Linux SCSI子系统知识总结

2013.3 Version 1.01
yalung929@gmail.com

1 SCSI

1.1 基本模型

Initiator（启动器）：下发**SCSI命令**给Target ；Linux主机就是一个Initiator。

Target（服务器）：处理主机侧下发的命令，并返回结果；存储就是一个Target。

1.2 SCSI 命令

常见SCSI命令：

命令	用途
Test unit ready	查询设备是否已经准备好进行传输
Inquiry	请求设备基本信息
Request sense	请求之前命令的错误信息
Read capacity	请求存储容量信息
Read	从设备读取数据
Write	向设备写入数据

在内核代码中SCSI的命令定义在：

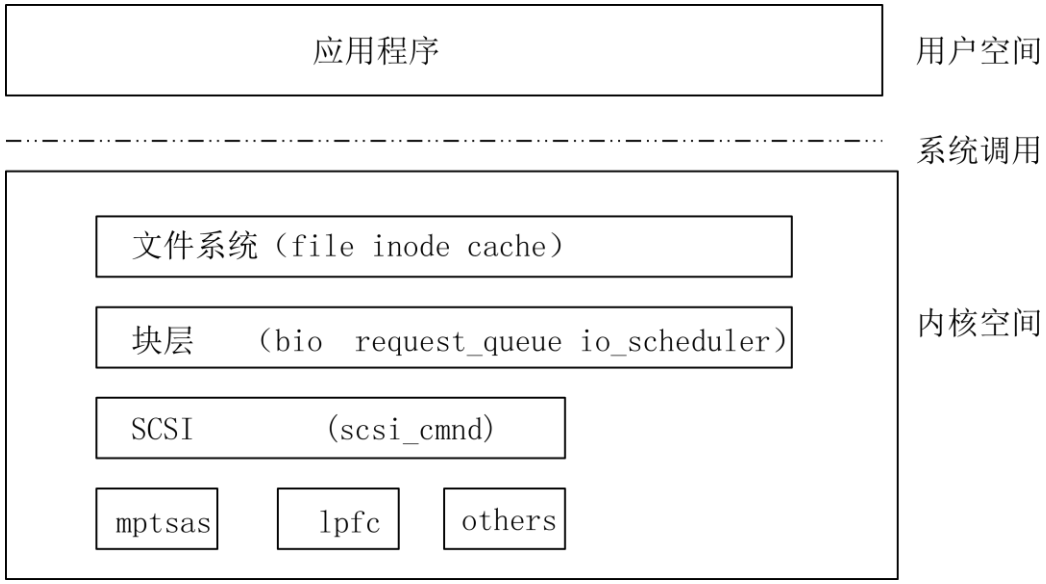
"include/scsi/scsi.h"

```
/*
 *       SCSI opcodes
 */

#define TEST_UNIT_READY       0x00
#define REZERO_UNIT           0x01
#define REQUEST_SENSE         0x03
#define FORMAT_UNIT           0x04
#define READ_BLOCK_LIMITS     0x05
#define REASSIGN_BLOCKS       0x07
#define INITIALIZE_ELEMENT_STATUS 0x07
#define READ_6                0x08
#define WRITE_6               0x0a
#define SEEK_6                0x0b
#define READ_REVERSE          0x0f
#define WRITE_FILEMARKS       0x10
#define SPACE                 0x11
#define INQUIRY               0x12
```

可以看到SCSI命令用于一个字节来表示。

1.3 SCSI 在 Linux 中的工作层次



工作在Linux的块层之下。

内核使用struct scsi_cmnd来描述SCSI命令。

块层的IO请求会被转化为struct scsi_cmnd并下发给底层设备驱动。

1.4 块层相关概念与定义

1.4.1 块设备

一个磁盘以及一个分区都是一个块设备。

一个块设备对应一个设备文件，位于/dev目录。

内核使用struct block_device来描述它。

1.4.2 磁盘

内核使用struct gendisk来描述它。

1.4.3 IO 请求队列

每一个磁盘（NOT 分区）都有一个IO请求队列。内核使用struct request_queue来描述它。

```
crash> dev -d
MAJOR GENDISK    NAME    REQUEST QUEUE      TOTAL ASYNC   SYNC    DRV
8     0xffff8801189a7400 sda            0xffff8801153860b8      0     0       0       0
```

我们对文件的读写操作通过文件系统层转化为对磁盘扇区的读写操作，下发给块层；
块层使用struct request封装这些操作请求，并把它们放入所操作磁盘的IO请求队列struct request_queue；
队列中的请求会使用IO调度算法（又称电梯算法）合并、排序，以减少磁头的来回移动。
这个算法可以在线调整:

```
# cat /sys/block/sda/queue/scheduler
noop anticipatory deadline [cfq]
```

1.5 SCSI 层相关概念与定义

1.5.1 SCSI 寻址

内核使用struct scsi_device来描述SCSI设备，每个SCSI设备通过一个唯一的四层地址结构标识：

HOST

主机侧每一个SCSI Adapter称为一个SCSI HOST。内核使用struct Scsi_Host 来描述它。

CHANNEL

一个SCSI Adapter可以有多个CHANNEL（又称为BUS）。

TARGET

连接到一个SCSI CHANNEL上的对端主体[1]，即执行主机侧下发命令的主体。内核使用struct scsi_target 来描述它。

LUN

一个TARGET上可以划分多个LUN，形成多个SCSI设备。

示例：

```
# lsscsi
[0:0:0:0]    disk    SEAGATE   ST9300605SS      B002   -
[0:0:1:0]    disk    SEAGATE   ST9300605SS      B002   -
[0:1:0:0]    disk    LSI       Logical Volume   3000   /dev/sda
[1:0:0:0]    disk    HUAWEI    S3900-M200       2102   -
[1:0:0:1]    disk    HUAWEI    S3900-M200       2102   -
[2:0:0:0]    disk    HUAWEI    S3900-M200       2102   -
[2:0:1:0]    disk    HUAWEI    S3900-M200       2102   -
[2:0:1:1]    disk    HUAWEI    S3900-M200       2102   -
```

这是一个linux环境上显示的SCSI设备，有三个SCSI HOST[2]：

本地的磁盘对应SCSI控制器以及HBA(Host Bus Adapter)卡上的两个HOST。

```
/sys/class/scsi_host # ll
total 0
lrwxrwxrwx 1 root root 0 Dec 23 12:14 host0
-> ../../devices/pci0000:00/0000:00:03.3/0000:17:00.0/host0/scsi_host/host0
lrwxrwxrwx 1 root root 0 Dec 23 12:14 host1
-> ../../devices/pci0000:00/0000:00:03.0/0000:0b:00.0/host1/scsi_host/host1
lrwxrwxrwx 1 root root 0 Dec 23 12:14 host2
-> ../../devices/pci0000:00/0000:00:03.0/0000:0b:00.1/host2/scsi_host/host2
```

我们的本地SCSI控制器通常有两个Channel，分别对应原始磁盘和raid控制器。
HBA上只有一个Channel。

1.5.2 SCSI DISK

一个SCSI DEVICE可以关联一个SCSI DISK对象，内核使用struct scsi_disk来描述它。
并非每一个SCSI DEVICE都会有这个关联。比如：

```
[0:0:0:0]    disk    SEAGATE   ST9300605SS      B002   -
  state=running queue_depth=254 scsi_level=7 type=0 device_blocked=0 timeout=60
```

1本地磁盘物理上与主机在一起，但在逻辑结构上和通过光纤连接的外置存储没有区别。

2 可以看到这些SCSI卡都被挂载PCI总线上，作为一个PCI设备存在。

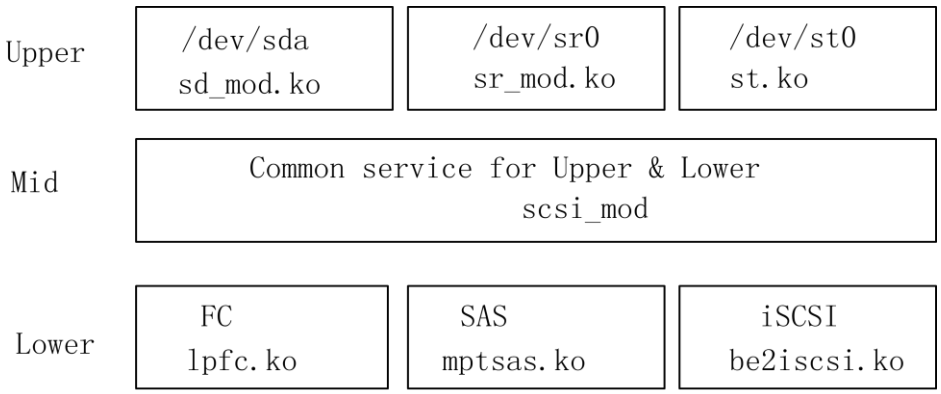
[0:0:1:0] disk SEAGATE ST9300605SS B002 -
state=running queue_depth=254 scsi_level=7 type=0 device_blocked=0 timeout=60 （raid）
[0:1:0:0] disk LSI Logical Volume 3000 /dev/sda

由于主机是通过RAID控制器访问RAID之后的磁盘，所以底层驱动只会为RAID对应的SCSI DEVICE建立磁盘对象。

一个SCSI DISK对象会对应：
一个磁盘对象（struct gendisk）
至少一个块设备对象（struct block_device）
以及一个IO队列（struct request_queue）。

这些对象都建立好之后，SCSI层就和块层联系起来了。
下发到块设备/dev/sda的IO就会进入SCSI层转化为SCSI命令，并发往远端的SCSI Target来处理。

1.6 Linux SCSI 子系统内部层次



1.6.1 Upper level

这一层主要定义和管理SCSI设备对上层的抽象，例如：

如果SCSI设备是一个磁盘，那么这一层就负责创建和管理一个块设备（struct block_device），就是我们看到的/dev/sda、/dev/sdb等等；对应的代码在drivers/scsi/sd.c内；对应内核模块sd_mod.ko。

如果SCSI设备是一个磁带机，那么这一层就负责创建和管理一个磁带机设备，就是我们看到的/dev/st0、/dev/st1等等；对应的代码在drivers/scsi/st.c内；对应内核模块st.ko。

如果SCSI设备是一个光驱，那么这一层就负责创建和管理一个光驱设备，就是我们看到的

/dev/sr0、/dev/sr1等等；对应的代码在drivers/scsi/sr.c内；对应内核模块sr_mod.ko。

这一层把SCSI设备的[host:channel:target:lun]形式映射为Linux块设备对象。

1.6.2 Low level

这一层主要是指底层设备驱动。例如这个服务器上有两种驱动：

```
# lsmod   |grep scsi
scsi_transport_fc       54918   1 lpfc
scsi_transport_sas      36393   1 mpt2sas
```

外部存储对应的驱动是lpfc和scsi_transport_fc。

其中lpfc由HBA卡厂家提供，scsi_transport_fc是内核SCSI子系统的一部分。

代码定义在"drivers/scsi/scsi_transport_fc.c"。

scsi_transport_xxx完成各种传输协议到SCSI层的适配，把底层的传输设备和SCSI层关联起来。

本地硬盘对应的驱动是mpt2sas和scsi_transport_sas，这说明本地硬盘是SAS硬盘。

这一层的驱动直接和底层设备打交道，接受上层下发的SCSI命令，并使用本设备对应的协议封装，发送到对端。

在驱动初始化的时候，会建立SCSI_HOST对象并注册SCSI命令入队函数。

比如lpfc注册了lpfc_queuecommand。IO经SCSI层转化为SCSI命令后，会调用lpfc注册的lpfc_queuecommand函数进入lpfc驱动，lpfc驱动再把SCSI命令封装为FC协议发送出去。

```
struct scsi_host_template lpfc_template = {
        .module                 = THIS_MODULE,
        .name                   = LPFC_DRIVER_NAME,
        .info                   = lpfc_info,
        .queuecommand           = lpfc_queuecommand,
       ……
};
```

1.6.3 Mid level

中间层为上层和下层提供一些公共的方法和接口。其实体是一个内核模块：

```
# lsmod   |   grep scsi_mod
scsi_mod      184454   9
```

lpfc,scsi_transport_fc,sg,scsi_tgt,sd_mod,megaraid_sas,mpt2sas,scsi_transport_sas,raid_class

1.7 工作流程

1.7.1 IO 下发流程

每一个磁盘设备都有一个请求队列，文件系统层的IO被丢到这队列里面。

```
crash> dev -d
MAJOR GENDISK            NAME       REQUEST QUEUE      TOTAL ASYNC  SYNC
DRV
    8      0xffff8801189a7400 sda        0xffff8801153860b8       3        0        3    3
```

每个请求队列都有一个对应的出队函数，对于SCSI磁盘设备来说它的出队函数是scsi_request_fn。

```
crash> struct request_queue 0xffff8801153860b8 |grep scsi_request_fn
request_fn = 0xffffffff81389c70 <scsi_request_fn>,
```

scsi_request_fn就是SCSI层的入口函数：

1. scsi_request_fn 从IO队列中取出一个请求request；(__elv_next_request)
2. 为之构造一个对应的SCSI命令；（sd_prep_fn）
3. 建立DMA映射，把要写的页面映射给底层设备；（scsi_init_io）
4. 从IO队列中删除这个请求；(blk_dequeue_request)
5. 调用底层驱动注册的queuecommand函数把scsi_cmnd下发给底层设备驱动。

1.7.2 IO 完成流程

在调用queuecommand把命令下发给底层驱动的时候，会注册回调函数scsi_done。

```
static void scsi_done(struct scsi_cmnd *cmd);
```

在命令完成之后底层设备会发中断，驱动在中断处理函数内调用scsi_done, scsi_done是IO完成的入口函数:

1. scsi_done在中断上下文干的活儿非常少，只是把这个完成的scsi_cmnd对应的请求（request）放到当前执行cpu的blk_cpu_done队列内，每一个cpu都有一个blk_cpu_done队列。(__blk_complete_request)
2. 在软中断内进行实际的工作，SCSI的软中处理函数是scsi_softirq_done；

3. scsi_softirq_done解析命令的完成结果，根据结果做不同的处理：

   如果是成功的，就回调上层注册的函数通知上层[3]。（scsi_finish_command）

   如果是需要重试，就重新进入IO请求队列，等待下次调度执行。（scsi_queue_insert）

   如果是错误的，就加入错误队列。（scsi_eh_scmd_add）

1.7.3 IO 异常流程

异常有两种：失败和超时，两种都会进入错误队列（shost->eh_cmd_q）等待处理。

失败的命令由scsi_softirq_done丢到错误队列（shost->eh_cmd_q）上。

超时的命令靠如下机制丢到错误队列（shost->eh_cmd_q）上：

1. 每一个命令在下发给底层驱动之前设置了一个定时器。
2. 定时器的回调函数是scsi_times_out。
3. scsi_times_out把超时的命令丢掉错误队列上。

每一个SCSI HOST都有一个错误队列和对应的处理线程。

```
# ls /sys/class/scsi_host/
host0    host1    host2
# ps -ef |grep scsi_eh |grep -v grep
root          374        2    0 11:01 ?          00:00:00 [scsi_eh_0] #host0
root         3151        2    0 11:02 ?          00:00:00 [scsi_eh_1] #host1
root         3471        2    0 11:02 ?          00:00:00 [scsi_eh_2] #host2
```

处理线程的入口函数是scsi_error_handler。

```
int scsi_error_handler(void *data);
```

scsi_error_handler的工作流程如下：

1. 对每个失败的命令，下发REQUEST_SENSE命令，获取错误信息。（scsi_eh_get_sense）
2. 错误处理成功的转移到错误处理完成队列。
3. 对超时的命令调用底层设备驱动注册的eh_abort_handler函数，让底层驱动取消这条命令的执行。（scsi_eh_abort_cmds）
4. Abort成功的转移到错误处理完成队列。
5. 如果经过上述两个处理之后，错误队列中还有命令[4]，就检查设备就绪状态，并一次尝试

[3] 被阻塞在read/write等系统调用内的进程在这个时候被唤醒结束D状态。

重置device（lun）、target、bus（channel）、host。（scsi_eh_ready_devs）

6. 如果重置都不凑效，就会下线设备，错误队列中所有的未处理成功的命令都会丢到错误处理完成队列。（scsi_eh_offline_sdevs）
7. 对错误处理完成队列中的所有命令，进行重试或者结束流程通知上层结果。（scsi_eh_flush_done_q）

1.8 日志输出

1.8.1 scsi_logging_level

SCSI 层使用一个32 bit的无符号整型控制日志输出

extern unsigned int scsi_logging_level;

32 bit被分割为10组。每一组有3bit，可以表示 0~7 8个日志级别（0是关闭，实际上就是7个级别，越高越详细）。

```
#define SCSI_LOG_ERROR_SHIFT            0
#define SCSI_LOG_TIMEOUT_SHIFT          3
#define SCSI_LOG_SCAN_SHIFT             6
#define SCSI_LOG_MLQUEUE_SHIFT          9
#define SCSI_LOG_MLCOMPLETE_SHIFT       12
#define SCSI_LOG_LLQUEUE_SHIFT          15
#define SCSI_LOG_LLCOMPLETE_SHIFT       18
#define SCSI_LOG_HLQUEUE_SHIFT          21
#define SCSI_LOG_HLCOMPLETE_SHIFT       24
#define SCSI_LOG_IOCTL_SHIFT            27
```

1.8.2 在线设置接口

/proc/sys/dev/scsi/logging_level

比如想最高级别打印超时和错误的命令处理情况：

即 0~2 bit都设置为111，3~5 bit都设置为1，111 111(二进制) –> 63（十进制）

echo 63 > /proc/sys/dev/scsi/logging_level

根据代码分析，对于定位IO问题，设置为4607比较合适：

[4] 这说明错误处理过程中又失败了。

4607 = 1 000 111 111 111即打开ERROR TIMEOUT SCAN 最高级别 和 MLCOMPLETE的1级。

（不能全开，全开会跟踪每一个IO无论成功与失败，由于本地硬盘也是SCSI的，日志会写到本地硬盘，就死循环了，非要全开必须把syslog关掉）

2 FC 与 SCSI

FC工作在SCSI层的Lower level层，可以理解为SCSI协议的承载（传输）协议，FC用于连接SCSI的Initiator和Target。

SCSI和FC之间的适配代码定义在"drivers/scsi/scsi_transport_fc.c"（scsi_transport_fc.ko）

2.1 主要概念

2.1.1 HBA

HBA (Host Bus Adapter)是连接主机内部IO通道和外部存储的适配器。就是我们说的Emulex光纤扣卡。

2.1.2 FC_HOST

HBA上的端口对应FC_HOST。

```
# ls -l /sys/class/fc_host/
total 0
lrwxrwxrwx 1 root root 0 Dec 25 14:51 host1
-> ../../devices/pci0000:00/0000:00:03.0/0000:0b:00.0/host1/fc_host/host1
lrwxrwxrwx 1 root root 0 Dec 25 14:52 host2
-> ../../devices/pci0000:00/0000:00:03.0/0000:0b:00.1/host2/fc_host/host2
```

每一个 FC_HOST 映射为一个 SCSI_HOST （由HBA卡驱动注册）。

2.1.3 FC_REMOTE_PORTS

FC网络上所有对HBA卡可见的其他FC设备体现为一个FC_REMOTE_PORTS。

```
# ls -l /sys/class/fc_remote_ports/
total 0
lrwxrwxrwx 1 root root 0 Dec 25 15:10 rport-1:0-3
```

-> ../../devices/pci0000:00/0000:00:03.0/0000:0b:00.0/host1/rport-1:0-3/fc_remote_ports/rport-1:0-3
lrwxrwxrwx 1 root root 0 Dec 25 15:10 rport-1:0-4
-> ../../devices/pci0000:00/0000:00:03.0/0000:0b:00.0/host1/rport-1:0-4/fc_remote_ports/rport-1:0-4
lrwxrwxrwx 1 root root 0 Dec 25 15:10 rport-1:0-5
-> ../../devices/pci0000:00/0000:00:03.0/0000:0b:00.0/host1/rport-1:0-5/fc_remote_ports/rport-1:0-5
lrwxrwxrwx 1 root root 0 Dec 25 15:11 rport-2:0-3
-> ../../devices/pci0000:00/0000:00:03.0/0000:0b:00.1/host2/rport-2:0-3/fc_remote_ports/rport-2:0-3
lrwxrwxrwx 1 root root 0 Dec 25 15:11 rport-2:0-4
-> ../../devices/pci0000:00/0000:00:03.0/0000:0b:00.1/host2/rport-2:0-4/fc_remote_ports/rport-2:0-4
lrwxrwxrwx 1 root root 0 Dec 25 15:11 rport-2:0-5
-> ../../devices/pci0000:00/0000:00:03.0/0000:0b:00.1/host2/rport-2:0-5/fc_remote_ports/rport-2:0-5
lrwxrwxrwx 1 root root 0 Dec 25 15:11 rport-2:0-6
-> ../../devices/pci0000:00/0000:00:03.0/0000:0b:00.1/host2/rport-2:0-6/fc_remote_ports/rport-2:0-6

2.1.4 FC_TRANSPORT

FC_REMOTE_PORTS上如果有SCSI TARGET（存储）会建立一个FC_TRANSPORT。

ls -l /sys/class/fc_transport
total 0
lrwxrwxrwx 1 root root 0 Dec 24 16:43 target1:0:0
-> ../../devices/pci0000:00/0000:00:03.0/0000:0b:00.0/host1/rport-1:0-5/target1:0:0/fc_transport/target1:0:0
lrwxrwxrwx 1 root root 0 Dec 24 16:43 target2:0:0
-> ../../devices/pci0000:00/0000:00:03.0/0000:0b:00.1/host2/rport-2:0-4/target2:0:0/fc_transport/target2:0:0

如果进而可以扫描到Lun，就建立一个SCSI DEVICE。

ls -l /sys/class/scsi_device/ |grep "rport"
lrwxrwxrwx 1 root root 0 Dec 25 15:49 1:0:0:0
-> ../../devices/pci0000:00/0000:00:03.0/0000:0b:00.0/host1/rport-1:0-5/target1:0:0/1:0:0:0/scsi_device/1:0:0:0

如上说明只有HOST1连接的存储给主机映射了Lun。

2.2 HBA 卡驱动的日志开关

emulex HBA卡驱动lpfc.ko的默认日志级别是0，我们可以在需要的时候在线设置为0xffffffff最高级别，提供更多的定位信息。

find /sys |grep lpfc_log_verbose

/sys/devices/pci0000:00/0000:00:03.0/0000:0b:00.0/host1/scsi_host/host1/lpfc_log_verbose

/sys/devices/pci0000:00/0000:00:03.0/0000:0b:00.1/host2/scsi_host/host2/lpfc_log_verbose

3 附录

3.1 lpfc 驱动 BUG 导致插拔光纤的时候主机 KDUMP 重启问题介绍

这个问题对SCSI处理流程有一个很好的解读，可以帮助理解。

3.1.1 堆栈

[93055.435456] BUG: unable to handle kernel NULL pointer dereference at (null)
[93055.435465] IP: [<ffffffffa0321bd3>] lpfc_scsi_cmd_iocb_cmpl+0x53/0x1ee0 [lpfc] [93055.435690] RIP:
0010:[<ffffffffa0321bd3>] [<ffffffffa0321bd3>] lpfc_scsi_cmd_iocb_cmpl+0x53/0x1ee0 [lpfc]
[93055.435710] RSP: 0018:ffff88004aec3b38 EFLAGS: 00010282
[93055.435716] RAX: 0000000000000000 RBX: ffff8810465ade70 RCX: ffff8810465ade70
[93055.435723] RDX: ffff88004aec3d68 RSI: ffff881076b5d5a0 RDI: ffff88107c762000
[93055.435729] RBP: ffff8810465ade00 R08: 0000000000000000 R09: 000000000000000a
[93055.435736] R10: ffff88004aec3d68 R11: 0000000000000000 R12: 00000000000f091a
[93055.435743] R13: 000000000000008a R14: ffff88004aec3d68 R15: ffff88107c762000
[93055.435792] Stack:
[93055.435796] ffff88004aec3bc8 ffffffff811e9ec3 0000000000000092 ffffffff8106a026
[93055.435808] <0> ffff88004aec3b68 00000000004afaea 0000000000000046 ffffffff8104edc5
[93055.435823] <0> ffff88004aec3bd7 ffffffff81ab4ba4 ffff88004aec3bd7 000000000000000f
[93055.435840] Call Trace:
[93055.435915] [<ffffffffa02e863c>] lpfc_sli_handle_fast_ring_event+0x1cc/0x730 [lpfc]
[93055.435942] [<ffffffffa02e8c59>] lpfc_sli_fp_intr_handler+0xb9/0x130 [lpfc]
[93055.435968] [<ffffffffa02e94c2>] lpfc_sli_intr_handler+0x122/0x190 [lpfc]
[93055.435990] [<ffffffff810a11b9>] handle_IRQ_event+0x39/0xe0
[93055.436004] [<ffffffff810a3924>] handle_fasteoi_irq+0x74/0xe0
[93055.436016] [<ffffffff81005d47>] handle_irq+0x17/0x20
[93055.436025] [<ffffffff81005255>] do_IRQ+0x65/0xe0
[93055.436037] [<ffffffff81003913>] ret_from_intr+0x0/0xa
[93055.436050] [<ffffffff8100af02>] mwait_idle+0x62/0x70
[93055.436061] [<ffffffff8100204a>] cpu_idle+0x5a/0xb0

根据堆栈分析，异常发生在以下红色的代码位置：

```
2414 static void
2415 lpfc_scsi_cmd_iocb_cmpl(struct lpfc_hba *phba, struct lpfc_iocbq *pIocbIn,
2416                         struct lpfc_iocbq *pIocbOut)
2417 {
2418         struct lpfc_scsi_buf *lpfc_cmd =
2419                 (struct lpfc_scsi_buf *) pIocbIn->context1;
```

```
2420            struct lpfc_vport          *vport = pIocbIn->vport;
2421            struct lpfc_rport_data *rdata = lpfc_cmd->rdata;
2422            struct lpfc_nodelist *pnode = rdata->pnode;
2423            struct scsi_cmnd *cmd;
2424            int result;
2425            struct scsi_device *tmp_sdev;
2426            int depth;
2427            unsigned long flags;
2428            struct lpfc_fast_path_event *fast_path_evt;
2429            struct Scsi_Host *shost;
2430            uint32_t queue_depth, scsi_id;
2431
2432            /* Sanity check on return of outstanding command */
2433            if (!(lpfc_cmd->pCmd))
2434                    return;
2435            cmd = lpfc_cmd->pCmd;
2436            shost = cmd->device->host;
2437

0000000000044b80 <lpfc_scsi_cmd_iocb_cmpl>:
    44b80:      41 57                   push   %r15
    44b82:      49 89 d2                mov    %rdx,%r10
    44b85:      41 56                   push   %r14
    44b87:      41 55                   push   %r13
    44b89:      41 54                   push   %r12
    44b8b:      55                      push   %rbp
    44b8c:      53                      push   %rbx
    44b8d:      48 81 ec 38 01 00 00    sub    $0x138,%rsp
    44b94:      48 89 7c 24 68          mov    %rdi,0x68(%rsp)
    44b99:      48 8b ae f0 00 00 00    mov    0xf0(%rsi),%rbp
    44ba0:      48 8b b6 e8 00 00 00    mov    0xe8(%rsi),%rsi
    44ba7:      48 89 b4 24 90 00 00    mov    %rsi,0x90(%rsp)
    44bae:      00
    44baf:      48 8b 45 18             mov    0x18(%rbp),%rax
    44bb3:      48 8b 00                mov    (%rax),%rax
    44bb6:      48 89 84 24 98 00 00    mov    %rax,0x98(%rsp)
    44bbd:      00
    44bbe:      48 8b 45 10             mov    0x10(%rbp),%rax # <=== rbp is lpfc_cmd
    44bc2:      48 85 c0                test   %rax,%rax
    44bc5:      48 89 44 24 70          mov    %rax,0x70(%rsp)
    44bca:      0f 84 a0 04 00 00       je     45070 <lpfc_scsi_cmd_iocb_cmpl+0x4f0>
    44bd0:      48 8b 00                mov    (%rax),%rax
    44bd3:      48 8b 00                mov    (%rax),%rax    #<=== rax is cmd->device
```

```
    44bd6:       48 89 84 24 a0 00 00      mov    %rax,0xa0(%rsp)
```

```
crash> struct lpfc_scsi_buf.pCmd ffff8810465ade00
  pCmd = 0xffff881198db6880,
crash> struct lpfc_scsi_buf.pCmd
struct lpfc_scsi_buf {
    [16] struct scsi_cmnd *pCmd;
}
```

其直接原因是struct scsi_cmnd *cmd（0xffff881198db6880）的device（rax）字段为0。进一步分析可以知道，struct scsi_cmnd被破坏了。再不考虑踩内存的情况，那就是这个命令被释放的了，而释放的只有在这个命令进入scsi_finish_command才会释放。

那么什么时候命令会进入scsi_finish_command执行释放动作？根据SCSI流程有几种可能：

1. 命令成功完成。
2. 命令错误，错误处理流程结束。
3. 命令超时，abort处理流程结束。

对于第一种可能，lpfc在调用scsi_done通知上层完成之前会释放命令对应的lpfc_scsi_buf。不会再访问到这个命令。

对于第二种可能，等同于第一种。

对于第三种可能，abort过程会调用lpfc驱动注册的lpfc_abort_handler，在返回取消成功之前，会释放cmd对应的lpfc_scsi_buf。不会再访问到这个命令。

当然这只是在代码没BUG的情况下。既然现在lpfc访问到了一个已经被SCSI层释放了的命令，就一定是在某个流程内出了问题。

经过和lpfc供应商确认，问题出在abort流程。lpfc_abort_handler有一个BUG。

```
static int
lpfc_abort_handler(struct scsi_cmnd *cmnd)
{
        struct Scsi_Host    *shost = cmnd->device->host;
        struct lpfc_vport *vport = (struct lpfc_vport *) shost->hostdata;
        struct lpfc_hba     *phba = vport->phba;
        struct lpfc_iocbq *iocb;
        struct lpfc_iocbq *abtsiocb;
        struct lpfc_scsi_buf *lpfc_cmd;
        IOCB_t *cmd, *icmd;
        int ret = SUCCESS;
        DECLARE_WAIT_QUEUE_HEAD_ONSTACK(waitq);
```

```
ret = fc_block_scsi_eh(cmnd);
if (ret)
        return ret;
lpfc_cmd = (struct lpfc_scsi_buf *)cmnd->host_scribble;
if (!lpfc_cmd) {
        lpfc_printf_vlog(vport, KERN_WARNING, LOG_FCP,
                "2873 SCSI Layer I/O Abort Request IO CMPL Status "
                "x%x ID %d "
                "LUN %d snum %#lx\n", ret, cmnd->device->id,
                cmnd->device->lun, cmnd->serial_number);
        return SUCCESS;
}
/*
 * If pCmd field of the corresponding lpfc_scsi_buf structure
 * points to a different SCSI command, then the driver has
 * already completed this command, but the midlayer did not
 * see the completion before the eh fired.    Just return
 * SUCCESS.
 */
iocb = &lpfc_cmd->cur_iocbq;
if (lpfc_cmd->pCmd != cmnd)
        goto out;

BUG_ON(iocb->context1 != lpfc_cmd);

abtsiocb = lpfc_sli_get_iocbq(phba);
```

fc_block_scsi_eh 是SCSI层的一个接口，其返回值有两种：SUCCESS和FAST_IO_FAIL，分别对应0x2002和0x2009 。

但这段代码的if判断无论什么情况下都会返回成功。

```
ret = fc_block_scsi_eh(cmnd);
if (ret)
        return ret;
```

进而结束lpfc_abort_handler函数，直接返回给SCSI层，然后SCSI层就释放了命令。但是驱动这侧没有做任何清理动作，与之关联的lpfc_scsi_buf还没有释放！

也就是说，如果后面驱动收到命令的返回结果，就会访问它，就会触发问题。

但是在这个环境超时的命令永远不会返回（超时是另外一个问题引起的），这样只会造成lpfc_scsi_buf积累过多，引起lpfc缓冲池满阻塞IO的问题。

```
lpfc_cmd = lpfc_get_scsi_buf(phba, ndlp);
if (lpfc_cmd == NULL) {
```

```
                lpfc_rampdown_queue_depth(phba);

                lpfc_printf_vlog(vport, KERN_INFO, LOG_FCP,
                                 "0707 driver's buffer pool is empty, "
                                 "IO busied\n");
                goto out_host_busy;
        }
```

Dec 17 16:02:12 cbs139 kernel: [412644.056188] lpfc 0000:0b:00.0: 0:(0):0707 driver's buffer pool is empty, IO busied

Dec 17 16:04:12 cbs139 kernel: [412764.013707] lpfc 0000:0b:00.0: 0:(0):0707 driver's buffer pool is empty, IO busied

那么为什么在插拔光纤的时候驱动收到了这个命令的处理结果？经确认这是因为在插拔光纤的时候，HBA卡会进入链路重置过程，这个过程中firmware会把原先还没有结果的命令都返回（发中断）通知驱动处理，于是就触发了问题。

根据上面的分析，如果是这个BUG导致的KDUMP，那么前面应该有abort这条命令的日志记录。（之前打开了日志开关）

但是在KDUMP VMCORE的dmesg中查不到任何abort记录，有可能是日志太多，被冲掉了。

在一天前的归档的message文件中找到了这条命令（0xffff881198db6880）！

Dec 20 23:48:32 cbs141 kernel: [31872.529269] sd 3:0:1:8: [sdd] Done: TIMEOUT

Dec 20 23:48:32 cbs141 kernel: [31872.529275] sd 3:0:1:8: [sdd] Result: hostbyte=DID_OK driverbyte=DRIVER_OK

Dec 20 23:48:32 cbs141 kernel: [31872.529280] sd 3:0:1:8: [sdd] CDB: Write(10): 2a 00 03 7c 75 68 00 02 00 00

Dec 20 23:48:32 cbs141 kernel: [31872.529294] Waking error handler thread

Dec 20 23:48:32 cbs141 kernel: [31872.529303] Error handler scsi_eh_3 waking up

Dec 20 23:48:32 cbs141 kernel: [31872.529325] sd 3:0:1:8: scsi_eh_prt_fail_stats: cmds failed: 0, cancel: 1

Dec 20 23:48:32 cbs141 kernel: [31872.529330] Total of 1 commands on 1 devices require eh work

Dec 20 23:48:32 cbs141 kernel: [31872.529334] scsi_eh_3: aborting cmd:0xffff881198db6880

Dec 20 23:48:32 cbs141 kernel: [31872.529465] scsi_eh_done scmd: ffff881198db6880 result: 0

Dec 20 23:48:32 cbs141 kernel: [31872.529474] scsi_send_eh_cmnd: scmd: ffff881198db6880, timeleft: 7500

Dec 20 23:48:32 cbs141 kernel: [31872.529477] scsi_send_eh_cmnd: scsi_eh_completed_normally 2002

Dec 20 23:48:32 cbs141 kernel: [31872.529480] scsi_eh_tur: scmd ffff881198db6880 rtn 2002

Dec 20 23:48:32 cbs141 kernel: [31872.529484] scsi_eh_3: flush retry cmd: ffff881198db6880

4 参考资料

《Linux kernel source code 2.6.32》