openEuler

内核问题定位分析

目录

CONTENT

openEuler

问题描述

- 在执行 ltp 测试用例 ioctl_sg01 时，内核发生崩溃。

- 由于我们事先开启了 kdump 功能，所以在内核崩溃时，我们能够获取到 vmcore，vmcore 是内核崩溃时的内核状态。vmcore 中包含了非常多的信息，对于我们定位问题非常有帮助。

vmcore分析

- 通过 crash 工具分析 vmcore，首先执行crash vmcore vmlinux，打印出内核PANIC的原因：

```
WARNING: kernel version inconsistency between vmlinux and dumpfile

    KERNEL: vmlinux
    DUMPFILE: vmcore  [PARTIAL DUMP]
    ...
    PANIC: "BUG: unable to handle kernel NULL pointer dereference at 0000000000000008"
        PID: 387
    COMMAND: "kswapd1"
        TASK: ffff97763d725ac0  [THREAD_INFO: ffff97763d725ac0]
        CPU: 20
    STATE: TASK_RUNNING (PANIC)
```

1.直接原因：访问空指针

2. 出错的线程

注：kswapd是linux中用于页面回收的内核线程。

openEuler

vmcore分析

- 接下来把系统崩溃前的堆栈等信息打印出来，再把崩溃处的代码和指令打印出来：

```
crash> bt
PID: 387    TASK: ffff97763d725ac0  CPU: 20  COMMAND: "kswapd1"
#0 [ffff97763d71fa08] machine_kexec at ffffffff826442c2
#1 [ffff97763d71fa50] __crash_kexec at ffffffff82721c09
#2 [ffff97763d71fb08] crash_kexec at ffffffff827229e8
#3 [ffff97763d71fb20] oops_end at ffffffff826199ba
#4 [ffff97763d71fb40] no_context at ffffffff826513ed
#5 [ffff97763d71fb90] __do_page_fault at ffffffff82651b3d
#6 [ffff97763d71fbf8] do_page_fault at ffffffff82651f7a
#7 [ffff97763d71fc20] page_fault at ffffffff8300116e
    [exception RIP: deferred_split_scan+286]
    RIP: ffffffff82812a7e  RSP: ffff97763d71fcd8  RFLAGS: 00010046
    RAX: ffffe2843fa00118  RBX: ffff97763d71fd90  RCX: 0000000000000000
    RDX: ffffe2843fa00080  RSI: 0000000000000286  RDI: 0000000000000001
    RBP: ffff97763d71fce0   R8: 0000000000000000   R9: ffff977741413478
    R10: 0000000000000040  R11: 0000000000000000  R12: ffff97b63c701008
    R13: ffff97b63c701000  R14: ffff97b63c701018  R15: ffffffff83900ae0
    ORIG_RAX: ffffffffffffffff  CS: 0010  SS: 0000
#8 [ffff97763d71fd20] do_shrink_slab at ffffffff827a7893
#9 [ffff97763d71fd70] shrink_slab at ffffffff827a7cfb
#10 [ffff97763d71fde0] shrink_node at ffffffff827abc73
#11 [ffff97763d71fe58] kswapd at ffffffff827aca36
#12 [ffff97763d71ff10] kthread at ffffffff82
#13 [ffff97763d71ff50] ret_from_fork at ffff
```

3. 内核崩溃在这里，
RIP = ffffffff82812a7e

5.链表操作！

```
crash> dis -l ffffffff82812a7e
.../include/linux/list.h: 105
0xffffffff82812a7e <deferred_split_scan+286>:   mov    %rdi,0x8(%r8)
```

4. 访问空指针是因为r8的值为0

openEuler

vmcore分析

- 我们把 deferred_split_scan 反汇编出来，看一下这个r8是哪个变量，中间发生了：

```
crash> dis deferred_split_scan
...     // 省略一部分代码
0xffffffff82812a67 <deferred_split_scan+263>:   jmp    0xffffffff82812a59 <deferred_sp
0xffffffff82812a69 <deferred_split_scan+265>:   mov    0x98(%rdx),%r8
0xffffffff82812a70 <deferred_split_scan+272>:   mov    0xa0(%rdx),%rdi
0xffffffff82812a77 <deferred_split_scan+279>:   lea    0x98(%rdx),%rax
0xffffffff82812a7e <deferred_split_scan+286>:   mov    %rdi,0x8(%r8)
0xffffffff82812a82 <deferred_split_scan+290>:   mov    %r8,(%rdi)
0xffffffff82812a85 <deferred_split_scan+293>:   mov    %rax,0x98(%rdx)
0xffffffff82812a8c <deferred_split_scan+300>:   mov    %rax,0xa0(%rdx)
0xffffffff82812a93 <deferred_split_scan+307>:   subq   $0x1,(%r14)
0xffffffff82812a97 <deferred_split_scan+311>:   subq   $0x1,0x8(%rbx)
...     // 省略一部分代码
```

9.根据前面的堆栈信息可以查到rdx = ffffe2843fa00080

```
static unsigned long deferred_split_scan(struct shrinker *shrink,
                                         struct shrink_control *sc)
{
    ...
    list_for_each_safe(pos, next, ds_queue.split_queue) {
        page = list_entry((void *)pos, struct page, mapping);
        page = compound_head(page);
        if (get_page_unless_zero(page)) {
            list_move(page_deferred_list(page), &list);
        } else {
            /* We lost race with put_compound_page() */
            list_del_init(page_deferred_list(page));
            (*ds_queue.split_queue_len)--;
        }
        if (!--sc->nr_to_scan)
            break;
    }
    ...
}
```

6.链表操作

```
static inline struct list_head *page_deferred_list(struct page *page)
{
    /*
     * Global or memcg deferred list in the second tail pages is
     * occupied by compound_head.
     */
    return &page[2].deferred_list;
}
```

7. 这个会不会是0x98偏移

openEuler

vmcore分析

```
crash> struct page -o
struct page {
[0] unsigned long flags;
    union {
        struct {                        // 复合页的page[0]使用这个结构体
[8]         struct list_head lru;
[24]         struct address_space *mapping;
[32]         unsigned long index;
[40]         unsigned long private;
        };
        ...
        struct {                        // 复合页的page[1]使用这个结构体
[8]         unsigned long compound_head;
[16]         unsigned char compound_dtor;
[17]         unsigned char compound_order;
[20]         atomic_t compound_mapcount;
        };
        struct {                        // 复合页的page[2]使用这个结构体
[8]         unsigned long _compound_pad_1;
[16]         unsigned long _compound_pad_2;
[24]         struct list_head deferred_list;
        };
        ...
    };
    ...
[52] atomic_t _refcount;
[56] struct mem_cgroup *mem_cgroup;
}
SIZE: 64
```

8. 这里可以验证一下page[2].deferred_list的偏移就是0x98

```
struct list_head {
        struct list_head *next, *prev;
};
```

openEuler

vmcore分析

- 观察这段内存，每个 page 占 64 字节，它们应该组成一个复合页，但对照上面 struct page 结构提的定义，可以看出它们的值明显是不合理的，特别是这些页的引用计数已经为 -1 或 -128 了。

```
crash> rd -s ffffe2843fa00000 -e ffffe2843fa00140
ffffe2843fa00000:  0017ffffc0000000 ffffe283abe00008
ffffe2843fa00010:  ffffe2840ae30008 0000000000000000
ffffe2843fa00020:  00000007fc730600 000000000000000a
ffffe2843fa00030:  00000000ffffff7f 0000000000000000
ffffe2843fa00040:  0017ffffc0000000 0000000000000000
ffffe2843fa00050:  ffffffff00000903 0000000000000000
ffffe2843fa00060:  0000000000000001 0000000000000000
ffffe2843fa00070:  00000000ffffffff 0000000000000000
ffffe2843fa00080:  0017ffffc0000000 0000000000000000    // rdx寄存器所指示的地址
ffffe2843fa00090:  dead000000000200 0000000000000000
ffffe2843fa000a0:  dead000000000200 0000000000000000
ffffe2843fa000b0:  00000000ffffffff 0000000000000000
ffffe2843fa000c0:  0017ffffc0000000 0000000000000000
ffffe2843fa000d0:  dead000000000200 0000000000000000
ffffe2843fa000e0:  0000000000000001 0000000000000000
ffffe2843fa000f0:  00000000ffffffff 0000000000000000
ffffe2843fa00100:  0017ffffc0000000 0000000000000000
ffffe2843fa00110:  dead000000000200 0000000000000000
ffffe2843fa00120:  0000000000000001 0000000000000000
ffffe2843fa00130:  00000000ffffffff 0000000000000000
```

openEuler

猜想&验证

- 复合页可以使用 compound_head 函数找到复合页的 “head page”，如果按上一页page段内存信息来看，ffffe2843fa00080 所指示的这个 page 的 compound_head 将会返回它本身，所以我们猜测ffffe2843fa00080可能是 page[2]；
- 如果这一点成立，那么：
  - ffffe2843fa00098 对应 deferred_list.next=0000000000000000
  - ffffe2843fa000a0 对应 deferred_list.prev=dead000000000200
- 如果ffffe2843fa00080是被释放了的，那么
  - ffffe2843fa00098 对应的是 mapping=0000000000000000

```
static unsigned long deferred_split_scan(struct shrinker *shrink,
                                         struct shrink_control *sc)
{
    ...
    list_for_each_safe(pos, next, ds_queue.split_queue) {
        page = list_entry((void *)pos, struct page, mapping);
        page = compound_head(page);
        if (get_page_unless_zero(page)) {
            list_move(page_deferred_list(page), &list);
        } else {
            /* We lost race with put_compound_page() */
            list_del_init(page_deferred_list(page));
            (*ds_queue.split_queue_len)--;
        }
        if (!--sc->nr_to_scan)
            break;
    }
    ...
}
```

openEuler

猜想&验证

- 如果上面的猜测是正确的，那么我们看一下 deferred_list 都在哪里做修改，有没有并发修改或者异常修改的可能性：
- page.deferred_list 都是通过 page_deferred_list 获取的，通过跟踪 page_deferred_list 这个函数的调用点，我们发现相关的操作都是有锁进行保护的。获取的锁是通过 page->mem_cgroup 或 pglist_data 得到的，理论上是不会有并发问题的。
- 那我们继续猜想：会不会在执行过程中，page 的 mem_cgroup 发生变化，但是 deferred_list 仍然在原 mem_cgroup 上。进而转为跟踪 mem_cgroup 的变化，最终我们发现在 move_active_pages_to_lru 函数流程中会调用 mem_cgroup_uncharge(page)，但此时 page 的 deferred_list 没有改变，在这里调用 mem_cgroup_uncharge() 之后，实际上会对 deferred_list 进行操作，然后释放 page，这里会导致链表发生异常。
- 进一步分析发现这里不应该执行 mem_cgroup_uncharge()，因为在 free_compound_page() 时会执行 mem_cgroup_uncharge。修复之后重新测试，确认内核不再崩溃。

代码推演

- 首先在触发内存回收时，会调用 shrink_active_list 函数扫描 active pages ， 接着调用 move_active_pages_to_lru 函数：

```
static unsigned move_active_pages_to_lru(struct lruvec *lruvec,
                                         struct list_head *list,
                                         struct list_head *pages_to_free,
                                         enum lru_list lru)
{
    ...
    while (!list_empty(list)) {
        ...
        if (put_page_testzero(page)) {      // 引用计数减为0的情况
                        ...
                        if (unlikely(PageCompound(page))) {     // 复合页的处理流程，也就包括透明大页
                                spin_unlock_irq(&pgdat->lru_lock);
                                mem_cgroup_uncharge(page);      // 从 page 从 memcg 中移除，将 page->memcg 置空
                                (*get_compound_page_dtor(page))(page);  // 执行对应的释放函数，下面介绍
                                spin_lock_irq(&pgdat->lru_lock);
                        } else
                                list_add(&page->lru, pages_to_free);
        }
    }
    ...
}
```

这里把page->memcg置空，但是page[2].deferred_list仍然挂在memcg->split_queue上

openEuler

代码推演

```
compound_page_dtor * const compound_page_dtors[] = {
        NULL,
        free_compound_page,     // 一般复合页的释放函数
#ifdef CONFIG_HUGETLB_PAGE
        free_huge_page,         // Hugetlb的释放函数
#endif
#ifdef CONFIG_TRANSPARENT_HUGEPAGE
        free_transhuge_page,    // 透明大页的释放函数
#endif
};

static inline compound_page_dtor *get_compound_page_dtor(struct page *page)
{
        VM_BUG_ON_PAGE(page[1].compound_dtor >= NR_COMPOUND_DTORS, page);
        return compound_page_dtors[page[1].compound_dtor];
}
```

```
void free_transhuge_page(struct page *page)
{
        struct deferred_split ds_queue;
        unsigned long flags;

        get_deferred_split_queue(page, &ds_queue);      // 获取 page 所属的 split_queue
        spin_lock_irqsave(ds_queue.split_queue_lock, flags); // 对上述 split_queue 加锁
        if (!list_empty(page_deferred_list(page))) {
                (*ds_queue.split_queue_len)--;
                list_del(page_deferred_list(page));     // 把 page 从 split_queue 上删除
        }
        spin_unlock_irqrestore(ds_queue.split_queue_lock, flags);
        free_compound_page(page);
}
```

这里获取到错误的split_queue_lock
导致下面的链表操作发生错误

split_queue：
用于透明大页的拆分，每个 pgdat 以及每个 memcg 都会有一个 split_queue 链表和相应的 split_queue_lock 锁。如果 page 属于某个 memcg 就会把 page[2].deferred_list 挂在 memcg->split_queue 上，否则挂在 pgdat->split_queue 上，之后根据 page->memcg 是否为空来判断在哪个 split_queue 上。

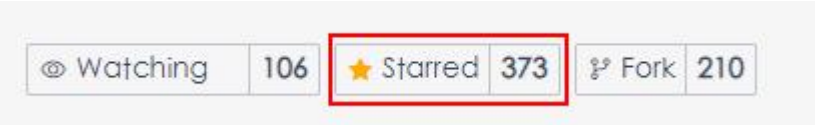
openEuler

✓ openEuler kernel gitee 仓库

源代码仓库

https://gitee.com/openeuler/kernel

欢迎大家多多 Star，多多参与社区开发，多多贡献补丁。



✓ maillist、issue、bugzilla

可以通过邮件列表、issue、bugzilla 参与社区讨论

欢迎大家多多讨论问题，发现问题多提 issue、bugzilla

https://gitee.com/openeuler/kernel/issues

https://bugzilla.openeuler.org

kernel@openeuler.org

✓ openEuler kernel SIG 微信技术交流群

请扫描右方二维码添加小助手微信

或者直接添加小助手微信（微信号：openeuler-kernel）

备注“交流群”或“技术交流”

加入 openEuler kernel SIG 技术交流群

技术交流

openEuler

Thank you

openEuler