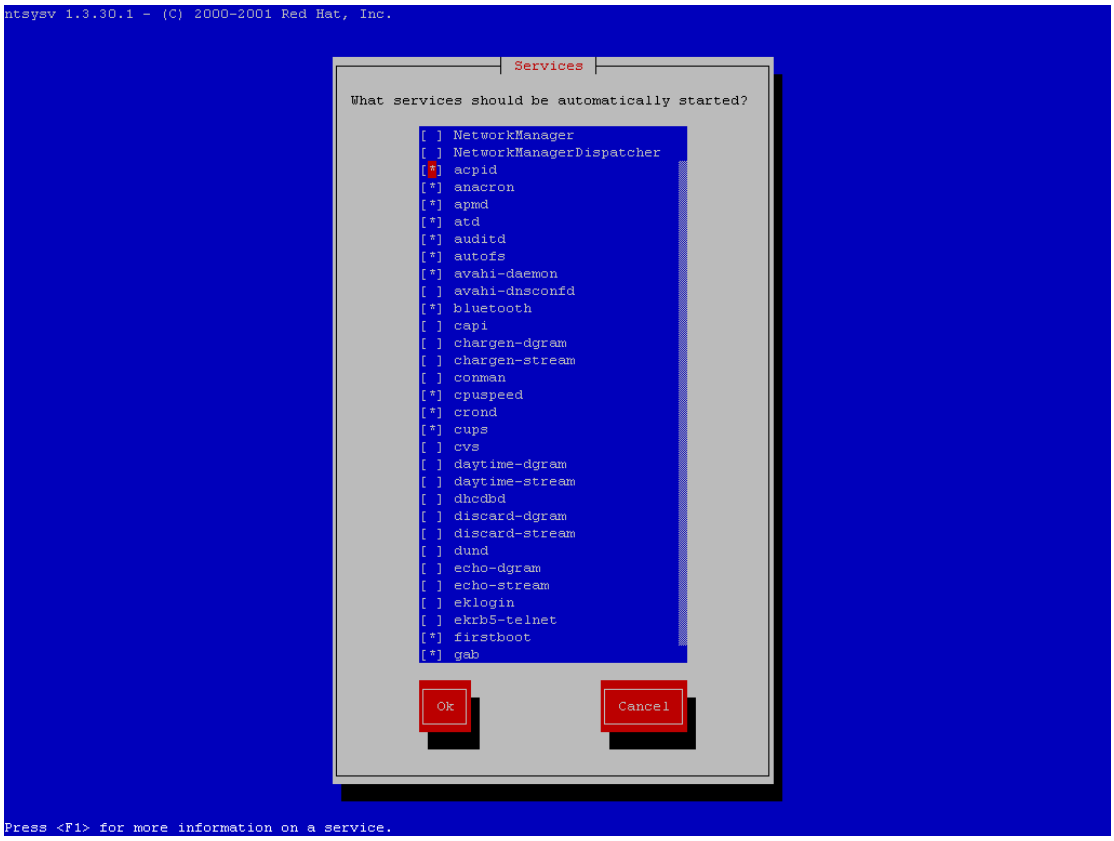
不说代码说理论

当女作家们越来越多的使用下半身来告诉我什么是文学的时候,当模特们越来越多的使用裸体来告诉我什么是人体艺术的时候,我开始对这个社会困惑了,当行为艺术家们越来越多的使用垃圾堆砌来告诉我什么是波谱的时候,当地下音乐者们越来越多的使用烦躁不安的敲打来告诉我什么是原创的时候,我开始对这个时代迷茫了,当我们系的教授们越来越多的使用旧得不能再旧的教材来告诉我什么是微电子前沿技术的时候,当我们区(开源社区)的兄弟们越来越多的使用复杂得不能再复杂的函数来告诉我什么是编程艺术的时候,我开始对这些电源管理代码晕菜了.

可是没办法,谁叫我不幸生在中国呢,所以我觉得应该抽点时间,找点空闲,抛开代码看看理论.没有一点理论基础,这代码肯定是看不懂的.

电源管理其实发展挺多年了,也算是比较成熟了,主流的技术有两种,APM 和 ACPI,APM,即 Advanced Power Management,高级电源管理,ACPI,即 Advanced Configuration and Power Interface,高级配置和电源接口,相比之下,APM 容易实现,但是,APM 属于一个 BIOS 的 spec,也就是说需要 BIOS 的支持,这种情况过去在笔记本电脑中比较普遍,不过自从 ACPI 横空出世之后,APM 就将走向没落了,并被行家认为将在不久的将来从市场中消失,毕竟一种特性依赖于 BIOS 不是什么好事,这个时代强调的是独立.APM 是盖茨他们家和 Intel 一起于 1992 年提出的,而 ACPI 则是多了一家小日本的企业,东芝+Intel+Microsoft,这三家于 1996 年提出了 ACPI 1.0,Microsoft 的 Windows2000 也是第一个支持 ACPI 的操作系统.很显然,ACPI 是更为先进的技术,它提供了更为灵活的接口,功能也更为强大,它最有型的地方大概就在于它基本不需要 BIOS 插手,基本可以通过 OS 来搞定.它的出现就是为了替代 APM,或者说为了克服 APM 的不足.在如今的 Linux 发行版里,基本上你可以自己选择使用这两种电源管理的方式,默认应该是 ACPI.选择了其中一种就得关掉另一种,不可能说两种方法同时使用,道理很简单,古训有云:一山不能容二虎,除非一公和一母.(顺便友情提醒一下,电源管理只是 ACPI 的功能中的一部分,除此以外,ACPI 还有很多别的功能,干了不少和 BIOS 抢饭碗的事情.)

比如 Red Hat 中我们进入 setup 就可以看到系统服务里面有一个叫做 acpid 的,还有一个叫做 apmd 的,就是与这两种电源管理方式对应的服务进程.打开了一个就不要再打开另一个.



Ok,第一个概念,首先必须知道,Linux 中的电源管理作为一个子系统(subsystem),它是一个系统工程,这个工程规模大,波及范围广,技术复杂,建设周期长,材料设备消耗大,施工生产流动性强,受自然和社会环境因素影响大,如果说内存管理相当于我国的南水北调工程,那么电源管理就相当于我国的西电东送工程,这两者都可谓任务重,要求高,而且还被全区(开源社区)人民寄予厚望.谁也不希望劳民伤财之后造出一个像三峡大坝这样的垃圾工程来.别以为我这样说很夸张,我们都知道 Linux 2.6 内核一个最伟大的变化就是建立了一个新的统一的设备模型,可是你知道当年区领导决定建立这个设备模型的初衷是什么吗?就是为了让电源管理工作变得更加容易.(“The device model was originally intended to make power management tasks easier through the maintenance of a representation of the host system's hardware structure.”)只不过后来失去了党的英明指引,写代码的兄弟们走火入魔,踏上了一条不归路,把这个设备模型做得偏离了最初的方向,然而不曾想做出来的东西意义更大了.不仅解决了电源管理的复杂性,而且把所有的设备管理任务都给集中起来了.

那么我们先抛出几个问题,第一,电源管理的含义是什么?一个字,省电.具体来说,电源管理意味着让世界充满爱的同时,让所有的设备处于尽可能低的耗电状态.如果你计算机中某些部分没有被使用,那么就把它关闭(比如显示器)或者让它进入省电的睡眠模式(比如硬盘).

其次,什么情况设备要挂起呢?比如,合上笔记本,又比如用户自己定义了一个系统电源管理策略(像 30 分钟没有 console 活动的话就挂起),再比如设备自身有它的电源管理游戏规则(像一个设备五分钟没有活动的话就挂起).于是这就意味着有几种可能,一种是系统级的,即当你合上笔记本的时候,OS 负责通知所有的驱动程序,告诉它,需要 suspend,即驱动程序提供的 suspend 函数会被调用,这种情况在江湖上被人叫做 system pm,也叫 System Sleep Model,另一种情况,不是系统级的,设备级的,就是说我虽然没有合上笔记本,但就单个设备而言,如果我用户希望这个设备处于低耗电的状态,那么驱动程序也应该能够支持,这种情况被道上的兄弟称之为 runtime

pm,即 Runtime Power Management Model.换句话说,我不管别的设备死还是活,总之我这个设备自己想睡就睡,睡到自然醒,谁也别烦我.

因此,第三,从设备驱动的角度来说,我们应该如何作出自己应有的贡献呢?众所周知,电源管理最重要的两个概念就是 suspend 和 resume,即挂起和恢复.而设备驱动被要求保存好设备的上下文,即在挂起的时候,你作为设备驱动,你得保存好设备的一切状态信息,而在 resume 的时候,你要能够负责恢复这些信息.所以,你必须申请相关的 buffer,把东东存在里面,关键的时候拿出来恢复.总的来说,suspend 这个过程就是,上级下命令通知 driver,driver 保存状态,然后执行命令.即 notify before save state;save state before power down.而 resume 这个过程则是,power on and restore state.

第四,刚才说了,建立统一设备模型的初衷是为了打造更佳的电源管理模型,这究竟是如何体现的呢?我们说过,2.6 的内核,不管你是 usb 还是 pci 还是 scsi,你最终会有一棵树,会通过父子关系,兄弟关系把所有的设备连接起来,这样做不是为了体现亲情,而是电源管理的基础,因为操作系统需要以一种合理的顺序去唤醒或者催眠一堆的设备,即,比如,一个 PCI 总线上的设备必须在它的父设备睡眠之前先进入睡眠,反过来,又必须在它的父设备醒来以后才能醒来.开源战士范仲淹有一句话把这电源管理机制下的子设备诠释得淋漓尽致:先天下之睡而睡,后天下之醒而醒.

第五,从微观经济学来看,设备挂起意味着什么?意味着没有任何进出口贸易的发生,用英语说这叫,quiesce all I/O,即四个坚持,坚持不进行任何 DMA 操作,坚持不发送任何 IRQ,坚持不读写任何数据,坚持不接受上层驱动发过来的任何请求.

最后提两个术语.STR 和 STD,这是两种 Suspend 的状态.STR 即 Suspend to RAM,挂起到内存,STD 就是 Suspend to Disk,挂起到磁盘.STR 就是把系统进入 STR 前的工作状态数据都存放在内存中去.在 STR 状态下,电源仍然继续为内存和主板芯片组供电,以确保数据不丢失,而其它设备均处于关闭状态,系统的耗电量极低.一旦我们按下 Power 按钮,系统就被唤醒,马上从内存中读取数据并恢复到 STR 之前的工作状态,STR 的优点是休眠快唤醒也快,因为数据本来就在内存中.而 STD 则是把数据保存在磁盘中,很显然,保存在磁盘中要比保存在内存中慢.不过 STD 最酷的是因为它写到了磁盘中,所以即使电源完全断了,数据也不会丢失.STD 就是我们在 Windows 里面看到的那个 Hibernate,即冬眠,或曰休眠,而 STR 就是我们在 Windows 里面看到的那个 Standby.STR 和 STD 是计算机休眠的两种主要方式.关于 STD,多说两句,我想你永远不会忘记,第一次装 Linux 的时候,有人要你分区的时候分一个 swap 分区吧?这里就体现了 swap 分区的一个作用,如果你安装了 Suse 操作系统,看你的 grub 里面,一定有一项类似于 resume=/dev/sda4 吧,就是断电以后重起了之后,从这个分区里把东西读出来的意思.再补一句,ACPI 的状态一共有五种,分别是 S1,S2,S3,S4,S5,实际上 S4 就是 STD,而 STR 就是 S3,只不过 S1,S2,S3 差别不大,不过,在 Linux 中,S1 被叫做 Standby,而 S3 被叫做 STR.而 S5 就是 Shutdown.在 Linux 中说挂起,主要说的就是 S1,S3 和 S4.关于 S1,S3,S4 这三种状态,在 /sys/power/state 文件里可以有所体现,cat /sys/power/state, 你会看到"standby","mem","disk"的字样.

我们来做一个实验.你打开一个网络连接,比如你去某 FTP 站点下一部武藤兰的 A 片,大小 500M 的那种,然后正在下的时候(即现在进行时的下载),比如下到 100M 左右,或者说刚开始下,下了 3,5M 了,这时候你另外开一个终端,执行下面两个命令:

```
# echo shutdown > /sys/power/disk
```

```
# echo disk > /sys/power/state
```

你的机器将进入传说中的 Hibernate 状态,你如果觉得不够刺激,可以把电源线都拔掉,然后隔一会儿再按动电源开关,开机,你会发现,你又得重新面对 grub,重新选择启动选项,但是再次启动好了之后,那部 A 片仍然在继续下载.

以上这个实验做的就是 STD.当然,你自己做的话失败了别怪我,我早就说过,电源管理这部分是这几年里开源社区最 hot 的话题之一,问题其实很多,被你遇上了也不必惊讶.不过我需要提醒一下,做之前得确认 grub 里面 kernel 那行有那个 resume 的定义.基本上 Suse Linux 有这个,Redhat 默认好像没有设置这一项.另外,在第一个命令中,你还可以把 shutdown 换成 reboot,这样的话在你执行了第二条命令以后,系统会立刻重起,而不是直接进入 power off 的状态.重起之后你看到的效果也是一样的,原来什么样,起来之后就还是什么样.

而进行 STR 的实验就稍微复杂一点,我们这里不多说了,从设备驱动来说,没有必要知道这次挂起是 STR 还是 STD,总之,以上这个 STD 的实验自从第二个命令执行之后,各个驱动提供的 suspend 函数就会相继被执行,而之后重起的时候,就会相继调用各驱动提供的 resume 函数.PM core 那边知道如何遍历设备树,所以从 usb 这边来说,我们甚至不需要做太多递归的事情.

Ok,现在让我们来介绍一下 PM Core 那边给出的接口.我早就说过,Linux 中基本上就是这么几招.首先是把整个内核分成很多个子系统,或者美其名曰 Subsystem,然后很多子系统又分为一个核心部分和其他部分,比如我们的 usb,就是 usbcore 和其它部分,usb core 这部分负责提供整个 usb 子系统的初始化,主机控制器的驱动,(当然主机控制器的驱动由于 host controller 的种类越来越多,host controller driver 也便单独构成了一个目录,所以我们有时候说 usb core 实际上指的是 drivers/usb/core 和 drivers/usb/host 目录,但这是相对外围设备而言的,实际上 usb core 本身是一个模块,而主机控制器的驱动又是单独成为一个模块.)而外围设备的驱动,比如 usb-storage,就是直接使用 usb core 提供出来的接口即可.这种伎俩在 Linux 内核中比比皆是.而电源管理也是如此,在电源管理子系统中,有一个叫做 PM core 的,它所包含的是一些核心的代码.而其它各大子系统里要加入 power management 的功能,就必须使用 PM core 提供的接口函数.因此,现在是时候让我们来看一看了.

首先,既然我们说电源管理是一个系统工程,那么必然是自上而下的一次大规模改革,不仅仅是设备驱动需要支持它,总线驱动也必须提供相应的支持.还记得在讲八大函数的时候,我们贴出来的结构体变量 struct bus_type usb_bus_type 么?其中我们把.suspend 赋值为 usb_suspend,而把.resume 赋值为 usb_resume.也就是说这两个函数是各类总线不相同的,每类总线都需要实现自己特定的函数,比如 PCI 总线,我们也能看到类似的定义,来自 drivers/pci/pci-driver.c:

```
542 struct bus_type pci_bus_type = {
543         .name           = "pci",
544         .match          = pci_bus_match,
545         .uevent         = pci_uevent,
546         .probe          = pci_device_probe,
547         .remove         = pci_device_remove,
548         .suspend        = pci_device_suspend,
549         .suspend_late   = pci_device_suspend_late,
550         .resume_early   = pci_device_resume_early,
```

```
551         .resume         = pci_device_resume,
552         .shutdown       = pci_device_shutdown,
553         .dev_attrs      = pci_dev_attrs,
```

它的.suspend 就被赋值为 pci_device_suspend,而.resume 被赋值为 pci_device_resume.

而设备驱动呢?提供自己的 suspend/resume 函数,于是最终总线的那个 suspend/resume 函数就会去调用设备自己的 suspend/resume,比如我们回归 usb,hub 驱动提供了两个函数 ,hub_suspend/hub_resume, 于是 usb_suspend/usb_resume 最终就会调用 hub_suspend/hub_resume,换句话说,总线的 suspend/resume 是包装,是面子工程,而设备自己的 suspend/resume 是实质.

所以,接下来我们有两种选择,第一,直接讲 hub_suspend/hub_resume, 第二,从 usb_suspend/usb_resume 开始讲.很显然,前者相对简单,选择前者意味着我们的故事在讲完这两个函数之后就可以结束.我也轻松你也轻松.而选择后者意味着我们选择了一条更加艰难的道路.选择是一个崭新的开端,选择高耸入云的峭崖便需有"路漫漫其修远兮,吾将上下而求索"的信念;选择波涌浪滚的大海便需有"直挂云帆济沧海"的壮志豪情;选择寒风劲厉的荒漠便需有"醉卧沙场君莫笑,古来征战几人回"的博大胸怀.我很无奈,但是人生是一定要选择的,你不可能什么都抓在手上,如果这样,你什么都会失去!经过谨慎思考,我还是决定铤而走险,去探索一下这个复杂的乱世.

但在真正开始讲代码之前,我还是想强调一下,首先,基本上如果你看完了此前的 hub_events 和那八大函数,你就算是了解 hub 驱动了.下面的代码属于 usb core 中电源管理的部分,对于大多数人来说是可以不用再看的,除非:

1. 你和我一样,很无聊,很孤独,对人生很失望.
2. 或者,你欲投身于开源社区的开发事业,想了解最新的开发进展,你想和 Alan Stern,想和 Oliver Neukum,想和 David Brownell 同流合污,为 Linux 内核中 usb 子系统做出自己的贡献.

支持计划生育--看代码的理由

北大校长马寅初先生曾斩钉截铁地跟毛主席讲:"中国人口太多是因为农村晚上没有电."

因此,为了支持计划生育这项基本国策,每一个男人都有义务认真看一下电源管理的代码.

另一方面,虽然现在已经不住在农村了,但我一直坚定不移的认为,这个世界,最慢的是我家的网速,最快的是我家电表的转速.

所以,为了了解如何让电表转速更慢,让我们一起来看看 usb 子系统里是如何支持电源管理的吧.

上节说了应该从 usb_suspend/usb_resume 开始看,那就开始吧.

usb_suspend/usb_resume 这两个函数很显然是一对,但是我们不可能同时讲,只能一个一个来.倒不是故意把它们拆开,实在是没有办法.须知,形影不离并不代表相知相惜,感情在乎的是心与心的距离.两情若是久长时,又岂在朝朝暮暮. 先讲 usb_suspend,再讲 usb_resume.

来看 usb_suspend,定义于 drivers/usb/core/driver.c:

```
1497 static int usb_suspend(struct device *dev, pm_message_t message)
1498 {
1499         if (!is_usb_device(dev))        /* Ignore PM for interfaces */
1500                 return 0;
1501                 return  usb_external_suspend_device(to_usb_device(dev),
message);
1502 }
```

刚说过,usb_suspend 是 usb 子系统提供给 PM core 调用的,所以这里两个参数 dev/message 都是那边传递过来的,要不是 usb device 当然就不用做什么了.直接返回.然后调用 usb_external_suspend_device(),后者也是来自 drivers/usb/core/driver.c.

```
1443 /**
1444  * usb_external_suspend_device - external suspend of a USB device and
its interfaces
1445  * @udev: the usb_device to suspend
1446  * @msg: Power Management message describing this state transition
1447  *
1448  * This routine handles external suspend requests: ones not generated
1449  * internally by a USB driver (autosuspend) but rather coming from the
user
1450  * (via sysfs) or the PM core (system sleep).  The suspend will be carried
1451  * out regardless of @udev's usage counter or those of its interfaces,
1452  * and regardless of whether or not remote wakeup is enabled.  Of
course,
1453  * interface drivers still have the option of failing the suspend (if
1454  * there are unsuspended children, for example).
1455  *
1456  * The caller must hold @udev's device lock.
1457  */
1458 int usb_external_suspend_device(struct usb_device *udev,
pm_message_t msg)
1459 {
1460         int     status;
1461
1462         usb_pm_lock(udev);
1463         udev->auto_pm = 0;
1464         status = usb_suspend_both(udev, msg);
1465         usb_pm_unlock(udev);
```

```
1466            return status;
1467 }
```

1462 行和 1465 行,锁的代码暂时先一律飘过.

我们看到,这个函数就做了两件事情,第一,让 udev 的 auto_pm 为 0,第二,调用 usb_suspend_both.

继续跟踪 usb_suspend_both.仍然是来自于 drivers/usb/core/driver.c:

```
 993 /**
 994  * usb_suspend_both - suspend a USB device and its interfaces
 995  * @udev: the usb_device to suspend
 996  * @msg: Power Management message describing this state transition
 997  *
 998  * This is the central routine for suspending USB devices.  It calls the
 999  * suspend methods for all the interface drivers in @udev and then calls
1000  * the suspend method for @udev itself.  If an error occurs at any stage,
1001  * all the interfaces which were suspended are resumed so that they remain
1002  * in the same state as the device.
1003  *
1004  * If an autosuspend is in progress (@udev->auto_pm is set), the routine
1005  * checks first to make sure that neither the device itself or any of its
1006  * active interfaces is in use (pm_usage_cnt is greater than 0).  If they
1007  * are, the autosuspend fails.
1008  *
1009  * If the suspend succeeds, the routine recursively queues an autosuspend
1010  * request for @udev's parent device, thereby propagating the change up
1011  * the device tree.  If all of the parent's children are now suspended,
1012  * the parent will autosuspend in turn.
1013  *
1014  * The suspend method calls are subject to mutual exclusion under control
1015  * of @udev's pm_mutex.  Many of these calls are also under the protection
1016  * of @udev's device lock (including all requests originating outside the
1017  * USB subsystem), but autosuspend requests generated by a child device or
1018  * interface driver may not be.  Usbcore will insure that the method calls
1019  * do not arrive during bind, unbind, or reset operations.  However, drivers
1020  * must be prepared to handle suspend calls arriving at unpredictable times.
1021  * The only way to block such calls is to do an autoresume (preventing
1022  * autosuspends) while holding @udev's device lock (preventing outside
```

```
1023  * suspends).
1024  *
1025  * The caller must hold @udev->pm_mutex.
1026  *
1027  * This routine can run only in process context.
1028  */
1029 static int usb_suspend_both(struct usb_device *udev, pm_message_t
msg)
1030 {
1031         int                     status = 0;
1032         int                     i = 0;
1033         struct usb_interface    *intf;
1034         struct usb_device       *parent = udev->parent;
1035
1036         if (udev->state == USB_STATE_NOTATTACHED ||
1037                         udev->state == USB_STATE_SUSPENDED)
1038                 goto done;
1039
1040                                         udev->do_remote_wakeup     =
device_may_wakeup(&udev->dev);
1041
1042         if (udev->auto_pm) {
1043                 status = autosuspend_check(udev);
1044                 if (status < 0)
1045                         goto done;
1046         }
1047
1048         /* Suspend all the interfaces and then udev itself */
1049         if (udev->actconfig) {
1050                 for (; i < udev->actconfig->desc.bNumInterfaces; i++) {
1051                         intf = udev->actconfig->interface[i];
1052                         status = usb_suspend_interface(intf, msg);
1053                         if (status != 0)
1054                                 break;
1055                 }
1056         }
1057         if (status == 0)
1058                 status = usb_suspend_device(udev, msg);
1059
1060         /* If the suspend failed, resume interfaces that did get suspended
*/
1061         if (status != 0) {
1062                 while (--i >= 0) {
1063                         intf = udev->actconfig->interface[i];
```

```
1064                        usb_resume_interface(intf);
1065                }
1066
1067                /* Try another autosuspend when the interfaces aren't
busy */
1068                if (udev->auto_pm)
1069                        autosuspend_check(udev);
1070
1071        /* If the suspend succeeded, propagate it up the tree */
1072        } else {
1073                cancel_delayed_work(&udev->autosuspend);
1074                if (parent)
1075                        usb_autosuspend_device(parent);
1076        }
1077
1078 done:
1079         // dev_dbg(&udev->dev, "%s: status %d\n", __FUNCTION__,
status);
1080        return status;
1081 }
```

这里有两个重要的概念,autosuspend/autoresume.autosuspend,即自动挂起,这是由 driver 自行决定,它自己进行判断,当它觉得应该挂起设备的时候,它就会去 Just do it!关于 autosuspend 我们后面会讲.

1040行, device_may_wakeup(),我们前面说过,设备有没有被唤醒的能力有一个flag可以标志,即 can_wakeup,那么如果有这种能力,用户仍然可以根据实际需要关掉这种能力,或者打开这种能力,这就体现在 sysfs 下的一个文件.比如:

```
localhost:~ # cat /sys/bus/usb/devices/1-5/power/wakeup
enabled
localhost:~ # cat /sys/bus/usb/devices/1-5\:1.0/power/wakeup

localhost:~ #
```

可以看到后者的输出值为空,这说明该设备是不支持 remote wakeup 的,换句话说,其 can_wakeup 也应该是设置为了 0,这种情况 device_may_wakeup 返回值必然是 false,而前者的输出值为 enabled,说明该设备是支持 remote wakeup 的,并且此刻 remote wakeup 的特性是打开的.别的设备也一样,用户可以通过 sysfs 来进行设置,你可以把 wakeup 从 enabled 改为 disabled.

为什么需要有这么一个 sysfs 的接口呢?我们知道 usb 设备有一种特性,叫做 remote wakeup,这种特性不是每个 usb 设备都支持,而一个设备是否支持 remote wakeup 可以在它的配置描述符里体现出来,但问题是,以前,区里的人们总是相信设备的各种描述符,可是你知道,现实生活中,被骗比骗人容易.制造商生产出来的产品总是有着各种问题的,它的各种描述符也许只是一种假

象,比如,很多案例表明,一个设备的配置描述里声称自己支持 remote wakeup,但是实际上却并不支持,当它进入睡眠之后,你根本唤不醒它.所以弟兄们学乖了,决定为用户提供一种选择,即,用户可以自己打开或者关闭这种特性,就为了对付这种设备掩耳盗铃自欺欺人的社会现象.不过仔细想想,其实制造商也不容易,也许他们想通过这些设备传达一种思想,那就是:人们总说快乐很难,其实快乐很容易,只要你学会欺骗自己,你天天都是快乐的.

那么我们这里用 udev->do_remote_wakeup 来记录下这个值,日后会用得着的,到时候再看.

1042 行,刚才咱们设置了 auto_pm 为 0,所以这段不会执行.如果我们设置了 auto_pm 为 1,那么就会调用.autosuspend_check(),这个函数我们以后再回过来看.现在先根据我们实际的情景走,不执行.auto_pm 为 0 就是告诉人们我们现在没有做 autosuspend.以后我们会看到,auto_pm 这个变量将在 autosuspend 的代码中被设置为 1.

接下来是一段循环,按接口进行循环,即,设备有几个接口,就循环几次,因为我们知道 usb 中,驱动程序往往是针对 interface 的,而不是针对 device 的,所以每一个 interface 就可能对应一个驱动程序,进而就可能有一个单独的 suspend 函数.

遍历各个接口之后,usb_suspend_interface 这个函数如果能够顺利的把各个接口都给挂起了,那么再调用一个 usb_suspend_device 函数来执行一次总的挂起.为什么要有这两个函数我们看了就知道.先看第一个,usb_suspend_interface,来自 drivers/usb/core/driver.c:

```
850 /* Caller has locked intf's usb_device's pm mutex */
851 static int usb_suspend_interface(struct usb_interface *intf, pm_message_t
msg)
852 {
853         struct usb_driver       *driver;
854         int                     status = 0;
855
856         /* with no hardware, USB interfaces only use FREEZE and ON
states */
857                                 if (interface_to_usbdev(intf)->state ==
USB_STATE_NOTATTACHED ||
858                                         !is_active(intf))
859                 goto done;
860
861         if (intf->condition == USB_INTERFACE_UNBOUND)   /* This
can't happen */
862                 goto done;
863         driver = to_usb_driver(intf->dev.driver);
864
865         if (driver->suspend && driver->resume) {
866                 status = driver->suspend(intf, msg);
867                 if (status == 0)
868                         mark_quiesced(intf);
869                 else if (!interface_to_usbdev(intf)->auto_pm)
```

```
870                         dev_err(&intf->dev, "%s error %d\n",
871                                         "suspend", status);
872         } else {
873                 // FIXME else if there's no suspend method, disconnect...
874                 // Not possible if auto_pm is set...
875                 dev_warn(&intf->dev, "no suspend for driver %s?\n",
876                                 driver->name);
877                 mark_quiesced(intf);
878         }
879
880 done:
881           // dev_dbg(&intf->dev, "%s: status %d\n", __FUNCTION__,
status);
882         if (status == 0)
883                 intf->dev.power.power_state.event = msg.event;
884         return status;
885 }
```

一路陪我们走过来的兄弟们一定不会看不懂这个函数,最关键的代码就是 866 那行,driver->suspend(intf,msg),这就是调用具体的 interface 所绑定的那个驱动程序的 suspend 函数.比如,对于 hub 来说,这里调用的就是 hub_suspend()函数.具体的 hub_suspend()我们倒是不用先急着看,顺着现在的情景往下过一遍,至于 hub_suspend/hub_resume,咱们跟它秋后算账.

mark_quiesced 是一个内联函数,咱们一次性把相关的三个内联函数都贴出来,来自 drivers/usb/core/usb.h 中:

```
 98 /* Interfaces and their "power state" are owned by usbcore */
 99
100 static inline void mark_active(struct usb_interface *f)
101 {
102         f->is_active = 1;
103 }
104
105 static inline void mark_quiesced(struct usb_interface *f)
106 {
107         f->is_active = 0;
108 }
109
110 static inline int is_active(const struct usb_interface *f)
111 {
112         return f->is_active;
113 }
```

其实就是 struct usb_interface 中有一个成员,unsigned is_active,这位为 1 就标志该 interface 没有 suspended.反之就是记录该 interface 已经是 suspended 了.suspended 了也被老外称作 quiesced,反之就叫做 active.所以呢,这里对应的两个函数就叫做 mark_active 和 mark_quiesced.

电源管理的四大消息

如果真的有一种水
可以让你让我喝了不会醉
那么也许有一种泪
可以让你让我流了不伤悲

如果真的有一种硬件
可以让你让我用了不耗电
那么也许有一种代码
可以让你让我看了不得不崩溃

这一节涉及电源管理中的一些核心概念,所以你如果可以选择看,也可以选择不看,it's up to you.

883 行,令 dev.power.power_state.event 等于 msg.event.现在是时候来讲一讲两样东西了,一个是 msg,一个是 event.细心的你一定注意到连续几个函数的第二个参数都是 pm_message_t msg,它姓甚名谁,是何许人也?来自 include/linux/pm.h,是电源管理部分的一个很重要的结构体,

```
202 typedef struct pm_message {
203         int event;
204 } pm_message_t;
```

这个结构体也够酷,居然只有一个成员,那就是 event.很显然 pm_message 的意思就是 PM core 传递给设备驱动的消息,事实上 suspend 有好几种境界,比如一种是简单的停止驱动并且把设备设置为低功耗的状态,一种是停止驱动但并不真正的把设备设置为低功耗的状态,(比如因为你实际上只想做一次 snapshot,或者叫做一次系统内存 image 快照,而并不需要真正的把设备持续挂起),还有一种更复杂的状态叫做 PRETHAW,稍后会说.那么你说 PM core 部分如何让设备驱动知道你想进入哪种境界?有这么一个参数来传达这种思想不就 Ok 了么?关于这个 event,我们来看 include/linux/pm.h 中的介绍,

```
206 /*
207  * Several driver power state transitions are externally visible, affecting
208  * the state of pending I/O queues and (for drivers that touch hardware)
209  * interrupts, wakeups, DMA, and other hardware state.  There may also
be
210  * internal transitions to various low power modes, which are transparent
211  * to the rest of the driver stack (such as a driver that's ON gating off
```

```
212  * clocks which are not in active use).
213  *
214  * One transition is triggered by resume(), after a suspend() call; the
215  * message is implicit:
216  *
217  * ON            Driver starts working again, responding to hardware
events
218  *               and software requests.  The hardware may have gone
through
219  *               a power-off reset, or it may have maintained state from
the
220  *               previous suspend() which the driver will rely on while
221  *               resuming.  On most platforms, there are no restrictions
on
222  *               availability of resources like clocks during resume().
223  *
224  * Other transitions are triggered by messages sent using suspend().  All
225  * these transitions quiesce the driver, so that I/O queues are inactive.
226  * That commonly entails turning off IRQs and DMA; there may be rules
227  * about how to quiesce that are specific to the bus or the device's type.
228  * (For example, network drivers mark the link state.)  Other details may
229  * differ according to the message:
230  *
231  * SUSPEND       Quiesce, enter a low power device state appropriate for
232  *               the upcoming system state (such as PCI_D3hot), and
enable
233  *               wakeup events as appropriate.
234  *
235  * FREEZE        Quiesce operations so that a consistent image can be
saved;
236  *               but do NOT otherwise enter a low power device state, and
do
237  *               NOT emit system wakeup events.
238  *
239  * PRETHAW       Quiesce as if for FREEZE; additionally, prepare for
restoring
240  *               the system from a snapshot taken after an earlier FREEZE.
241  *               Some drivers will need to reset their hardware state
instead
242  *               of preserving it, to ensure that it's never mistaken for the
243  *               state which that earlier snapshot had set up.
244  *
245  * A minimally power-aware driver treats all messages as SUSPEND, fully
246  * reinitializes its device during resume() -- whether or not it was reset
```

```
247  * during the suspend/resume cycle -- and can't issue wakeup events.
248  *
249  * More power-aware drivers may also use low power states at runtime as
250  * well as during system sleep states like PM_SUSPEND_STANDBY.  They
may
251  * be able to use wakeup events to exit from runtime low-power states,
252  * or from system low-power states such as standby or suspend-to-RAM.
253  */
254
255 #define PM_EVENT_ON 0
256 #define PM_EVENT_FREEZE 1
257 #define PM_EVENT_SUSPEND 2
258 #define PM_EVENT_PRETHAW 3
259
260  #define  PMSG_FREEZE          ((struct  pm_message){  .event  =
PM_EVENT_FREEZE, })
261  #define  PMSG_PRETHAW         ((struct  pm_message){  .event  =
PM_EVENT_PRETHAW, })
262  #define  PMSG_SUSPEND         ((struct  pm_message){  .event  =
PM_EVENT_SUSPEND, })
263  #define  PMSG_ON               ((struct  pm_message){  .event  =
PM_EVENT_ON, })
```

貌似一大段,其实就是定义了八个宏,不过这八个宏咱们在 usb 子系统里面都会遇到,只是早晚的事情.我们慢慢来说,首先,message,即消息,也被叫做事件,即 event,于是每条具体的消息最终被道上的兄弟以一个叫做事件码的东西区分,即 event code.而当前 Linux 内核中,一共有四种事件码,它们是:

ON,或者叫 PM_EVENT_ON,这个事件码实际上是不会用来传达消息的,倒是经常用来表征设备当前所处于的一种状态,即,告诉世界,本设备当前并没有处于挂起的状态,咱目前一切正常.我们这里看到的这个 dev.power.power_state.event 就是用来记录设备的这个状态的,如果没有挂起,那么这个值就应该是 PM_EVENT_ON.

EVENT_SUSPEND,或曰 PM_EVENT_SUSPEND,这个不用说了,传达这个消息的意思就是想让设备进入低功耗的状态,用 David Brownell 在 Documentation/power/devices.txt 文件中原话来说,就是 put hardware into a low-power state.而我们这里的赋值令 dev.power.power_state.event 等于 msg.event.这就很好理解了,在调用了设备驱动的 suspend 函数去执行实际的任务之后,为设备记录下这个事件,从此以后你这个设备就不再是处于 PM_EVENT_ON 的状态了,你已经挂起了.

EVENT_FREEZE,或曰 PM_EVENT_FREEZE,和 EVENT_SUSPEND 的区别咱们前面也说了,这里注释也说了,总之就是挂起的境界不太一样,毕竟其挂起的目的也不一样.

EVENT_PRETHAW,或曰 PM_EVENT_PRETHAW,这就属于挂起的另一种境界.这一事件类型是为了支持 STD 的.确切地说,2.6 内核实现了一种叫做 swsusp 的 STD 方法.(swsusp 就是

Software Suspend 的意思.)而当时 David Brownell 在 Linux 中引入这么一个宏的目的就是为了支持 swsusp 的 snapshot image 恢复,这件事情的缘由是,我们知道软件挂起需要做系统内存映像快照,然后要恢复这个快照,然而,对于某些设备来说,直接恢复这个快照会导致错误,因为 resume()函数通常会读硬件的状态,然后它会认为硬件的这种状态是被 suspend 函数设置的,或者另一种情况是由于掉电重起而复位(reset)的,然而 swsusp 比较变态,它会像 kexec 一样,先使用另一个小内核实例,然后 load 那个 snapshot,从而切换成新的内核.而问题出在哪里呢?问题就在于 swsusp 会在 load snapshot 之前把硬件们都 suspend,于是硬件们将进入某个状态,请你注意了,由于这时候那个 snapshot 内核还没有加载,所以这时候硬件们进入的这个状态只能说是不三不四的状态.(注:kexec 是 Eric Biederman 的作品,其作用是让您可以从当前正在运行的内核直接引导到一个新内核.Hoho,不懂了吧?不懂就不懂吧,这个我也没法让您懂,网上有关于 kexec 的介绍,不过要对 kexec 有一个直观的了解的话可以做一次 kdump 的实验,网上也有文档,RHEL4/5 发行版里边都包含有相关的 rpm 包,没记错的话,Redhat 官方网站上有介绍如何在他们家的系统上配置 kdump 的文档.经常调试内核的兄弟们应该会比较清楚.不懂也没有关系,总之,你只需要知道,自从有了 kexec,你的系统里就不再是一个内核了,它可以有两个,启动的时候,先启动一个小的,然后小的负责加载大的,当大了挂了之后小的可以让它快速的重起,也就是说,这样做的好处就是在调试这个大的内核的时候,当这个大的内核崩溃了之后小的就可以让它重起.)而一旦硬件们进入了那个不三不四的状态之后,等到 snapshot 被加载了之后,resume 会执行,resume 并不知道这个状态是个不三不四的状态,它以为这个状态就是正常的那种 suspend 的状态或者是 reset 之后的状态,这样子就会出现问题,从而导致设备没法工作.因此需要对 resume 的设备执行一次 reset,而这个宏就为了支持这个.具体来说,比如我们可以在 EHCI 主机控制器的驱动程序中看到如下的代码,来自 drivers/usb/host/ehci-pci.c:

```
    257         /* make sure snapshot being resumed re-enumerates everything
*/
    258         if (message.event == PM_EVENT_PRETHAW) {
    259                 ehci_halt(ehci);
    260                 ehci_reset(ehci);
    261         }
```

同样在 UHCI 或者 OHCI 驱动中也能看到类似的代码,比如 OHCI 的,drivers/usb/host/ohci-pci.c:

```
    229         /* make sure snapshot being resumed re-enumerates everything
*/
    230         if (message.event == PM_EVENT_PRETHAW)
    231                 ohci_usb_reset(ohci);
```

比如 UHCI 的,drivers/usb/host/uhci-hcd.c:

```
    768         /* make sure snapshot being resumed re-enumerates everything
*/
    769         if (message.event == PM_EVENT_PRETHAW)
    770                 uhci_hc_died(uhci);
```

我们如果到代码中具体去看,会发现这些代码来自相应驱动中的 suspend()函数的最后,即无论如何把这些设备给我 reset 一次,以清除在小内核上执行 suspend()留下的恶果,从而保证 resume 函数能够正确的工作.之所以专门拿主机控制器驱动程序来举例子说,是因为当前内核中,USB 子系统里会使用到 PRETHAW 的只有 Host controller driver,因为只有它们有这样一个问题.

另外,关于 SUSPEND,FREEZE,它们都属于 suspended 状态的一种,而设备只可能由 ON 进入这两种状态之一或者由这两种状态之一进入 ON,即比如,设备可以从 ON 进入 FREEZE,也可以从 ON 进入 SUSPEND,但设备绝不可以从 FREEZE 进入 SUSPEND,也不可以从 SUSPEND 进入 FREEZE.

当然,说这么细,估计您早晕了,别慌,其实我也晕了.不过没关系,因为实际上你看到最多的还是 PM_EVENT_ON 和 PM_EVENT_SUSPEND,另外两个宏您在整个 usb 子系统也难得遇上几次.至少在咱们的 Hub 相关的这个故事中,您不会遇见另外那两个宏被使用.

将 suspend 分析到底

伫倚危楼风细细
望极春愁
黯黯生天际
草色烟光残照里
无言谁会凭栏意

拟把疏狂图一醉
对酒当歌
强乐还无味
衣带渐宽终不悔
为伊消得人憔悴

北宋词人柳永曾用这首蝶恋花来抒发对 Linux 内核中电源管理部分代码的无奈.当年柳永痛苦的看这代码看得想跳楼自尽.这首词,上片写登楼伫望情景.以细风,草色,烟光,残阳几个关合着忧愁的意象,组成一幅黄昏春望图,多层次地描摹写词人愁之景,愁之态,笔意婉约.下片抒情,直抒胸臆,写词人情深志坚.“拟把”、“强乐”三句辞意顿折,写词人欲借疏狂之歌呼,陶然之酣醉,谋求醉而忘忧,歌而暂欢,以摆脱读不懂代码的压抑,却落得个“还无味”的无聊和空虚,可见其愁之浓深、刻骨,竟无法排遣.最后揭明词人对待这种愁的果决态度:“终不悔”.“为伊”,方始画龙点睛地道破憔悴无悔的隐秘:为了看懂这代码,我亦值得憔悴、瘦损,以生命相托!语直情切,挟带着市民式的激情,真是荡气回肠.全词成功地刻画出一个志诚男子的形象,描写心理充分细腻,尤其是词的最后两句,直抒胸臆,画龙点睛般地揭示出主人公的精神境界,被王国维称为“专作情语而绝妙者”.

我知道你也会觉得电源管理这部分的代码显得很复杂,调用关系一层又一层.但我们只能继续往下看.如果没有问题,usb_suspend_interface 函数就这么返回了.返回值为 status,当然就是 0.然后我们回到 usb_suspend_both,接着看下一个函数 usb_suspend_device().同样来自 drivers/usb/core/driver.c:

```
795 /* Caller has locked udev's pm_mutex */
796 static int usb_suspend_device(struct usb_device *udev, pm_message_t
msg)
797 {
798         struct usb_device_driver        *udriver;
799         int                             status = 0;
800
801         if (udev->state == USB_STATE_NOTATTACHED ||
802                         udev->state == USB_STATE_SUSPENDED)
803                 goto done;
804
805         /* For devices that don't have a driver, we do a standard suspend.
*/
806         if (udev->dev.driver == NULL) {
807                 udev->do_remote_wakeup = 0;
808                 status = usb_port_suspend(udev);
809                 goto done;
810         }
811
812         udriver = to_usb_device_driver(udev->dev.driver);
813         status = udriver->suspend(udev, msg);
814
815 done:
816          // dev_dbg(&udev->dev, "%s: status %d\n", __FUNCTION__,
status);
817         if (status == 0)
818                 udev->dev.power.power_state.event = msg.event;
819         return status;
820 }
```

这里有一个新鲜的东西,过去我们知道 usb 设备驱动程序都是针对 interface 的,不是针对 device 的,所以很早我们就见到过一个叫做 struct usb_driver 的结构体,但是敏感的你是否注意到 2.6.22.1 的内核中还有另一个结构体,struct usb_device_driver 呢?它定义于 include/linux/usb.h 中:

```
859 /**
860  * struct usb_device_driver - identifies USB device driver to usbcore
861  * @name: The driver name should be unique among USB drivers,
862  *       and should normally be the same as the module name.
863  * @probe: Called to see if the driver is willing to manage a particular
864  *       device.  If it is, probe returns zero and uses dev_set_drvdata()
865  *       to associate driver-specific data with the device.  If unwilling
866  *       to manage the device, return a negative errno value.
867  * @disconnect: Called when the device is no longer accessible, usually
868  *       because it has been (or is being) disconnected or the driver's
```

```
869  *       module is being unloaded.
870  * @suspend: Called when the device is going to be suspended by the system.
871  * @resume: Called when the device is being resumed by the system.
872  * @drvwrap: Driver-model core structure wrapper.
873  * @supports_autosuspend: if set to 0, the USB core will not allow autosuspend
874  *       for devices bound to this driver.
875  *
876  * USB drivers must provide all the fields listed above except drvwrap.
877  */
878 struct usb_device_driver {
879         const char *name;
880
881         int (*probe) (struct usb_device *udev);
882         void (*disconnect) (struct usb_device *udev);
883
884             int (*suspend) (struct usb_device *udev, pm_message_t message);
885         int (*resume) (struct usb_device *udev);
886         struct usbdrv_wrap drvwrap;
887         unsigned int supports_autosuspend:1;
888 };
889 #define to_usb_device_driver(d) container_of(d, struct usb_device_driver, \
890                 drvwrap.driver)
```

我们以前说过,usb 设备驱动程序往往是针对 interface 的,而不是针对 device 的,换言之,一个 interface 对应一个 driver,这一情况到今天来看仍然是正确的,但将来就未必了,因为有一些行为是针对整个 device 的,比如电源管理中的挂起,可能整个设备需要统一的行为,而不是说每个 interface 可以单独行动,想干嘛干嘛.一个设备多个 interface,那么它们就是一个整体,一个整体对外就会有整体的表现.interface driver 就是专门处理各个 interface 的个性的,而 device driver 么,就用来对付整体.而这里我们看到的两个函数 usb_suspend_device 和 usb_suspend_interface 就是这种情况的体现.而 usb_suspend_device 这段代码的意图更是相当的明显,如果有 device driver,那就调用它的 suspend 函数,如果没有,就调用一个通用的函数,usb_port_suspend.我们来看一下这个通用的 suspend 函数.

```
1684 /*
1685  * usb_port_suspend - suspend a usb device's upstream port
1686  * @udev: device that's no longer in active use
1687  * Context: must be able to sleep; device not locked; pm locks held
1688  *
1689  * Suspends a USB device that isn't in active use, conserving power.
1690  * Devices may wake out of a suspend, if anything important happens,
1691  * using the remote wakeup mechanism.  They may also be taken out of
```

```
1692  * suspend by the host, using usb_port_resume().  It's also routine
1693  * to disconnect devices while they are suspended.
1694  *
1695  * This only affects the USB hardware for a device; its interfaces
1696  * (and, for hubs, child devices) must already have been suspended.
1697  *
1698  * Suspending OTG devices may trigger HNP, if that's been enabled
1699  * between a pair of dual-role devices.  That will change roles, such
1700  * as from A-Host to A-Peripheral or from B-Host back to B-Peripheral.
1701  *
1702  * Returns 0 on success, else negative errno.
1703  */
1704 int usb_port_suspend(struct usb_device *udev)
1705 {
1706         return __usb_port_suspend(udev, udev->portnum);
1707 }
```

原来是一个幌子,真正干实事的是__usb_port_suspend(),当然从这些函数的名字我们也可以看出,实际上针对整个设备的挂起是与 usb hub 的某个端口有关的,确切的说就是设备所连接的那个端口,这也就是为什么这两个函数都是出现在drivers/usb/core/hub.c中,而从这里的注释我们也不难看出,这里的目标就是挂起设备所连接的那个端口.而在此之前,我们已经调用 usb_suspend_interface 挂起了设备的每一个 interface.而如果是 hub 的话,会先要求挂起其所有的子设备.这一点不用我们操心,因为有设备树的存在,PM core 那边自然就知道如何做这件事情了.向伟大的 PM core 致敬,少先队员行队礼,非少先队员行注目礼.

```
1644 /*
1645  * Devices on USB hub ports have only one "suspend" state, corresponding
1646  * to ACPI D2, "may cause the device to lose some context".
1647  * State transitions include:
1648  *
1649  *   - suspend, resume ... when the VBUS power link stays live
1650  *   - suspend, disconnect ... VBUS lost
1651  *
1652  * Once VBUS drop breaks the circuit, the port it's using has to go through
1653  * normal re-enumeration procedures, starting with enabling VBUS power.
1654  * Other than re-initializing the hub (plug/unplug, except for root hubs),
1655  * Linux (2.6) currently has NO mechanisms to initiate that:  no khubd
1656  * timer, no SRP, no requests through sysfs.
1657  *
1658  * If CONFIG_USB_SUSPEND isn't enabled, devices only really suspend when
1659  * the root hub for their bus goes into global suspend ... so we don't
1660  * (falsely) update the device power state to say it suspended.
1661  */
1662 static int __usb_port_suspend (struct usb_device *udev, int port1)
```

```
1663 {
1664         int      status = 0;
1665
1666         /* caller owns the udev device lock */
1667         if (port1 < 0)
1668                 return port1;
1669
1670         /* we change the device's upstream USB link,
1671          * but root hubs have no upstream USB link.
1672          */
1673         if (udev->parent)
1674                 status = hub_port_suspend(hdev_to_hub(udev->parent),
port1,
1675                                 udev);
1676         else {
1677                 dev_dbg(&udev->dev, "usb %ssuspend\n",
1678                                 udev->auto_pm ? "auto-" : "");
1679                 usb_set_device_state(udev, USB_STATE_SUSPENDED);
1680         }
1681         return status;
1682 }
```

我倒,貌似还是一个幌子,真正的幕后英雄是 hub_port_suspend.udev->parent 为空的是Root Hub,对于Root Hub,只要设置设备状态为USB_STATE_SUSPENDED即可.因为子设备已经挂起了.而 Root Hub 本身不存在说接在哪个 port 的问题.于是我们来看hub_port_suspend.

```
1587 /*
1588  * Selective port suspend reduces power; most suspended devices draw
1589  * less than 500 uA.  It's also used in OTG, along with remote wakeup.
1590  * All devices below the suspended port are also suspended.
1591  *
1592  * Devices leave suspend state when the host wakes them up.  Some
devices
1593  * also support "remote wakeup", where the device can activate the USB
1594  * tree above them to deliver data, such as a keypress or packet.  In
1595  * some cases, this wakes the USB host.
1596  */
1597 static int hub_port_suspend(struct usb_hub *hub, int port1,
1598                 struct usb_device *udev)
1599 {
1600         int      status;
1601
1602         // dev_dbg(hub->intfdev, "suspend port %d\n", port1);
1603
```

```
1604        /* enable remote wakeup when appropriate; this lets the device
1605         * wake up the upstream hub (including maybe the root hub).
1606         *
1607         * NOTE:  OTG devices may issue remote wakeup (or SRP) even
when
1608         * we don't explicitly enable it here.
1609         */
1610        if (udev->do_remote_wakeup) {
1611                status = usb_control_msg(udev, usb_sndctrlpipe(udev,
0),
1612                                         USB_REQ_SET_FEATURE,
USB_RECIP_DEVICE,
1613                                USB_DEVICE_REMOTE_WAKEUP, 0,
1614                                NULL, 0,
1615                                USB_CTRL_SET_TIMEOUT);
1616                if (status)
1617                        dev_dbg(&udev->dev,
1618                                "won't remote wakeup, status %d\n",
1619                                status);
1620        }
1621
1622        /* see 7.1.7.6 */
1623                        status = set_port_feature(hub->hdev, port1,
USB_PORT_FEAT_SUSPEND);
1624        if (status) {
1625                dev_dbg(hub->intfdev,
1626                        "can't suspend port %d, status %d\n",
1627                        port1, status);
1628                /* paranoia:  "should not happen" */
1629                (void) usb_control_msg(udev, usb_sndctrlpipe(udev, 0),
1630                                       USB_REQ_CLEAR_FEATURE,
USB_RECIP_DEVICE,
1631                                USB_DEVICE_REMOTE_WAKEUP, 0,
1632                                NULL, 0,
1633                                USB_CTRL_SET_TIMEOUT);
1634        } else {
1635                /* device has up to 10 msec to fully suspend */
1636                dev_dbg(&udev->dev, "usb %ssuspend\n",
1637                                udev->auto_pm ? "auto-" : "");
1638                usb_set_device_state(udev, USB_STATE_SUSPENDED);
1639                msleep(10);
1640        }
1641        return status;
1642 }
```

其实之前我们见到过 do_remote_wakeup,不过没有讲,现在不得不讲了.Linux 中,在 usb 系统里关于电源管理的部分已经做的不错了,这主要是因为 usb spec 本身就对 usb 设备做了这方面的规定,即 usb 设备天生就应该支持电源管理,在 usb spec 2.0 中,在第七章讲述电学特性的时候,专门有 7.17.6 和 7.1.7.7 两节介绍了 usb 设备的 Suspend 和 Resume.这其中,Suspend 还包括两种,一种是全局的,叫做 global suspend,另一种叫做选择性的,即可以选择单个的端口进行挂起,这叫 selective suspend.而咱们这里的的这个 hub_port_suspend 所执行的当然就是所谓的选择性挂起了,因为它针对的就是某个端口,而不是整个 hub.而我们现在要说的是 Remote Wakeup,从硬件角度来说,usb 设备定义了一个叫做 Remote Wakeup 的特性,所谓 Remote Wakeup 指的是设备可以发送一个信号,把自己唤醒,当然实际上唤醒的是总线,或者说最后的反应是唤醒主机.最简单的例子就是 usb 键盘,你半天不碰计算机可能大家都睡了,可是突然间你按一下某个键,可能就把大家都给唤醒了,因为你实际上是发送了一个硬件信号.再比如 hub,可能一开始是睡眠的,但如果 hub port 上有设备插入或者拔出,那么基本上就会唤醒 hub.

而一个设备是否具有 Remote Wakeup 这种特性,我们前面在设备的配置描述符里就已经说过,配置描述符中的 bmAttributes 就是标志着设备是否支持 Remote Wakeup.

比如下面是我执行 lsusb –v 命令看到的输出信息中的一部分,这是一个键盘/鼠标的结合体,

```
Bus 003 Device 002: ID 0624:0294 Avocent Corp.
Device Descriptor:
  bLength                18
  bDescriptorType         1
  bcdUSB                1.10
  bDeviceClass            0 (Defined at Interface level)
  bDeviceSubClass         0
  bDeviceProtocol         0
  bMaxPacketSize0         8
  idVendor           0x0624 Avocent Corp.
  idProduct          0x0294
  bcdDevice             1.00
  iManufacturer           1 Avocent
  iProduct                2 Dell 03R874
  iSerial                 0
  bNumConfigurations      1
  Configuration Descriptor:
    bLength                 9
    bDescriptorType         2
    wTotalLength           59
    bNumInterfaces          2
    bConfigurationValue     1
    iConfiguration          4 HID Keyboard / Mouse
    bmAttributes         0xa0
      (Bus Powered)
      Remote Wakeup
    MaxPower              100mA
```

```
Interface Descriptor:
  bLength                 9
  bDescriptorType         4
  bInterfaceNumber        0
  bAlternateSetting       0
  bNumEndpoints           1
  bInterfaceClass         3 Human Interface Devices
  bInterfaceSubClass      1 Boot Interface Subclass
  bInterfaceProtocol      1 Keyboard
  iInterface              5 EP1 Interrupt
    HID Device Descriptor:
      bLength                 9
      bDescriptorType        33
      bcdHID                1.10
      bCountryCode            33 US
      bNumDescriptors          1
      bDescriptorType         34 Report
      wDescriptorLength       64
     Report Descriptors:
       ** UNAVAILABLE **
  Endpoint Descriptor:
    bLength                 7
    bDescriptorType         5
    bEndpointAddress     0x81  EP 1 IN
    bmAttributes            3
      Transfer Type            Interrupt
      Synch Type               None
      Usage Type               Data
    wMaxPacketSize     0x0008  1x 8 bytes
    bInterval               10
```

我们可以看到 Configuration Descriptor 那一段有一个 Remote Wakeup.你在自己电脑上执行一下 lsusb –v 命令,你会发现很多设备的那一段并没有这么一个 Remote Wakeup,只有具有这种特性的才会在这里显示出来.很显然你会发现你的U盘是不具有这个特性的,因为 usb mass storage 协议里也没有定义这方面的特性.

举个例子吧,主持人里,我最喜欢董卿.然而,在上海滩这个地方,一直流传着一个美丽的传说,说央视的当家花旦董卿,原来是在上海,但是后来一步一步睡了上去.我且不说这种说法是否属实,即便真是,那也是人家天生丽质,你有能耐你换芙蓉睡去,看能睡上去不?所以说 Remote Wakeup 这东西,是设备的特性.不是每个设备都具备的.

当然有这种特性也并不意味着这种特性就是 enable 的,因为 usb spec 2.0 里 9.1.1.6 中有这么一句话,If a USB device is capable of remote wakeup signaling, the device must support the ability of the host to enable and disable this capability.即从软件的角度来说,我们可以 enable 这种特性,也可以 disable 这种特性.这道理很简单,董卿虽然天生丽质,但是

她可以选择睡也可以选择不睡.她有这种权利.即便那些做小姐的也有这种权利,一代烈女潘金莲说过:骚归骚,骚有骚的贞操;贱归贱,贱有贱的尊严.

那么对于先天性具有这种特性的设备,如何 enable 这种特性?

usb spec 2.0 中 9.4.5 中说的很好,当我们向一个设备发送 GetStatus()的请求时,其返回值中的 D1 就是表征此时此刻这种能力是否被 enable 了,默认情况 D1 应该是 0,表示 disabled,而如果 D1 被设置成了 1,那么就表示这种特性被 enable 了.如何设置呢?SetFeature()请求,请求的是 DEVICE_REMOTE_WAKEUP.用代码来说话,那就是咱们这里的 1611 行,这样就算是 enable 了 Remote Wakeup,如果你要 disable 掉的话,只要把 SetFeature 换成 ClearFeature 即可.DEVICE_REMOTE_WAKEUP 被称为标准的 Feature 选择器.如图所示:

Table 9-6. Standard Feature Selectors

Feature Selector	Recipient	Value
DEVICE_REMOTE_WAKEUP	Device	1
ENDPOINT_HALT	Endpoint	0
TEST_MODE	Device	2

而 do_remote_wakeup 作为 struct usb_device 结构体中的一个成员,其默认值为 1.只是刚才在 usb_suspend_device 函数中,我们判断如果设备没有和 driver 绑定,就先把其 do_remote_wakeup 设置为 0,理由很简单,没有驱动的话,就没必要找麻烦了,还是那句话,男人,简单就好.

紧接着 1623 行, set_port_feature,这次设置的是 USB_PORT_FEAT_SUSPEND,这个宏对应的 usb spec 2.0 中的 PORT_SUSPEND,设置了这个 feature 就意味着停止这个端口上的总线交通,也因此就意味着该端口连的设备进入了 suspend 状态.而这也正是我们的最终目标,这之后我们看到 1638 行就调用 usb_set_device_state 把设备的状态设置为 USB_STATE_SUSPENDED.当然,也要注意到,如果设置 PORT_SUSPEND 失败了的话,我们就将在 1629 行发送 ClearFeature 把 Remote Wakeup 的特性给清除掉.因为已经没有必要了,没有挂起就没有唤醒,没有共产党就没有新中国,没有新中国就没有性生活.

Ok,不小心把这个 suspend 的流程走了一遍,不过你一定还要问,为何 drivers/usb/core/hub.c 中的那个 hub_suspend 还没讲?于是现在来说 hub driver,hub_suspend 被赋值给了 hub_driver 中的 suspend 成员,而 hub driver 是一个 interface driver,所以实际上 hub_suspend 将会在当初那个 usb_suspend_interface 中被调用,没错就是 866 那行,status = driver->suspend(intf, msg);

于是我们就来具体看看 hub_suspend.

```
1919 static int hub_suspend(struct usb_interface *intf, pm_message_t msg)
1920 {
1921         struct usb_hub          *hub = usb_get_intfdata (intf);
```

```
1922        struct usb_device       *hdev = hub->hdev;
1923        unsigned                port1;
1924        int                     status = 0;
1925
1926        /* fail if children aren't already suspended */
1927        for (port1 = 1; port1 <= hdev->maxchild; port1++) {
1928                struct usb_device       *udev;
1929
1930                udev = hdev->children [port1-1];
1931                if (udev && msg.event == PM_EVENT_SUSPEND &&
1932 #ifdef  CONFIG_USB_SUSPEND
1933                                udev->state != USB_STATE_SUSPENDED
1934 #else
1935                                udev->dev.power.power_state.event
1936                                        == PM_EVENT_ON
1937 #endif
1938                                ) {
1939                        if (!hdev->auto_pm)
1940                                dev_dbg(&intf->dev, "port %d nyet
suspended\n",
1941                                                port1);
1942                        return -EBUSY;
1943                }
1944        }
1945
1946        dev_dbg(&intf->dev, "%s\n", __FUNCTION__);
1947
1948        /* stop khubd and related activity */
1949        hub_quiesce(hub);
1950
1951        /* "global suspend" of the downstream HC-to-USB interface */
1952        if (!hdev->parent) {
1953                status = hcd_bus_suspend(hdev->bus);
1954                if (status != 0) {
1955                        dev_dbg(&hdev->dev, "'global' suspend %d\n",
status);
1956                        hub_activate(hub);
1957                }
1958        }
1959        return status;
1960 }
```

其实也没做什么.不过我们可以看到msg在整个suspend的情节里是代代相传,每个函数都把它当参数.

1927 至 1944 这一段循环的目的就是判断是否有任何一个 Hub 的子设备尚未挂起,我们说过只有底层的劳苦大众能够安详的睡眠,上层的公仆们才好意思安心去休息.当然你别美,写代码的无非是描绘一种乌托邦式的世界,这是他们的理想,仅此而已.关于 CONFIG_USB_SUSPEND,我们在 drivers/usb/core/Kconfig 中可以看到,

```
74 config USB_SUSPEND
75         bool "USB selective suspend/resume and wakeup (EXPERIMENTAL)"
76         depends on USB && PM && EXPERIMENTAL
77         help
78           If you say Y here, you can use driver calls or the sysfs
79           "power/state" file to suspend or resume individual USB
80           peripherals.
81
82           Also, USB "remote wakeup" signaling is supported, whereby some
83           USB devices (like keyboards and network adapters) can wake up
84           their parent hub.  That wakeup cascades up the USB tree, and
85           could wake the system from states like suspend-to-RAM.
86
87           If you are unsure about this, say N here.
```

毫无疑问当我们想在 usb 子系统里支持 suspend 我们完全可以理直气壮的打开这个编译开关.前面我们说过,dev.power.power_state.event 如果等于 PM_EVENT_ON,就相当于向世界宣布,本设备当前并不是挂起状态.而这里的 udev->state 不等于 USB_STATE_SUSPENDED 也是表征同样的含义,不过你需要注意,USB_STATE_SUSPENDED 将由 usb_set_device_state 来设置,而你不难发现,这个世界上一共有两个函数会调用这个函数来设置这个状态,它们就是刚才说过的__usb_port_suspend 和 hub_port_suspend(),其中,后者还是被前者调用的,即 __usb_port_suspend 调用 hub_port_suspend, 而 __usb_port_suspend 是被 usb_port_suspend 调用,我们继续跟踪的话就会发现,如果你没有打开 CONFIG_USB_SUSPEND,usb_port_suspend 只是一个空函数,啥也不做.

```
1886 #else   /* CONFIG_USB_SUSPEND */
1887
1888 /* When CONFIG_USB_SUSPEND isn't set, we never suspend or resume any ports. */
1889
1890 int usb_port_suspend(struct usb_device *udev)
1891 {
1892         return 0;
1893 }
```

所以, 这里判断 USB_STATE_SUSPENDED 之前要先判断编译开关.另一个问题,这里因为 USB_STATE_SUSPENDED 是 usb 这部分代码定义的宏,而 PM_EVENT_ON 毕竟是 PM core 那边定义的宏,所以,我们应该尽量使用自己的这个宏.实在不行才去使用外部的资源.

auto_pm 是用来设置自动挂起的,如果它为 1,表明这次挂起是自动挂起,而不是系统级的挂起.以咱们这个上下文来看,由于咱们在 usb_external_suspend_device 中设置了 auto_pm 为 0,所以这里就直接返回了.

为何 auto_pm 为 0 就返回,否则就不返回,稍后我们看到 autosuspend/autoresume 那边的代码就明白了,我们现在这里的思路是,没什么特别的理由,那么子设备如果没有睡眠,那么父设备的 suspend 就会失败,但是如果这是一次 autosuspend,那么就不会失败.因为 autosuspend 会有专门的方法来处理这种情况.后面会看到.

Ok,千呼万唤始出来的 hub_quiesce()函数.很久很久以前我们就见到过那个 hub->quiesce 了,我们曾多次判断过它是否为零,在 hub_events()中我们判断过,在 hub_irq()中我们判断过,在 led_work()中我们判断过.

```
500 static void hub_quiesce(struct usb_hub *hub)
501 {
502         /* (nonblocking) khubd and related activity won't re-trigger */
503         hub->quiescing = 1;
504         hub->activating = 0;
505
506         /* (blocking) stop khubd and related activity */
507         usb_kill_urb(hub->urb);
508         if (hub->has_indicators)
509                 cancel_delayed_work(&hub->leds);
510         if (hub->has_indicators || hub->tt.hub)
511                 flush_scheduled_work();
512 }
```

看到这么短小精悍的函数,不由得一阵喜悦涌上心头.真的,我的这种喜悦,可以直接和中央电视台播音员海霞的喜悦相比,我的微笑比她更加灿烂.(“我看到大坝周围有很多围观的群众,都带着过年的心情,是这样吗?”在播报淮河水灾的新闻节目当中,在连线王家坝现场的记者时,海霞面带微笑,略有些激动地问道.)

这个函数是唯一一处设置 hub->quiescing 为 1 的地方.它和另一个函数针锋相对,即 hub_activate(),hub_activate()中设置 hub->quiescing 为 0,而设置 hub->activating 为 1.而这个函数恰恰相反,仔细一看你会发现,这两个函数做的事情那几乎是完全相反.那边人家调用 usb_submit_urb()提交一个 urb,这边就给人拆台,调用 usb_kill_urb()来撤掉该 urb,那边人家调用 schedule_delayed_work 建立一个延时工作的函数,这边就调用 cancel_delayed_work 给人家拆了,flush_scheduled_work()通常在 cancel_delayed_work 后面被调用,这个函数会使等待队列中所有的任务都被执行.

1952 到 1958 行是专门针对 Root Hub 的,因为通常 Host Controller Driver 也会提供自己的 suspend 函数,所以如果挂起操作已经上升到了 Root Hub 这一层,就应该调用 hcd 的 suspend 函数.即 hcd_bus_suspend.1954 行,如果挂起失败了,那么就别挂起,还是调用 hub_activate()重新激活,苦海无涯,回头是岸.

梦醒时分

爱情就像拔河比赛,如果一方先放手,另一方就会受伤.

只可惜说出这句话的梁咏琪,最终还是放开了与郑伊健相牵的手.

suspend 和 resume 也是这样,如果你不调用 suspend,那么你永远也不需要调用 resume,它们就这样青梅竹马的存在于这个世界上,过着世外桃源般的日子.但是如果你不小心调用了 suspend 让设备睡眠,那么你就必然需要在将来某个时刻调用 resume 来唤醒设备.

看完了 suspend 我们来看 resume,变量 usb_bus_type 中的成员 resume 被赋值为 usb_resume,和 usb_suspend 对应.来自 drivers/usb/core/driver.c:

```
1504 static int usb_resume(struct device *dev)
1505 {
1506         struct usb_device       *udev;
1507
1508         if (!is_usb_device(dev))        /* Ignore PM for interfaces */
1509                 return 0;
1510         udev = to_usb_device(dev);
1511         if (udev->autoresume_disabled)
1512                 return -EPERM;
1513         return usb_external_resume_device(udev);
1514 }
```

看过了 usb_suspend 再来看这个 usb_resume 就显得很简单了,两个函数基本能体现一种对称美.autoresume_disabled 是 struct usb_device 中的一个成员,即,我们提供给用户一种选择,让用户可以自己来 disable 掉设备的 autoresume,一旦 disable 掉了,就意味着设备是不会唤醒了,所以这里直接返回错误码.

接着,usb_external_resume_device,

```
1469 /**
1470  * usb_external_resume_device - external resume of a USB device and its interfaces
1471  * @udev: the usb_device to resume
1472  *
1473  * This routine handles external resume requests: ones not generated
1474  * internally by a USB driver (autoresume) but rather coming from the user
1475  * (via sysfs), the PM core (system resume), or the device itself (remote
1476  * wakeup).  @udev's usage counter is unaffected.
1477  *
1478  * The caller must hold @udev's device lock.
```

```
1479  */
1480 int usb_external_resume_device(struct usb_device *udev)
1481 {
1482         int     status;
1483
1484         usb_pm_lock(udev);
1485         udev->auto_pm = 0;
1486         status = usb_resume_both(udev);
1487         udev->last_busy = jiffies;
1488         usb_pm_unlock(udev);
1489
1490            /* Now that the device is awake, we can start trying to
autosuspend
1491          * it again. */
1492         if (status == 0)
1493                 usb_try_autosuspend_device(udev);
1494         return status;
1495 }
```

也不干别的,设置好 udev->auto_pm 为 0,然后调用 usb_resume_both.再然后调用 usb_try_autosuspend_device,关于autosuspend/autoresume的部分我们稍候会单独讲.现在先来看 usb_resume_both.

```
1083 /**
1084  * usb_resume_both - resume a USB device and its interfaces
1085  * @udev: the usb_device to resume
1086  *
1087  * This is the central routine for resuming USB devices.  It calls the
1088  * the resume method for @udev and then calls the resume methods for all
1089  * the interface drivers in @udev.
1090  *
1091  * Before starting the resume, the routine calls itself recursively for
1092  * the parent device of @udev, thereby propagating the change up the
device
1093  * tree and assuring that @udev will be able to resume.  If the parent is
1094  * unable to resume successfully, the routine fails.
1095  *
1096  * The resume method calls are subject to mutual exclusion under control
1097  * of @udev's pm_mutex.  Many of these calls are also under the
protection
1098  * of @udev's device lock (including all requests originating outside the
1099  * USB subsystem), but autoresume requests generated by a child device
or
1100  * interface driver may not be.  Usbcore will insure that the method calls
```

```
1101  * do not arrive during bind, unbind, or reset operations.  However, drivers
1102  * must be prepared to handle resume calls arriving at unpredictable times.
1103  * The only way to block such calls is to do an autoresume (preventing
1104  * other autoresumes) while holding @udev's device lock (preventing outside
1105  * resumes).
1106  *
1107  * The caller must hold @udev->pm_mutex.
1108  *
1109  * This routine can run only in process context.
1110  */
1111 static int usb_resume_both(struct usb_device *udev)
1112 {
1113         int                     status = 0;
1114         int                     i;
1115         struct usb_interface    *intf;
1116         struct usb_device       *parent = udev->parent;
1117
1118         cancel_delayed_work(&udev->autosuspend);
1119         if (udev->state == USB_STATE_NOTATTACHED) {
1120                 status = -ENODEV;
1121                 goto done;
1122         }
1123
1124         /* Propagate the resume up the tree, if necessary */
1125         if (udev->state == USB_STATE_SUSPENDED) {
1126                 if (udev->auto_pm && udev->autoresume_disabled) {
1127                         status = -EPERM;
1128                         goto done;
1129                 }
1130                 if (parent) {
1131                         status = usb_autoresume_device(parent);
1132                         if (status == 0) {
1133                                 status = usb_resume_device(udev);
1134                                 if (status) {
1135
usb_autosuspend_device(parent);
1136
1137                                                         /* It's possible
usb_resume_device()
1138                                                  * failed after the port was
```

```
1139                                    * unsuspended, causing udev to
be
1140                                      * logically disconnected.  We
don't
1141                                       * want usb_disconnect() to
autosuspend
1142                                    * the parent again, so tell it that
1143                                    * udev disconnected while still
1144                                    * suspended. */
1145                                   if (udev->state ==
1146
USB_STATE_NOTATTACHED)
1147                                           udev->discon_suspended
= 1;
1148                                }
1149                        }
1150                } else {
1151
1152                                /* We can't progagate beyond the USB
subsystem,
1153                         * so if a root hub's controller is suspended
1154                         * then we're stuck. */
1155                                                                    if
(udev->dev.parent->power.power_state.event !=
1156                                           PM_EVENT_ON)
1157                                status = -EHOSTUNREACH;
1158                        else
1159                                status = usb_resume_device(udev);
1160                }
1161        } else {
1162
1163                                    /*  Needed  only  for  setting
udev->dev.power.power_state.event
1164                 * and for possible debugging message. */
1165                status = usb_resume_device(udev);
1166        }
1167
1168        if (status == 0 && udev->actconfig) {
1169                for (i = 0; i < udev->actconfig->desc.bNumInterfaces;
i++) {
1170                        intf = udev->actconfig->interface[i];
1171                        usb_resume_interface(intf);
1172                }
1173        }
```

```
1174
1175  done:
1176          // dev_dbg(&udev->dev, "%s: status %d\n", __FUNCTION__,
status);
1177          return status;
1178 }
```

和 usb_suspend_both 的结构比较类似,不过这里是先针对 device,再针对 interface,而 usb_suspend_both 那儿是先针对 interface,再针对 device.我们先来看一下 udev->autosuspend.我们刚刚才看到过 cancel_delayed_work,这里又出现了一次.

struct usb_device 结构体有一个成员,struct delayed_work autosuspend,要明白它,必须先明白另一个家伙,ksuspend_usb_wq.在 drivers/usb/core/usb.c 中:

```
51 /* Workqueue for autosuspend and for remote wakeup of root hubs */
52 struct workqueue_struct *ksuspend_usb_wq;
```

之前咱们在讲 hub->leds 的时候提到过 struct workqueue_struct 代表一个工作队列,不过作为 hub->leds,咱们没有单独建立一个工作队列,而是使用默认的公共队列,但是这里咱们需要单独建立自己的队列.很显然,hub->leds 代表着与指示灯相关的代码,其地位是很低下的,不可能受到足够的重视,而这里这个工作队列代表着整个 usb 子系统里与电源管理非常相关的一部分代码的利益,当然会被受到重视.

不信的话,咱们可以再一次看一下 usb 子系统初始化的代码,即 usb_init 函数,其第一个重要的函数就是 ksuspend_usb_init(),drivers/usb/core/usb.c 中:

```
200 #ifdef  CONFIG_PM
201
202 static int ksuspend_usb_init(void)
203 {
204         /* This workqueue is supposed to be both freezable and
205          * singlethreaded.  Its job doesn't justify running on more
206          * than one CPU.
207          */
208                                                 ksuspend_usb_wq     =
create_freezeable_workqueue("ksuspend_usbd");
209         if (!ksuspend_usb_wq)
210                 return -ENOMEM;
211         return 0;
212 }
213
214 static void ksuspend_usb_cleanup(void)
215 {
216         destroy_workqueue(ksuspend_usb_wq);
217 }
```

```
218
219 #else
220
221 #define ksuspend_usb_init()      0
222 #define ksuspend_usb_cleanup()   do {} while (0)
223
224 #endif   /* CONFIG_PM */
```

如果你没有打开CONFIG_PM这个编译开关,当然就什么也不会发生,这俩函数也就是空函数,如果打开了,那么 ksuspend_usb_init 在 usb_init 中被调用,而 ksuspend_usb_cleanup 反其道而行之,在 usb_exit 中被调用.

我们看到,这两个函数都非常短,但这并不要紧.生命的长短并不是最重要的,而是精彩与否.利群广告词:人生就像是一场旅行,不在乎目的地,在乎的是沿途的风景和看风景的心情.怎样理解这段广告词?一个吸烟的同学说,可不可以这样认为,吸烟就像是男人的春药,不在乎浓缩生命的长河,在乎的是吸烟时的沉醉和沉醉时的快乐.

没错,ksuspend_usb_init 虽然超级短,但是它却做了一件相当有意义的事情,那就是调用create_freezeable_workqueue(),这其实就是创建一个工作队列,函数的参数就是这个工作队列的名字,即 ksuspend_usbd,而函数的返回值就是工作队列结构体指针,即 struct workqueue_struct 指针,然后赋值给了 ksuspend_usb_wq.所以我们接下来就需要和ksuspend_usb_wq 打交道了.我们所要知道的是,如果我们要把一个任务加入到工作队列中来,我们可以调用 queue_work 或者 queue_delayed_work.在 2.6.22.1 的内核中,往这个工作队列中加工作的地方只有两处,一个是 drivers/usb/core/driver.c 中的autosuspend_check() 函数内部,调用的是 queue_delayed_work, 一个是drivers/usb/core/hcd.c 中,调用的是 queue_work.autosuspend_check 我们后面会讲,现在把与 ksuspend_usb_wq 相关的两行贴出来,先睹为快.

```
976                                 queue_delayed_work(ksuspend_usb_wq,
&udev->autosuspend,
977                                         suspend_time - jiffies);
```

这里第三个参数 suspend_time-jiffies,表明一个延时的时间,即至少经过这么多时间之后,这个任务才可以真正执行.这就和我们前面见过的那个 schedule_delayed_work()函数类似.

而正是这里让我们看到了 udev->autosuspend 和 ksuspend_usb_wq 之间的关系,即后者代表一个工作队列,而前者代表一个工作,这里的做法就是把 autosuspend 给加入到了ksuspend_usb_wq 这个队列里,并且在经过一段延时之后执行这个工作.工作队列中的任务由相关的工作线程执行,可能是在一个无法预期的时间(取决于负载,中断等等),或者是在一段延迟之后.

于是你该问了,udev->autosuspend 是一个 struct delayed_work 结构体,那么它所对应的那个函数是谁?其实我们见过,不过也许你已经忘记了,还记得八大函数的第一个么 ,usb_alloc_dev, 当时就有这么一行,INIT_DELAYED_WORK(&dev->autosuspend,usb_autosuspend_work),所以说,每

一个 usb 设备在它刚问世的时候就已经和一个叫做 usb_autosuspend_work 给捆绑了起来,即它还少不更事的时候就已经和 usb_autosuspend_work()函数签了这么一个卖身契,因此,不管你是 usb 鼠标还是 usb 键盘,或者是 usb mass storage,总之你都和 usb_autosuspend_work 得发生关系.

至此,我们就很好理解 usb_resume_both 函数中 cancel_delayed_work 那行的意思了.我们且不管 autosuspend 和一般的 suspend 有什么区别,一个很简单的道理,既然要 resume,那当然就不要 suspend 了.cancel 了一个设备的 autosuspend 的 work,自然它就不会再自动挂起,而如果你以后要让它能够自动挂起,你可以再次调用 queue_delayed_work,正如在 autosuspend_check 中做的那样.具体代码我们后面讲 autosuspend 了再看.

继续在 usb_resume_both 中往下看,刚才我们设置了 auto_pm 为 0,所以这里 1126 这个 if 内部不会执行.

1130 行,如果不是 Root Hub,那么调用针对父设备调用 usb_autoresume_device,还是我们一直强调的那个道理,挂起的时候要从下至上,而唤醒的时候要自上而下.一个自来水管系统,如果上面没水,下面开关全打开也没有任何意义.

上面醒来了,才调用 usb_resume_device 唤醒下面的当前这个 device.如果当前设备的唤醒失败了,那么调用 usb_autosuspend_device()来把刚才做的事情取消掉,道理很简单,本来咱们的目的就是为了唤醒当前设备,为此我们先唤醒了上层的设备,结果上层的设备唤醒了,但是咱们自己却没有唤醒,那咱们所作的就是无用功了,所以还是把刚刚唤醒的上层设备给催眠吧.

虽然我们还没有讲 autosuspend/autoresume,但是凭一个男人的直觉,我们基本上能够感觉出,usb_autosuspend_device 和 usb_autoresume_device 这两个函数可以很好的处理 usb 设备树,即他们不会仅仅对付一个设备,而是会很自然的沿着 usb 设备树去往上走或者往下走,从而保证咱们刚才说的那个挂起时从下至上,唤醒时从上而下.其实,在 usb_suspend_both 中我们还剩下一个函数没有提,它正是 usb_autosuspend_device(),而且是针对父设备的,如果你有耐心,你会发现 usb_autosuspend_device 会调用 usb_autopm_do_device(),而后者又会调用 usb_suspend_both,这样一层一层往上走,其效果就像一只蜗牛,一步一步往上爬,在最高点乘着叶片往前飞,任风吹干流过的泪和汗.而这里的 usb_autoresume_device 的原理恰好相反,它也会调用 usb_autopm_do_device(),而后者这时候又会调用 usb_resume_both,于是就会一层一层往下走,效果就相当于我们以前往的那个游戏—“是男人就下一百层”.也正是因为这种你调用我我调用你的复杂关系,我们才决定先不去深入看 autosuspend 相关的函数,等我们看完了非 autosuspend 的函数再去看就会很容易理解.

Ok,让我们来看 usb_resume_device(),

```
822 /* Caller has locked udev's pm_mutex */
823 static int usb_resume_device(struct usb_device *udev)
824 {
825         struct usb_device_driver        *udriver;
826         int                             status = 0;
827
828         if (udev->state == USB_STATE_NOTATTACHED ||
```

```
    829                         udev->state != USB_STATE_SUSPENDED)
    830                 goto done;
    831
    832         /* Can't resume it if it doesn't have a driver. */
    833         if (udev->dev.driver == NULL) {
    834                 status = -ENOTCONN;
    835                 goto done;
    836         }
    837
    838         udriver = to_usb_device_driver(udev->dev.driver);
    839         status = udriver->resume(udev);
    840
    841 done:
    842          // dev_dbg(&udev->dev, "%s: status %d\n", __FUNCTION__,
status);
    843         if (status == 0) {
    844                 udev->autoresume_disabled = 0;
    845                 udev->dev.power.power_state.event = PM_EVENT_ON;
    846         }
    847         return status;
    848 }
```

这种似曾相识的感觉是不言而喻的,和 usb_suspend_device 那是相当的对称啊!最重要的当然是 839 行,调用属于设备驱动的 resume 函数.不过,至少到 2.6.22.1 的内核为止,这个世界上总共只有一个 struct usb_device_driver 的结构体变量,它就是 struct usb_device_driver usb_generic_driver,而实际上你会发现你的每个设备默认情况下都会和这个 usb_generic_driver 相绑定,除非你自己定义了自己的 struct usb_device_driver 结构体,不过至少在标准内核中,暂时还没有人这么干,当然以后也许会有.否则 struct usb_device_driver 这个结构体就太浪费了点.关于 usb_generic_driver,你可以在 sysfs 下看到效果,比如:

```
localhost:/usr/src/linux-2.6.22.1/drivers/usb/core # ls /sys/bus/usb/drivers
hub  usb  usb-storage  usbfs
```

所有的 usb 驱动程序都会在这里有一个对应的目录,其中与 usb_generic_driver 对应的就是那个 usb 目录.这是因为:

```
    210 struct usb_device_driver usb_generic_driver = {
    211         .name = "usb",
    212         .probe = generic_probe,
    213         .disconnect = generic_disconnect,
    214 #ifdef  CONFIG_PM
    215         .suspend = generic_suspend,
    216         .resume = generic_resume,
    217 #endif
```

```
218         .supports_autosuspend = 1,
219 };
```

这里的 name 就对应了在/sys/bus/usb/drivers/下面的那个子目录名称.现在的内核的处理方式是,凡是有一个新的设备被探测到,就先把它的 struct usb_device 和这个 generic driver 相绑定,即首先被调用的 generic_probe,然后才会根据每一个具体的 interface 去绑定属于具体 interface 的驱动程序,去调用具体的那个 interface 对应的 driver 的 probe 函数,比如 storage_probe. 因此你会发现,不管你插入任何 usb 设备,你都会在/sys/bus/usb/drivers/usb/目录下面发现多出一个文件来,比如:

```
localhost:/usr/src/linux-2.6.22.1/drivers/usb/core # ls /sys/bus/usb/drivers/usb
1-1  bind  module  unbind  usb1  usb2  usb3  usb4  usb5
```

而如果你插入的是 usb-storage,那么接下来在/sys/bus/usb/drivers/usb-storage/下面也将多出一个对应的文件来,

```
localhost:/usr/src/linux-2.6.22.1/drivers/usb/core # ls
/sys/bus/usb/drivers/usb-storage/
1-1:1.0  bind  module  new_id  unbind
```

而在/sys/bus/usb/devices/目录下面你能看到所有的 usb 设备,

```
localhost:/usr/src/linux-2.6.22.1/drivers/usb/core # ls /sys/bus/usb/devices/
1-0:1.0  1-1  1-1:1.0  2-0:1.0  3-0:1.0  4-0:1.0  5-0:1.0  usb1  usb2  usb3
usb4  usb5
```

贴出这些只是为了给你有一个直观的印象,而我们需要知道的是对于当前的设备来说,其默认情况下所对应的设备驱动级的 suspend 函数就是 generic_suspend,resume 函数就是 generic_resume,所以我们来看一下这两个函数,

```
192 #ifdef  CONFIG_PM
193
194 static int generic_suspend(struct usb_device *udev, pm_message_t msg)
195 {
196         /* USB devices enter SUSPEND state through their hubs, but can
be
197          * marked for FREEZE as soon as their children are already idled.
198          * But those semantics are useless, so we equate the two (sigh).
199          */
200         return usb_port_suspend(udev);
201 }
202
203 static int generic_resume(struct usb_device *udev)
204 {
205         return usb_port_resume(udev);
```

```
 206 }
 207
 208 #endif  /* CONFIG_PM */
```

呵呵,原来也就是调用 usb_port_suspend 和 usb_port_resume 而已.前面我们已经看过 usb_port_suspend 函数,现在我们来看 usb_port_resume,

```
1838 /*
1839  * usb_port_resume - re-activate a suspended usb device's upstream port
1840  * @udev: device to re-activate
1841  * Context: must be able to sleep; device not locked; pm locks held
1842  *
1843  * This will re-activate the suspended device, increasing power usage
1844  * while letting drivers communicate again with its endpoints.
1845  * USB resume explicitly guarantees that the power session between
1846  * the host and the device is the same as it was when the device
1847  * suspended.
1848  *
1849  * Returns 0 on success, else negative errno.
1850  */
1851 int usb_port_resume(struct usb_device *udev)
1852 {
1853         int      status;
1854
1855         /* we change the device's upstream USB link,
1856          * but root hubs have no upstream USB link.
1857          */
1858         if (udev->parent) {
1859                 // NOTE this fails if parent is also suspended...
1860                 status = hub_port_resume(hdev_to_hub(udev->parent),
1861                                 udev->portnum, udev);
1862         } else {
1863                 dev_dbg(&udev->dev, "usb %sresume\n",
1864                                 udev->auto_pm ? "auto-" : "");
1865                 status = finish_port_resume(udev);
1866         }
1867         if (status < 0)
1868                    dev_dbg(&udev->dev, "can't resume, status %d\n",
status);
1869         return status;
1870 }
```

结构很清晰,对于非 Root Hub,调用 hub_port_resume,对于 Root Hub,调用 finish_port_resume,先看前者再看后者.

```
1770 static int
1771 hub_port_resume(struct usb_hub *hub, int port1, struct usb_device
*udev)
1772 {
1773         int     status;
1774         u16     portchange, portstatus;
1775
1776         /* Skip the initial Clear-Suspend step for a remote wakeup */
1777         status = hub_port_status(hub, port1, &portstatus, &portchange);
1778         if (status == 0 && !(portstatus & USB_PORT_STAT_SUSPEND))
1779                 goto SuspendCleared;
1780
1781         // dev_dbg(hub->intfdev, "resume port %d\n", port1);
1782
1783         set_bit(port1, hub->busy_bits);
1784
1785         /* see 7.1.7.7; affects power usage, but not budgeting */
1786         status = clear_port_feature(hub->hdev,
1787                         port1, USB_PORT_FEAT_SUSPEND);
1788         if (status) {
1789                 dev_dbg(hub->intfdev,
1790                         "can't resume port %d, status %d\n",
1791                         port1, status);
1792         } else {
1793                 /* drive resume for at least 20 msec */
1794                 if (udev)
1795                         dev_dbg(&udev->dev, "usb %sresume\n",
1796                                         udev->auto_pm ? "auto-" : "");
1797                 msleep(25);
1798
1799 #define LIVE_FLAGS      ( USB_PORT_STAT_POWER \
1800                         | USB_PORT_STAT_ENABLE \
1801                         | USB_PORT_STAT_CONNECTION)
1802
1803                 /* Virtual root hubs can trigger on GET_PORT_STATUS to
1804                  * stop resume signaling.  Then finish the resume
1805                  * sequence.
1806                  */
1807                     status = hub_port_status(hub, port1, &portstatus,
&portchange);
1808 SuspendCleared:
1809                 if (status < 0
1810                                         || (portstatus & LIVE_FLAGS) !=
LIVE_FLAGS
```

```
1811                                                    || (portstatus &
USB_PORT_STAT_SUSPEND) != 0
1812                                    ) {
1813                            dev_dbg(hub->intfdev,
1814                                    "port %d status %04x.%04x after resume,
%d\n",
1815                                    port1, portchange, portstatus, status);
1816                            if (status >= 0)
1817                                    status = -ENODEV;
1818                    } else {
1819                            if (portchange & USB_PORT_STAT_C_SUSPEND)
1820                                    clear_port_feature(hub->hdev, port1,
1821
USB_PORT_FEAT_C_SUSPEND);
1822                            /* TRSMRCY = 10 msec */
1823                            msleep(10);
1824                            if (udev)
1825                                    status = finish_port_resume(udev);
1826                    }
1827            }
1828            if (status < 0)
1829                    hub_port_logical_disconnect(hub, port1);
1830
1831            clear_bit(port1, hub->busy_bits);
1832            if (!hub->hdev->parent && !hub->busy_bits[0])
1833                    usb_enable_root_hub_irq(hub->hdev->bus);
1834
1835            return status;
1836 }
```

看起来还挺复杂.但目的很明确,唤醒这个 hub 端口.首先是察看该 port 是否已经是 resume 了.然后最直观最实在的代码就是 1786 行那个 clear_port_feature,这行代码很显然是和 hub_port_suspend 中那行 set_port_feature 遥相呼应的.你给 hub port 设置了 USB_PORT_FEAT_SUSPEND,我就给你清掉,偏要跟你对着干.

如果成功了,status 为 0,进入 1792 行这个 else,睡眠 25ms,usb spec 2.0 规定,resume 信号应该维持至少 20 毫秒才能有效,这个时间被称为 T_{DRSMDN}.这个道理很显然,我们说技术是无国界的,今天早上在城铁上看见一个女的用小刀割伤了一个男的手,说那哥们对人家性骚扰,那么这道理是一样的,只有性骚扰持续了一段时间,那女的才能感觉到,否则你比如说那个男的技术足够好,那手从人家某部位像闪电一样过去,那估计双方都没有任何感觉.

1807 至 1826 行这里是一种特殊情况,你这里想得很周到,睡眠 25ms,可是 Root Hub 可能会把你的 resume 信号给 stop 掉,只要它在这期间发送了 GET_PORT_STATUS 请求.所以这里就再次读取端口的状态,如果这个端口对应的 USB_PORT_STAT_POWER/USB_PORT_STAT_ENABLE/USB_PORT_STAT_CONNECT

ION 中有一位为 0,那么说明出错了,也就不用继续折腾了.而如果 USB_PORT_STAT_SUSPEND 这一位又被设置了,那么咱的猜测也就对了.这种情况咱们这里先把 status 设置为-ENODEV.

如果不是以上情况,那么 1818 行进去.如果 SUSPEND 位确实有变化,说明 resume 操作基本上达到了目的,那么先清掉这一位.睡眠 10ms,这个 10 毫秒被称为 T_{RSMRCY},或称为 resume 恢复时间(resume recovery time),这个道理也很简单,通常我睡一觉醒来之后必然不是马上很清醒,肯定还需要一段时间恢复一下,特别是前一天晚上玩游戏玩到半夜,第二天又有喜欢点名的变态老师的课,那么我基本上是匆匆从南区赶到本部教室,但意识却还未清醒,不过我和设备不同的是,设备恢复了之后就可以工作了,而我赶到教室之后就是接着睡.

```
1709 /*
1710  * If the USB "suspend" state is in use (rather than "global suspend"),
1711  * many devices will be individually taken out of suspend state using
1712  * special" resume" signaling.  These routines kick in shortly after
1713  * hardware resume signaling is finished, either because of selective
1714  * resume (by host) or remote wakeup (by device) ... now see what changed
1715  * in the tree that's rooted at this device.
1716  */
1717 static int finish_port_resume(struct usb_device *udev)
1718 {
1719         int     status;
1720         u16     devstatus;
1721
1722         /* caller owns the udev device lock */
1723         dev_dbg(&udev->dev, "finish resume\n");
1724
1725         /* usb ch9 identifies four variants of SUSPENDED, based on what
1726          * state the device resumes to.  Linux currently won't see the
1727          * first two on the host side; they'd be inside hub_port_init()
1728          * during many timeouts, but khubd can't suspend until later.
1729          */
1730         usb_set_device_state(udev, udev->actconfig
1731                         ? USB_STATE_CONFIGURED
1732                         : USB_STATE_ADDRESS);
1733
1734         /* 10.5.4.5 says be sure devices in the tree are still there.
1735          * For now let's assume the device didn't go crazy on resume,
1736          * and device drivers will know about any resume quirks.
1737          */
1738         status = usb_get_status(udev, USB_RECIP_DEVICE, 0, &devstatus);
1739         if (status >= 0)
1740                 status = (status == 2 ? 0 : -ENODEV);
```

```
1741
1742        if (status)
1743                dev_dbg(&udev->dev,
1744                        "gone after usb resume? status %d\n",
1745                        status);
1746        else if (udev->actconfig) {
1747                le16_to_cpus(&devstatus);
1748                if ((devstatus & (1 << USB_DEVICE_REMOTE_WAKEUP))
1749                                && udev->parent) {
1750                        status = usb_control_msg(udev,
1751                                        usb_sndctrlpipe(udev, 0),
1752                                        USB_REQ_CLEAR_FEATURE,
1753                                                USB_RECIP_DEVICE,
1754                                        USB_DEVICE_REMOTE_WAKEUP,
0,
1755                                        NULL, 0,
1756                                        USB_CTRL_SET_TIMEOUT);
1757                        if (status)
1758                                dev_dbg(&udev->dev, "disable remote "
1759                                        "wakeup, status %d\n", status);
1760                }
1761                status = 0;
1762
1763        } else if (udev->devnum <= 0) {
1764                dev_dbg(&udev->dev, "bogus resume!\n");
1765                status = -EINVAL;
1766        }
1767        return status;
1768 }
```

这个函数其实就是做一些收尾的工作.我们刚才在 usb_port_resume 中看到,对于 Root Hub 就是直接调用这个函数,因为 Root Hub 不存在说接在别的 Hub 口上的说法.

1730 行调用 usb_set_device_state 设置状态, USB_STATE_CONFIGURED 或者是 USB_STATE_ADDRESS,如果配置好了,就记录为前者,如果没有配置好,就记录为后者.

1738 行,如注释所言,spec 10.5.4.5 节建议这么做.以确认设备还在,而不是说在 suspend/resume 的过程中被拔走了.于是调用 usb_get_status,获取设备的状态,返回值是设备返回的数据的长度.按 spec 规定,返回的数据应该是 16 位,即 2 个 bytes,所以 1740 行判断是否为 2.如果为 2,就将 status 置为 0,并开始新的判断.即 1742 行的判断,

1746 行,如果设备配置好了,然后设备又不是 Root Hub,然后设备的 Wakeup 功能是 enabled 的,那么就发送 ClearFeature 请求把这个功能给关掉,因为很显然,当一个设备在醒来的时候就没有必要打开这个功能,只有在将要睡去的时候才有必要打开,就好比我们起床以后就会把闹钟关掉,只有在我们将要睡觉的时候才有必要定闹钟.

然后如果一切正常就返回 0.看完 finish_port_resume 函数,我们回到 hub_port_resume 中来,接下来,如果 status 小于 0,说明出了问题, 于是调用 hub_port_logical_disconnect,

```
1560 /*
1561  * Disable a port and mark a logical connnect-change event, so that some
1562  * time later khubd will disconnect() any existing usb_device on the port
1563  * and will re-enumerate if there actually is a device attached.
1564  */
1565 static void hub_port_logical_disconnect(struct usb_hub *hub, int port1)
1566 {
1567         dev_dbg(hub->intfdev, "logical disconnect on port %d\n", port1);
1568         hub_port_disable(hub, port1, 1);
1569
1570         /* FIXME let caller ask to power down the port:
1571          *   - some devices won't enumerate without a VBUS power cycle
1572          *   - SRP saves power that way
1573          *   - ... new call, TBD ...
1574          * That's easy if this hub can switch power per-port, and
1575          * khubd reactivates the port later (timer, SRP, etc).
1576          * Powerdown must be optional, because of reset/DFU.
1577          */
1578
1579         set_bit(port1, hub->change_bits);
1580         kick_khubd(hub);
1581 }
```

很简单,这个函数其实就是先把该端口关了,然后重新枚举该设备.因为刚才的返回值为负值说明出了某种问题,但并不确定究竟是何种问题,所以最省事的办法就是重新初始化该端口.而 1579 行这个 set_bit 设置了 hub->change_bits,于是我们在 hub_events()中就会根据这个来处理这个端口.(kick_khubd 会触发 hub_events,这我们早就知道.)

这时候我们注意到,在 hub_port_resume 的一开始我们调用 set_bit 设置了该端口对应的 busy_bits,而在 hub_port_resume 快结束的时候我们调用 clear_bit 清掉了这个 busy_bits. 唯一受此影响的函数是 hub_events(),当初我们其实提过,但我们在对一个端口进行 resume 或者 reset 的时候,hub_events 是不会对该端口进行任何操作的.

而 1832 行, busy_bits[0]为 0 就意味着所有的端口都没有处于 resume 或者 reset 阶段,hub->hdev->parent 为 NULL 则意味着当前 Hub 是 Root Hub,于是还是调用 usb_enable_root_hub_irq,当初我们在 hub_events()的结尾阶段也调用了这个函数.这就是调用 host controller driver 的一个函数 hub_irq_enable,某些主机控制器的端口连接是使用以电平触发的中断,这类主机控制器的驱动会提供这样一个函数,这个和具体的硬件有关,各家的产品不一样,咱们就不多说了.

至此,hub_port_resume 函数就返回了.回到 usb_port_resume,我们发现其实这个函数我们也已经看完了,因为 finish_port_resume 不小心也被我们讲完了.于是我们回到了

usb_resume_device,如果一切 Ok,那么 dev.power.power_state.event 也就设置为 PM_EVENT_ON.

然后我们经过跋山涉水翻山越岭之后再次回到了 usb_resume_both.1150 行,如果是 Root Hub,进入 else,我们一直说 Root Hub 没有 parent,其实这是不严谨的.我们注意到 struct usb_device 结构体有一个成员 struct usb_device *parent,同时我们还注意到 struct device 结构体本身也有一个成员 struct device *parent,其实对于 Root Hub 来说,是没有前者的那个 parent,但是却有后者这个 parent,后者的这个 parent 就是相应的 Host Controller 所对应的 struct device 结构体指针.所以这里的意思就很明白了,如果主机控制器没醒的话,Root Hub 以及其它的子设备再怎么玩也白搭.

如果 Host Controller 醒来了,那么 1159 行,对 Root Hub 来说,也调用 usb_resume_device 去唤醒它.

1161 行这个 else 的意思更直接,如果设备根本就没睡眠,那就没有什么唤醒它的意义了,调用 usb_resume_device 也不会做什么实事,无非就是设置 dev.power_power_state.event 为 PM_EVENT_ON,仅此而已.

1168 行,满足了集体的,再来满足个人的,社会主义制度优越性再次被体现.usb_resume_interface 按照 interface 的个数来循环调用,

```
887 /* Caller has locked intf's usb_device's pm_mutex */
888 static int usb_resume_interface(struct usb_interface *intf)
889 {
890         struct usb_driver       *driver;
891         int                     status = 0;
892
893                                 if    (interface_to_usbdev(intf)->state    ==
USB_STATE_NOTATTACHED ||
894                                         is_active(intf))
895                 goto done;
896
897         /* Don't let autoresume interfere with unbinding */
898         if (intf->condition == USB_INTERFACE_UNBINDING)
899                 goto done;
900
901         /* Can't resume it if it doesn't have a driver. */
902         if (intf->condition == USB_INTERFACE_UNBOUND) {
903                 status = -ENOTCONN;
904                 goto done;
905         }
906         driver = to_usb_driver(intf->dev.driver);
907
908         if (driver->resume) {
909                 status = driver->resume(intf);
```

```
910                 if (status)
911                         dev_err(&intf->dev, "%s error %d\n",
912                                                 "resume", status);
913                 else
914                         mark_active(intf);
915         } else {
916                 dev_warn(&intf->dev, "no resume for driver %s?\n",
917                                 driver->name);
918                 mark_active(intf);
919         }
920
921 done:
922           // dev_dbg(&intf->dev, "%s: status %d\n", __FUNCTION__,
status);
923         if (status == 0)
924                 intf->dev.power.power_state.event = PM_EVENT_ON;
925         return status;
926 }
```

898 行,关于 struct usb_interface 结构体的成员 condition 我们当初在 usb_reset_composite_device 中已经讲过,一共有四种状况,其含义正如其字面意义那样,无需多说.

908 行至 919 行这一段不用解释你也该明白吧,看过了当初那个 usb_suspend_interface()函数之后,我相信即便是西直门城铁站外面每天晚上等着招呼大家坐他的黑车的司机朋友也该知道现在这段代码的含义了.这里的 mark_active 和当初的那个 mark_quiesced 相对应,一个唱红脸一个唱白脸.而 909 行那个 driver->resume()就是调用属于该 interface 的驱动程序的 resume 函数,对于 hub driver,调用的自然就是 hub_resume,和前面那个 hub_suspend 相对应.

```
1962 static int hub_resume(struct usb_interface *intf)
1963 {
1964         struct usb_hub          *hub = usb_get_intfdata (intf);
1965         struct usb_device       *hdev = hub->hdev;
1966         int                     status;
1967
1968         dev_dbg(&intf->dev, "%s\n", __FUNCTION__);
1969
1970         /* "global resume" of the downstream HC-to-USB interface */
1971         if (!hdev->parent) {
1972                 struct usb_bus  *bus = hdev->bus;
1973                 if (bus) {
1974                         status = hcd_bus_resume (bus);
1975                         if (status) {
```

```
1976                                    dev_dbg(&intf->dev, "'global' resume
%d\n",
1977                                            status);
1978                                    return status;
1979                            }
1980                    } else
1981                            return -EOPNOTSUPP;
1982                    if (status == 0) {
1983                            /* TRSMRCY = 10 msec */
1984                            msleep(10);
1985                    }
1986            }
1987
1988            /* tell khubd to look for changes on this hub */
1989            hub_activate(hub);
1990            return 0;
1991 }
```

一路走来的兄弟们现在看着个函数是不是觉得有点小儿科,相当于一个游戏机高手去玩魂斗罗,菜鸟调出 30 条命来还未必能通关,可是高手也许一条命就能玩过八关(魂斗罗一代).这个函数我想就没有必要讲了,我们完全可以一目十行,它和 hub_suspend 实在是太他妈的对称了.对于 Roob Hub,需要调用 hcd_bus_resume,这个 host controller driver 那边的 resume 函数.最后调用 hub_activate()彻底激活 hub.

至此,我们算是把 usb_resume_both 看完了,看完了 usb_resume_both,也看完了 usb_suspend_both,我们就算是基本上知道了整个 usb 子系统是如何支持电源管理的,或者说如何支持 PM core 的.

挂起自动化

目睹了当今大学校园的素质流氓化,kiss 公开化,消费白领化,上课梦游化,逃课普遍化,补考专业化之后,区里的人们很时髦的提出了一个挂起自动化的概念.

接下来的一个话题就是 autosuspend/autoresume.

所谓的 autosuspend 就是 driver 自己判断是否需要挂起,而之前的 suspend/resume 是受外界影响的,比如说 PM core 统一的系统级的挂起,或者用户通过 sysfs 来触发的.于是我们现在就来看 driver 是如何自己判断的.首先从 autosuspend_check 看起,因为这个函数我们已经见过了,只是没有讲,在 usb_suspend_both 中就会调用它.它来自 drivers/usb/core/driver.c:

```
930 /* Internal routine to check whether we may autosuspend a device. */
931 static int autosuspend_check(struct usb_device *udev)
932 {
933         int                     i;
```

```
    934         struct usb_interface    *intf;
    935         unsigned long            suspend_time;
    936
    937         /* For autosuspend, fail fast if anything is in use or autosuspend
    938          * is disabled.  Also fail if any interfaces require remote wakeup
    939          * but it isn't available.
    940          */
    941         udev->do_remote_wakeup = device_may_wakeup(&udev->dev);
    942         if (udev->pm_usage_cnt > 0)
    943                 return -EBUSY;
    944                         if (udev->autosuspend_delay < 0 ||
udev->autosuspend_disabled)
    945                 return -EPERM;
    946
    947         suspend_time = udev->last_busy + udev->autosuspend_delay;
    948         if (udev->actconfig) {
    949                 for (i = 0; i < udev->actconfig->desc.bNumInterfaces;
i++) {
    950                         intf = udev->actconfig->interface[i];
    951                         if (!is_active(intf))
    952                                 continue;
    953                         if (intf->pm_usage_cnt > 0)
    954                                 return -EBUSY;
    955                         if (intf->needs_remote_wakeup &&
    956                                         !udev->do_remote_wakeup) {
    957                                 dev_dbg(&udev->dev, "remote wakeup
needed "
    958                                                 "for autosuspend\n");
    959                                 return -EOPNOTSUPP;
    960                         }
    961                 }
    962         }
    963
    964         /* If everything is okay but the device hasn't been idle for long
    965          * enough, queue a delayed autosuspend request.
    966          */
    967         if (time_after(suspend_time, jiffies)) {
    968                 if (!timer_pending(&udev->autosuspend.timer)) {
    969
    970                         /* The value of jiffies may change between the
    971                          * time_after() comparison above and the
subtraction
    972                          * below.  That's okay; the system behaves
sanely
```

```
    973                             * when a timer is registered for the present
moment
    974                           * or for the past.
    975                           */
    976                                   queue_delayed_work(ksuspend_usb_wq,
&udev->autosuspend,
    977                                                      suspend_time - jiffies);
    978                           }
    979                   return -EAGAIN;
    980           }
    981           return 0;
    982 }
```

首先, 获得 do_remote_wakeup,打不打开 remote wakeup 的功能是可以选择的.所以每次要记录下来.

pm_usage_cnt 表示引用计数,自动挂起的第一个重要条件就是 pm_usage_cnt 为 0.即只有一个设备没有被使用了我们才能把它挂起,否则比如你正在和恋人视频聊天,突然给你挂起,那你肯定会把开源社区的所有人的母亲都给问候一遍.

autosuspend_delay,也是在八大函数之一的 usb_alloc_dev 中赋的值,默认就是 2HZ,当然你可以自己设置,因为它是通过 usbcore 的模块参数 usb_autosuspend_delay 来设置的. usb_autosuspend_delay 缺省值为 2.另外这个值我们也可以通过 sysfs 来设置,如下所示:

```
localhost:/usr/src/linux-2.6.22.1/drivers/usb/core # ls
/sys/bus/usb/devices/1-1/power/
autosuspend  level  state  wakeup
```

autosuspend 文件就是记录这个值的,以秒为单位.

```
localhost:/usr/src/linux-2.6.22.1/drivers/usb/core # cat
/sys/bus/usb/devices/1-1/power/autosuspend
2
```

可以看到,我没有设置过的话,它这个值就是 2.这个值的意思是如果设备闲置了 2s,那么它将被自动挂起,这就是 autosuspend 的目的,你可以把它设为负值,为负就表示设备不能被 autosuspend,如果设备此时正处于 suspended 状态而你写一个负值进去,它将立刻被唤醒.写个 0 则表示设备将立刻被 autosuspended.

再来看 autosuspend_disabled.前面我们有看到一个 autoresume_disabled,这两个变量的含义都如字面意义一样.这两个变量都可以通过 sysfs 来改变,

```
localhost:/usr/src/linux-2.6.22.1/drivers/usb/core # ls
/sys/bus/usb/devices/1-1/power/
autosuspend  level  state  wakeup
```

这里的这个 level 就是反映的设备的电源级别,确切的说它就是通过 sysfs 为用户提供了一个自己挂起设备的方法.

localhost:/usr/src/linux-2.6.22.1/drivers/usb/core # cat /sys/bus/usb/devices/1-1/power/level
auto

它可以为 auto/on/suspend,这里我们看到它是 auto,auto 是最为常见的级别,而 on 就意味着我们不允许设备进行 autosuspend,即 autosuspend 被 disable 了,或者说这里的 autosuspend_disabled 被设置为了 1.如果是 suspend,就意味着我们不允许设备进行 autoresume,并且强迫设备进入 suspended 状态.即我们设置 autoresume_disabled 为 1,并且调用 usb_external_suspend_device()去挂起设备,这就是 sysfs 提供给用户的 suspend 单个设备的方法.

而 auto 状态意味着 autosuspend_disabled 和 autoresume_disabled 都没有设置,即都为 0,任其自然.

到这里你就能明白为什么我们当初在 usb_resume 中以及在 usb_resume_both 中会判断 udev->autoresume_disabled 了.因为设置了这个 flag 就等于对 resume 宣判了死刑.同样这里对 udev->autosuspend_disabled 的判断也是一样的道理.

last_busy,struct usb_device 的成员,unsigned long last_busy,有注释说 time of last use,不过我不知道该如何用中文表达,只可意会不能言传.不过没关系,我们慢慢看就明白了,其实你会发现在 resume 之后会更新它,在 suspend 之前也会更新它.你搜索一下源代码就会发现,其实这个变量基本上就被赋了一个值,那就是传说中的 jiffies,所以它实际上就是记录着这么一个时间值,我们这里 947 行,给一个局部变量 suspend_time 赋值,赋的就是 udev->last_busy 加上 udev->autosuspend_delay,我们马上就会看到 suspend_time 干嘛用的.

948 到 962 行,这一段就是判断各种异常条件,只有这些通通满足了才有必要作 autosuspend.其中,needs_remote_wakeup 咱们在 hub_probe()中见过,它是 struct usb_interface 的一个成员,unsigned needs_remote_wakeup,缺省值就是 1.咱们在 hub_probe 中也是设置为 1.如果设备需要 remote wakeup,而 do_remote_wakeup 被设置为了 0,那么就是说我本来需要被远程唤醒的,你却把我这项功能禁掉了,那么对于这种情况,这里保险起见,就不进行 autosuspend 了,因为万一把设备催眠了之后唤不醒了那就糟了,省电是省电了,设备醒不过来了,除非重起机器否则没办法了,这就属于捡了芝麻丢了西瓜的情况,咱们当然不能做.

964 行的注释说的很明白,如果一切 Okay,咱们才进行下面的代码,

这里有两个时间方面的函数,time_after,这个函数返回真如果从时间上来看,第一个参数在第二个参数之后.这里就是说如果 suspend_time 比 jiffies 后,我们刚才刚看了 suspend_time 的赋值,缺省来说 suspend_time 就比 jiffies 要多一个 autosuspend_delay,即 2 秒钟,所以这里为真.

第二个函数 timer_pending 就是判断一个计时器到点了没有,如果没有到点,即所谓的 pending 状态,那么函数返回真,否则返回 0.这里参数是 udev->autosuspend.timer,这个东东我们在八

大函数之一的usb_alloc_dev中调用INIT_DELAYED_WORK进行了初始化,进一步跟踪会发现实际上是调用 init_timer()来初始化 timer,而 init_timer 中 timer->entry.next=NULL,我们不用管这句话啥意思,但是我们可以看到,timer_pending 就是一个内联函数,来自include/linux/timer.h,

```
51 /**
52  * timer_pending - is a timer pending?
53  * @timer: the timer in question
54  *
55  * timer_pending will tell whether a given timer is currently pending,
56  * or not. Callers must ensure serialization wrt. other operations done
57  * to this timer, eg. interrupt contexts, or other CPUs on SMP.
58  *
59  * return value: 1 if the timer is pending, 0 if not.
60  */
61 static inline int timer_pending(const struct timer_list * timer)
62 {
63         return timer->entry.next != NULL;
64 }
```

所以很显然,这个函数将返回 0.因为 timer->entry.next==NULL.当然,我是说第一次.以后就不一样了. 因为这里 queue_delayed_work()会执行,我们前面已经提过,所以这行的意思就很简单了,把 udev->autosuspend 这个任务加入到 ksuspend_usb_wq 这个工作队列中去.并且设置了延时 suspend_time – jiffies,基本上这就意味着 2 秒之后就调用udev->autosuspend所对应的函数,即我们在usb_alloc_dev中用INIT_DELAYED_WORK注册的那个函数 usb_autosuspend_work 函数.因此我们就可以继续看usb_autosuspend_work这个函数了,来自drivers/usb/core/driver.c:

```
1209 /* usb_autosuspend_work - callback routine to autosuspend a USB device
*/
1210 void usb_autosuspend_work(struct work_struct *work)
1211 {
1212         struct usb_device *udev =
1213                                 container_of(work, struct usb_device,
autosuspend.work);
1214
1215         usb_autopm_do_device(udev, 0);
1216 }
```

其实调用的是usb_autopm_do_device,

```
1182 /* Internal routine to adjust a device's usage counter and change
1183  * its autosuspend state.
1184  */
```

```
1185 static int usb_autopm_do_device(struct usb_device *udev, int
inc_usage_cnt)
1186 {
1187         int     status = 0;
1188
1189         usb_pm_lock(udev);
1190         udev->auto_pm = 1;
1191         udev->pm_usage_cnt += inc_usage_cnt;
1192         WARN_ON(udev->pm_usage_cnt < 0);
1193         if (inc_usage_cnt >= 0 && udev->pm_usage_cnt > 0) {
1194                 if (udev->state == USB_STATE_SUSPENDED)
1195                         status = usb_resume_both(udev);
1196                 if (status != 0)
1197                         udev->pm_usage_cnt -= inc_usage_cnt;
1198                 else if (inc_usage_cnt)
1199                         udev->last_busy = jiffies;
1200         } else if (inc_usage_cnt <= 0 && udev->pm_usage_cnt <= 0) {
1201                 if (inc_usage_cnt)
1202                         udev->last_busy = jiffies;
1203                 status = usb_suspend_both(udev, PMSG_SUSPEND);
1204         }
1205         usb_pm_unlock(udev);
1206         return status;
1207 }
```

1193 行,inc_usage_cnt 咱们传递进来的是 0,但你会发现有的函数传递进来的是 1,比如 usb_autoresume_device,有的函数传递进来的是-1,比如 usb_autosuspend_device,还有另一个地方, usb_try_autosuspend_device,传递进来的也是 0.不过在 2.6.22.1 的内核中,总共也就这四处调用了 usb_autopm_do_device 这个函数.所以我们这里一并来看.

对于 inc_usage_cnt 大于等于 0 的情况,如果 pm_usage_cnt 也大于 0,那么如果设备此时又处于 SUSPENDED 状态,那么我们就调用 usb_resume_both 把它给恢复过来,恢复过来之后,设置 last_busy 为 jiffies,记录下这一神圣的时刻.如果没能恢复,那么就把 pm_usage_cnt 减回去.

如果 inc_usage_cnt 小于等于 0,如果 pm_usage_cnt 也小于等于 0,那么没什么好说的,把设备给挂起,挂起之前用 last_busy 记录下设备活着的那一时刻.

由于咱们这个情景传递进来的 inc_usage_cnt 是 0,所以 last_busy 没有改变.而 pm_usage_cnt 也没有改变,这就意味着咱们并不做什么引用计数上的改变,只是纯粹的 check,如果当前引用计数大于 0 而设备居然是挂起的状态,那么赶紧唤醒,如果当前引用计数已经小于等于 0,那么就自觉地挂起.

于是我们接下来要做的是两件事情,第一个,看一下另外三个调用 usb_autopm_do_device 的函数,第二个,回到 usb_suspend_both 中去看一下 autosuspend_check 是在什么情景下被调用的.

先看第二个问题,回过头来看 usb_suspend_both,发现,1042 行和 1068 行,如果 udev->auto_pm 不为 0,就调用 autosuspend_check,而 auto_pm 不为 0 恰恰是在 usb_autopm_do_device 中设置的,比如这里的 1190 行,还有一种可能是在 usb_autopm_do_interface 中设置的.后者我们暂时先不看.

调用 usb_suspend_both 的地方总共有三处,usb_autopm_do_device/usb_autopm_do_interface/usb_external_suspend_device,很显然,前两者属于同一情况,它们属于 autosuspend/autoresume 类型的,第三者属于另一种情况,它属于对非 autosuspend 的支持,即对 PM core 或者 sysfs 接口的支持.在 usb_external_suspend_device 中,调用 usb_suspend_both 之前先设置了 auto_pm 为 0.

所以,对于 autosuspend,usb_suspend_both 首先会调用 autosuspend_check,进而 usb_autopm_do_device 会被调用,而后者又会根据实际情况调用 usb_suspend_both 或者 usb_resume_both.

而对于非 autosuspend,usb_suspend_both 虽然也会被调用,但是 autosuspend_check 是不会执行的.

那么我们现在来看是另外三种调用 usb_autopm_do_device 的情况.三个函数全都来自 drivers/usb/core/driver.c:

```
1218 /**
1219  * usb_autosuspend_device - delayed autosuspend of a USB device and its interfaces
1220  * @udev: the usb_device to autosuspend
1221  *
1222  * This routine should be called when a core subsystem is finished using
1223  * @udev and wants to allow it to autosuspend.  Examples would be when
1224  * @udev's device file in usbfs is closed or after a configuration change.
1225  *
1226  * @udev's usage counter is decremented.  If it or any of the usage counters
1227  * for an active interface is greater than 0, no autosuspend request will be
1228  * queued.  (If an interface driver does not support autosuspend then its
1229  * usage counter is permanently positive.)  Furthermore, if an interface
1230  * driver requires remote-wakeup capability during autosuspend but remote
1231  * wakeup is disabled, the autosuspend will fail.
1232  *
1233  * Often the caller will hold @udev's device lock, but this is not
```

```
1234  * necessary.
1235  *
1236  * This routine can run only in process context.
1237  */
1238 void usb_autosuspend_device(struct usb_device *udev)
1239 {
1240         int      status;
1241
1242         status = usb_autopm_do_device(udev, -1);
1243         // dev_dbg(&udev->dev, "%s: cnt %d\n",
1244         //              __FUNCTION__, udev->pm_usage_cnt);
1245 }
1246
1247 /**
1248  * usb_try_autosuspend_device - attempt an autosuspend of a USB device
and its interfaces
1249  * @udev: the usb_device to autosuspend
1250  *
1251  * This routine should be called when a core subsystem thinks @udev may
1252  * be ready to autosuspend.
1253  *
1254  * @udev's usage counter left unchanged.  If it or any of the usage
counters
1255  * for an active interface is greater than 0, or autosuspend is not allowed
1256  * for any other reason, no autosuspend request will be queued.
1257  *
1258  * This routine can run only in process context.
1259  */
1260 void usb_try_autosuspend_device(struct usb_device *udev)
1261 {
1262         usb_autopm_do_device(udev, 0);
1263         // dev_dbg(&udev->dev, "%s: cnt %d\n",
1264         //              __FUNCTION__, udev->pm_usage_cnt);
1265 }
1266
1267 /**
1268  * usb_autoresume_device - immediately autoresume a USB device and
its interfaces
1269  * @udev: the usb_device to autoresume
1270  *
1271  * This routine should be called when a core subsystem wants to use
@udev
1272  * and needs to guarantee that it is not suspended.  No autosuspend will
1273  * occur until usb_autosuspend_device is called.  (Note that this will not
```

```
1274  * prevent suspend events originating in the PM core.)  Examples would be
1275  * when @udev's device file in usbfs is opened or when a remote-wakeup
1276  * request is received.
1277  *
1278   * @udev's usage counter is incremented to prevent subsequent autosuspends.
1279   * However if the autoresume fails then the usage counter is re-decremented.
1280  *
1281  * Often the caller will hold @udev's device lock, but this is not
1282  * necessary (and attempting it might cause deadlock).
1283  *
1284  * This routine can run only in process context.
1285  */
1286 int usb_autoresume_device(struct usb_device *udev)
1287 {
1288         int     status;
1289
1290         status = usb_autopm_do_device(udev, 1);
1291         // dev_dbg(&udev->dev, "%s: status %d cnt %d\n",
1292         //              __FUNCTION__, status, udev->pm_usage_cnt);
1293         return status;
1294 }
```

不看不知道,一看吓一跳,竟然都是如此赤裸裸的调用 usb_autopm_do_device 函数,用黎叔的话说,那叫一点儿技术含量都没有!

不过调用这三个函数的地方却很多很多.甚至我们都曾经见过,比如在 usb_suspend_both 中就调用了 usb_autosuspend_device, 用它来挂起父设备.即有这么一种可能 ,usb_autosuspend_device 调用 usb_suspend_both 挂起当前设备,而 usb_suspend_both 则调用 autosuspend_check 并进而是 usb_autopm_do_device 去挂起当前设备,而 usb_autopm_do_device 又还是调用 usb_suspend_both 去挂起设备,咦,怎么看怎么觉得我们走进了一个迷宫,走进了一条死胡同.很明显的是你调用我我调用你,这还不挂了?其实您尽管放心,别以为写代码的兄弟们都是吃素的,事实上有一把锁专门来对付这些情况,我们看到,在 usb_external_suspend_device 中调用 usb_suspend_both 的前后,调用了这两个函数:usb_pm_lock/usb_pm_unlock,在 usb_external_resume_device 中,调用 usb_resume_both 前后也是如此,而在 usb_autopm_do_device 和 usb_autopm_do_interface 中,也需要使用这些两个函数,即只有获得了这把锁才能去调用 usb_resume_both 或者 usb_suspend_both.所以,你尽管放心,永远只有一个人能执行 usb_suspend_both 或者 usb_resume_both,绝不可能陷入所谓的死胡同.所以其实你也看出来了,对于挂起,无论是不是自动化,里面终不过围绕一个函数,usb_suspend_both,对于唤醒,无论是不是自动化,里面终不过围绕一个函数,usb_resume_both,就好比,无论男人给女人讲多么浪漫的童话故事,里面终不过围绕一个字:床!

我想现在是时候来做一次总结了,关于电源管理方面的总结,先说 autosuspend,我们来看对 usb_try_autosuspend_device 的调用,前面我们在 usb_external_resume_device 中就看见了对它的调用,唤醒了设备之后,就尝试着看是否可以自动挂起.这其实体现的是一种勤俭节约的理念,因为 autosuspend/autoresume 这东西吧,纯粹是一种软件角度的主动,即从 driver 这边来自己做判断,凭借着第六感,当它觉得应该挂起设备的时候,它就会去尝试调用相关的挂起函数,当它觉得应该唤醒设备的时候,它就会去调用相关的唤醒函数.

那么也就是说,在整个 USB 子系统里,对电源管理的支持是按照两步走的.头些年,我们先实现传统的挂起/唤醒,即比如合上笔记本的时候,PM core 那边会按设备树来依次调用各个驱动的 suspend 函数,醒来的时候则调用相应的 resume 函数.于是从整个 usb 子系统的角度来说,我们提供了 usb_suspend/usb_resume 这两个函数.另一方面,我们这里把函数取名为 usb_external_suspend_device 和 usb_external_resume_device 也是针对 PM core 的系统睡眠(System Sleep),即我们把来自 PM core 的挂起请求/唤醒请求称为外部请求.而 usb_suspend/usb_resume 内部所调用的正是这两个函数.另一种调用 usb_external_suspend_device/usb_external_resume_device 的情况是通过 sysfs 的接口,由用户来触发,即通过改变前面我们看到的 sysfs 下面那个 level 的值来触发挂起或者唤醒.

完了后来第二步,大家觉得光这样不过瘾,于是又引入了 autosuspend 的概念.就是说驱动程序在适当的位置,自己去调用那些挂起函数/唤醒函数.即便 PM core 那边没有这个需求.就比如刚才这里这个对 usb_try_autosuspend_device 的调用,即设备刚刚唤醒,驱动程序就去检查看看是否可以挂起设备,因为可能你不小心唤醒了它但是你也许并不使用它,那么从省电的角度来说,驱动程序有足够的理由再次把你挂起.而像这种情形有很多,我们只要搜索一下看看有多少地方调用了 usb_autosuspend_device/usb_autoresume_device 就可以知道,在许多地方我们都这么做了.现以我们前面见过但没有讲的一个地方为例子说一下:

当初我们在 usb_reset_composite_device 中看到的,3076 行我们调用了 usb_autoresume_device,而 3119 行我们调用了 usb_autosuspend_device.后者很好理解,把一个设备 reset 之后,首先就去尝试把它挂起,理由很简单,比如你开机之后,你可能只是开机,你根本没打算使用任何 usb 设备,那么 usb 这边就默认把所有的设备都给挂起.等你真正要用的时候再去唤醒.而前者的目的更加简单,就是为了阻止后者的执行,因为这两个函数都将会调用 usb_autopm_do_device,而那句 usb_pm_lock 注定了这是一条独木桥,有你就没有我,有我便没有你.

最后再来关注一个变量,last_busy.它正是为 autosuspend 而生的.我们不难发现,每次设备被唤醒之后我们会把 last_busy 设置为当时的 jiffies,每次设备将要被挂起之前我们会把 last_busy 设置为当时的 jiffies.而真正要利用 last_busy 的是 autosuspend_check 函数,因为在该函数内,suspend_time 被赋值为 last_busy 加上 autosuspend_delay,假设后者为 2s.那么就是说 suspend_time 为 last_busy 加上 2 秒.比如说我们记录下上次设备被使用的时候 last_busy 为 3 点 25 分 0 秒,而现在是北京时间 3 点 25 分 1 秒,那么我们调用 autosuspend_check 的话就会激发一次与之相关的函数 usb_autosuspend_work.反之如果现在已经是北京时间 4 点了,那么说明我们已经没有必要激发 usb_autosuspend_work 了.换言之,last_busy 就是被用来决定是否进行 autosuspend 的一个 flag.last_busy 记录的是设备正忙的时间,设备总是在闲置了足够长的时间才可以被挂起.很显然,没有 last_busy,这个 autosuspend_delay 也就没法起作用了,毕竟这个 2s 总要在一个时间起点上开始加上去.当然,last_busy 和 suspend_time 这两个变量也只是几个月前才被加入到内核中来的,以前的内

核中并没有这么两个变量.当时 Alan 大侠添加这个变量的目的是为了完善他的 autosuspend.喜欢考古的朋友们不难从今年 3 月底 linux-usb-devel 邮件列表里挖出他当时的陈述:

This patch (as877) adds a "last_busy" field to struct usb_device, for use by the autosuspend framework. Now if an autosuspend call comes at a time when the device isn't busy but hasn't yet been idle for long enough, the timer can be set to exactly the desired value. And we will be ready to handle things like HID drivers, which can't maintain a useful usage count and must rely on the time-of-last-use to decide when to autosuspend.

用中文来说,就是两个理由要加入这么一个变量,一个是比如 autosuspend 调用发生的时候,它希望知道设备是否忙,用充分必要条件理论来说,设备不忙是 autosuspend 的必要条件,但这个必要条件满足了并不意味着设备会马上挂起,因为我们有一个 autosuspend_delay,即我们可以设置延时,如果按默认的 2s 钟来说,那你设备至少要闲置了 2 秒钟才会被挂起.所以就需要这么一个 flag,经常去记录着某个时间点,比如我们在 autoresume 之后的那一瞬间,又比如我们在 autosuspend 之前的一瞬间.道理很简单,第一,设备醒来之后那一瞬间基本上可以认为是忙的,如果不忙它干嘛不继续睡?第二,设备睡觉之前那一瞬间基本上也可以认为是忙的,如果不忙干嘛不早点睡?

引入 last_busy 的第二个理由是为了 HID 设备驱动的,比如鼠标,键盘.因为它们的挂起需要用一个时间来判断.不过我说过了,autosuspend 是一个很新的理念,所以至少到 2.6.22.1 的内核中,HID 那边还没有提供对 autosuspend 足够的支持,但是相信在不久的将来 last_busy 会被 HID Drivers 用到的.我们拿触摸屏来举例,我们知道苹果的 ipod 系列很多都是有触摸屏的,那么从驱动程序的角度来说呢,所谓的 autosuspend,理想情况就是,driver 能够检测到你的手指离开了触摸屏,然后隔一段时间 driver 就可以把设备自动挂起,反过来,driver 一旦检测到你的手指回来了,它又把设备唤醒.

网友"丰胸化吉"问我,那为何 last_busy 记录的是 resume 之后以及 suspend 之前的时间,而不是记录别的时间?这其实无所谓,你想多记录几个也没人拦住你,记录这几个时间的作用就是让 autosuspend 能在它们之后的两秒之后再执行,并没有更多的意思.你要是觉得某个地方之后不能立刻被挂起,那么你也可以在该处记录一个 last_busy.不过目前来看其它地方似乎并没有这个需求罢了.

好了,我的故事又讲完了.我知道,我并非是刻意追求娱乐技术化或者技术娱乐化,或许我只是在写一个北漂人的无奈,又或许我只是借文字以寻觅自己的未来.此时此刻,电台里放起了赵传的我是一只小小鸟,耳边响起了那耐人寻味的歌词:有时候我觉得自己是一只小小鸟,想要飞却怎么也飞不高,也许有一天我攀上了枝头(读完了大学)却成为住不起房子的人,飞上了青天(来到了北京)才发现自己从此无依无靠...每次到了夜深人静的时候我总是睡不着,我怀疑是不是只有我的明天没有变得更好,未来会怎样究竟有谁会知道,幸福是否只是一种传说我永远都找不到...