背景知识

1、OOM 打分机制

当系统内存耗尽，触发 out of memory 时。内核会给当前系统的所有 task 进行打分，并选取得分最高的 task 并终止运行。

评选机制如下：

```
out_of_memory
    -> select_bad_process
        -> oom_evaluate_task
            -> oom_badness          -------------    对进程 task 进行打分
```

```c
/**
 * oom_badness - 启发函数 - 用于确定要终止的候选进程
 * @p: 参与计算的进程结构体
 * @totalpages: 用于页面分配的总物理内存
 *
 * 采用启发式的方法，尽可能简单地预测要终止的进程。目标是选择内存消耗最多的进程，将其终止。
 */
long oom_badness(struct task_struct *p, unsigned long totalpages)
{
    long points;
    long adj;

    // 如果进程不可被杀死，则直接跳过
    if (oom_unkillable_task(p))
        return LONG_MIN;
    // 找到进程 p，并使用 task_lock() 锁上
    p = find_lock_task_mm(p);
    if (!p)
        return LONG_MIN;

    /*
     * 不考虑哪些明确标记为不可被 OOM 杀死的 tasks，或者已经被 OOM 回收的 tasks，
     * 或者正在进行 vfork 的 tasks。
     */
    adj = (long)p->signal->oom_score_adj;   // 获取当前进程的 oom_score_adj 参数
    // adj == OOM_SCORE_ADJ_MIN 话，该进程不参与评比。OOM_SCORE_ADJ_MID = -1000
    if (adj == OOM_SCORE_ADJ_MIN ||
            test_bit(MMF_OOM_SKIP, &p->mm->flags) ||
            in_vfork(p)) {
        task_unlock(p);
        return LONG_MIN;
    }

    /*
     * 坏度分数的基准是每个任务的 RSS（常驻内存集）、页表和交换空间使用的内存比例。这里使用 page 个数统计。
     */
    points = get_mm_rss(p->mm) + get_mm_counter(p->mm, MM_SWAPENTS) +
        mm_pgtables_bytes(p->mm) / PAGE_SIZE;
    task_unlock(p);
```

```
    /* 归一化 oom_score_adj 单位 */
    adj *= totalpages / 1000;
    /* 将归一化后的 adj 和 points 求和，作为当前进程的分数 */
    points += adj;

    return points;
}
```

- `oom_score_adj` 为 `OOM_SCORE_ADJ_MIN` 的进程不参加评选，进程的 `oom_score_adj` 值在 `/proc/xxx/oom_score_adj` 中设置。
- `mm->flags` 为 `MMF_OOM_SKIP` 的进程不参加评选
- 处于 `vfork()` 中的进程不参加评选
- 进程得分要素为 `RSS`（常驻内存集）、页表和交换空间使用的内存

所以进程得分：`points = process_pages + oom_score_adj * totalpages/1000`

2、/proc/sysrq-trigger 参数用途

```
echo "m" > /proc/sysrq-trigger      # 导出内存分配的信息 （使用 /var/log/message 查看）
```

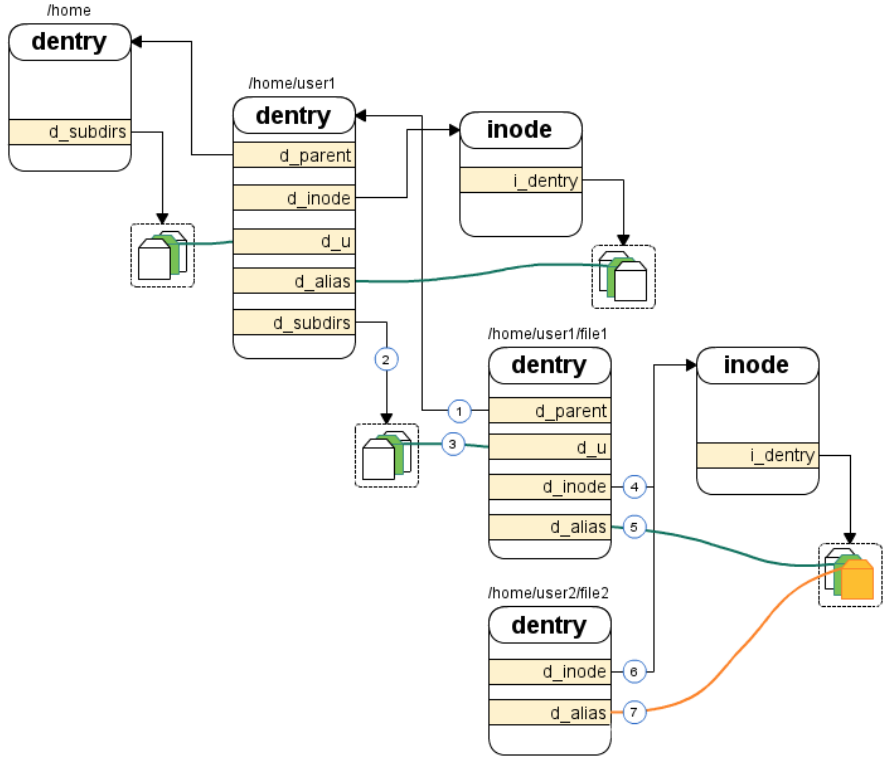
命令	功能
b	立即重新启动系统，而无需同步或卸载磁盘
c	通过 `NULL` 指针取消引用来执行系统崩溃
d	列出系统中所有被持有的锁
e	向系统中除 `init` 外的所有进程发出 `SIGTERM` 信号
f	调用 `oom killer` 杀死内存消耗进程，也会存在没有进程会被杀死
g	使能 `kgdb`（内核调试器）
i	发送 `SIGKILL` 到所有进程，初始化除外
j	强制 "仅解冻"；被 `FIFREEZE ioctl` 冻结的文件系统
k	安全访问密钥（`SAK`）杀死当前虚拟控制台上的所有程序
l	显示所有活动 `CPU` 的堆栈回溯
m	将当前的内存信息转储到您的控制台。
n	用于使RT任务变得更好
o	立即关闭计算机
p	将当前的寄存器和标志转储到您的控制台。
q	将按 `CPU` 转储所有配备的 `hrtimer` 的列表以及有关所有 `clockevent` 设备的详细信息。
r	关闭键盘原始模式并将其设置为 `XLATE`
s	重新挂载所有文件系统
t	将当前任务列表及其信息转储到控制台
u	尝试以只读方式重新挂载所有已挂载的文件系统。
v	强制还原帧缓冲控制台
w	转储处于不间断（阻塞）状态的任务
x	由 `ppc / powerpc` 平台上的 `xmon` 接口使用；在 `sparc64` 上显示全局 `PMU` 寄存器；在 `MIPS` 上转储所有 `TLB` 条目
y	显示全局 `CPU` 寄存器 【特定于SPARC-64】
z	转储 `ftrace` 缓冲区
0-9	设置控制台日志级别，控制将哪些内核消息打印到控制台

3、dentry 销毁机制

销毁一个决定释放的目录项，首先检查目录项是否关联了一个 `inode`，并尝试获取 `inode` 锁，如果无法立即获取则转到慢路径处理。然后，它获取父目录项的锁，如果无法立即获取则通过 `__lock_parent` 函数进行获取。接下来，调用 `__dentry_kill` 函数销毁目录项，并返回其父目录项。最后根据目录项的锁引用计数以及是否需要保留目录项的判断，决定是否销毁目录项，并返回其父目录项。

`dentry p_parent` 持锁的情况下，`dentry` 不会被立即销毁。

> `dentry` 就是 "目录项" 保存着诸如文件名、路径名等信息；`inode` 是索引点，保存具体文件的数据，比如权限、修改日期、设备号（如果是设备文件的话）等等。文件系统中所有文件（目录也是一种特殊的文件）都必有一个 `inode` 与之对应，而每个 `inode` 也至少有一个 `dentry` 与之对应（也可能有多个，比如硬链接）
>
> **`dentry` 和 `dentry` 建立关系：**
>
> `d_subdirs` 存储目录项子目录链表头。子目录 `dentry`（`/home/user1/file1`）的 `d_parent` 指向父目录 `dentry`（`/home/user1`），并将自己的 `d_u` 插入到 `d_subdirs` 链表。这样文件的上下关系就建立好了。
>
> **`dentry` 和 `inode` 建立关系：**
>
> `dentry` 中有 `d_inode` 指针，指向 `inode` 结构体，就是该文件的索引节点。`inode` 中的 `i_dentry` 是用于管理与该 `inode` 相关的目录项。`dentry` 将 `d_alias` 插入到 `i_dentry` 链表，这样就建立起了联系。
>
> **硬链接：**
>
> `file1` 和 `file2` 的 `dentry` 都指向 `indoe`，且 `d_alias` 都加入到 `inode` 的 `indoe` 的 `i_dentry`，目录也是准许硬链接的，但是不允许普通用户创建目录的硬链接。



图片地址：https://blog.csdn.net/jinking01/article/details/105682389

```
/*
```

```
 * Finish off a dentry we've decided to kill.
 * dentry->d_lock must be held, returns with it unlocked.
 * Returns dentry requiring refcount drop, or NULL if we're done.
 */
/*
* dentry_kill 函数用于销毁一个被决定释放的目录项 dentry。
* 调用者必须在进入函数时已经持有 dentry->d_lock 锁，并且函数在返回时会释放这个锁。
* 返回需要减少引用计数的目录项，如果不再需要返回NULL。
*/
static struct dentry *dentry_kill(struct dentry *dentry)
    __releases(dentry->d_lock)
{
    struct inode *inode = dentry->d_inode;
    struct dentry *parent = NULL;
    // 如果目录项中有关联的 inode，并且无法立即获取该 inode 的锁，则转到慢路径处理
    if (inode && unlikely(!spin_trylock(&inode->i_lock)))
        goto slow_positive;
    // 如果目录项不是根目录项
    if (!IS_ROOT(dentry)) {
        parent = dentry->d_parent;
        // 如果无法立即获取父目录项的锁（竞争激烈多线程、中断上下文等），则通过
__lock_parent 函数进行获取
        if (unlikely(!spin_trylock(&parent->d_lock))) {
            parent = __lock_parent(dentry); // 尝试获取父目录项的锁
            // 如果目录项关联的 inode 存在或者目录项没有关联的 inode，则获取锁成功，进入该
if 条件很大
            if (likely(inode || !dentry->d_inode))
                goto got_locks;
            /* negative that became positive */
            if (parent)
                spin_unlock(&parent->d_lock);  // 解除父目录项
            inode = dentry->d_inode; // 更新 inode 变量
            goto slow_positive; // 转到慢路径处理
        }
    }
    // 调用 __dentry_kill 函数销毁目录项，并返回其父目录项
    __dentry_kill(dentry);
    return parent;

slow_positive:
    spin_unlock(&dentry->d_lock); // 解锁当前目录项
    spin_lock(&inode->i_lock); // 锁定 inode
    spin_lock(&dentry->d_lock); // 再次锁定当前目录项
    parent = lock_parent(dentry); // 锁定父目录项
got_locks:
    // 如果目录项的锁引用计数不为 1，则将其减少
    if (unlikely(dentry->d_lockref.count != 1)) {
        dentry->d_lockref.count--; // 减少锁引用计数
    } else if (likely(!retain_dentry(dentry))) {
        // 如果目录项的锁引用计数为 1 且不再需要引用目录项，则销毁目录项并返回其父目录项
        __dentry_kill(dentry);
        return parent;
    }

    /* we are keeping it, after all */
    // 如果目录项关联了 inode，则解锁该 inode
    if (inode)
        spin_unlock(&inode->i_lock);
```

```
    // 如果存在父目录项，则解锁该父目录项
    if (parent)
        spin_unlock(&parent->d_lock);
    // 解锁目录项自身
    spin_unlock(&dentry->d_lock);
    return NULL; // 返回 NULL，表示不再需要减少引用计数
}
```

问题现场

【问题复现步骤】

同时执行下述两个脚本，可快速复现问题

```
#!/bin/bash

### file.sh 循环创建文件###
for k in `seq 10000 20000`; do
    echo "========$k========"
    mkdir -p /run/test
    for i in `seq 1 20000`; do
        echo cccccc > /run/test/$i
    done
    rm -rf /run/test
    sleep 1
done
```

```
#!/bin/bash

### oom.sh ：标识当前进程不可被 oom killer 终止，同时调用 oom killer 杀死内存消耗进程 ###
echo -1000 > "/proc/$$/oom_score_adj"
for i in `seq 10000 10000000`; do
    echo "=========$i========="
    echo f > /proc/sysrq-trigger
    sleep 2
done
```

【执行结果】

执行一段时候后，系统发生 `crash` 复位

分析定位

1、查看 `vmcore-dmesg.txt` 信息

```
# 指出当前故障类型为内核空指针引用故障
[ 4993.027672] BUG: kernel NULL pointer dereference, address: 0000000000000050
[ 4993.027673] #PF: supervisor read access in kernel mode
[ 4993.027674] #PF: error_code(0x0000) - not-present page
[ 4993.027675] PGD 11ae2e067 P4D 0
[ 4993.027677] Oops: 0000 [#1] SMP NOPTI
[ 4993.027678] kernel fault(0x1) notification starting on CPU 2
[ 4993.027681] kernel fault(0x1) notification finished on CPU 2
```

```
[ 4993.027685] CPU: 2 PID: 56829 Comm: kworker/2:3 Kdump: loaded Not tainted
5.10.0-136.75.0.155.u112.fos23.x86_64 #1
[ 4993.027686] Hardware name: QEMU Standard PC (Q35 + ICH9, 2009), BIOS rel-
1.12.1-0-ga5cab58e9a3f-prebuilt.qemu.org 04/01/2014
[ 4993.027690] Workqueue: events moom_callback
[ 4993.027694] RIP: 0010:do_show_dentry_tree+0x161/0x355
[ 4993.027696] Code: b6 e8 58 5e ff ff 48 ff 05 19 dd d2 01 48 8b 0d 12 dd d2 01
be e8 03 00 00 31 d2 48 89 c8 48 f7 f6 48 85 d2 74 0a 49 8b 47 30 <4c> 03 60 50
eb 5e 48 c7 c2 78 68 b6 b6 be 00 04 00 00 48 c7 c7 60
[ 4993.027697] RSP: 0018:ff6a361b00863cd8 EFLAGS: 00010202
[ 4993.027698] RAX: 0000000000000000 RBX: 0000000000000001 RCX: 000000000000000b
[ 4993.027699] RDX: 000000000000000b RSI: 00000000000003e8 RDI: ff43684dbfb20610
[ 4993.027700] RBP: ff43684c76244d80 R08: 0000000000000000 R09: ff6a361b00863b20
[ 4993.027700] R10: ff6a361b00863b18 R11: ffffffffb7385b20 R12: 0000000000000000
[ 4993.027701] R13: 0000000000000008 R14: ff43684c76244dd8 R15: ff43684d44cf57a0
[ 4993.027705] FS:  0000000000000000(0000) GS:ff43684dbfb00000(0000)
knlGS:0000000000000000
[ 4993.027706] CS:  0010 DS: 0000 ES: 0000 CR0: 0000000080050033
[ 4993.027707] CR2: 0000000000000050 CR3: 000000011ac06006 CR4: 0000000000771ee0
[ 4993.027707] DR0: 0000000000000000 DR1: 0000000000000000 DR2: 0000000000000000
[ 4993.027708] DR3: 0000000000000000 DR6: 00000000fffe0ff0 DR7: 0000000000000400
[ 4993.027709] PKRU: 55555554
[ 4993.027709] Call Trace:
[ 4993.027713]  do_show_dentry_tree+0x27f/0x355
[ 4993.027717]  ? is_empty_dir+0x120/0x120
[ 4993.027718]  show_file_info_by_name.cold+0xa0/0xc8
[ 4993.027721]  iterate_supers+0x8c/0x100
[ 4993.027722]  show_memfs_info.cold+0x20/0x2f
[ 4993.027724]  oom_show_debug_info.cold+0x9c/0xc2
[ 4993.027727]  out_of_memory+0x2fb/0x360
[ 4993.027728]  moom_callback+0x79/0xb0
[ 4993.027731]  process_one_work+0x1ad/0x350
[ 4993.027732]  worker_thread+0x49/0x310
[ 4993.027733]  ? rescuer_thread+0x3b0/0x3b0
[ 4993.027736]  kthread+0xfb/0x140
[ 4993.027737]  ? kthread_park+0x90/0x90
[ 4993.027740]  ret_from_fork+0x1f/0x30
[ 4993.027741] Modules linked in: nft_fib_inet nft_fib_ipv4 nft_fib_ipv6 nft_fib
nft_reject_inet nf_reject_ipv4 nf_reject_ipv6 nft_reject nft_ct nft_chain_nat
nf_tables ebtable_nat ebtable_broute ip6table_nat ip6table_mangle ip6table_raw
ip6table_security iptable_nat nf_nat nf_conntrack nf_defrag_ipv6 nf_defrag_ipv4
libcrc32c iptable_mangle iptable_raw iptable_security ip_set rfkill nfnetlink
ebtable_filter ebtables ip6table_filter ip6_tables iptable_filter ip_tables
intel_rapl_msr intel_rapl_common intel_uncore_frequency_common nfit libnvdimm
iTCO_wdt iTCO_vendor_support joydev lpc_ich i2c_i801 pcspkr i2c_smbus
virtio_balloon fuse ext4 mbcache jbd2 crct10dif_pclmul crc32_pclmul crc32c_intel
virtio_gpu virtio_dma_buf drm_kms_helper syscopyarea sysfillrect sysimgblt
fb_sys_fops cec drm ahci libahci libata ghash_clmulni_intel serio_raw virtio_net
virtio_blk virtio_console net_failover failover virtio_scsi dm_mirror
dm_region_hash dm_log dm_mod
[ 4993.027788] CR2: 0000000000000050
[ 4993.027789] ---[ end trace 8ce82bbfa09178a1 ]---
[ 4993.027791] RIP: 0010:do_show_dentry_tree+0x161/0x355
[ 4993.027792] Code: b6 e8 58 5e ff ff 48 ff 05 19 dd d2 01 48 8b 0d 12 dd d2 01
be e8 03 00 00 31 d2 48 89 c8 48 f7 f6 48 85 d2 74 0a 49 8b 47 30 <4c> 03 60 50
eb 5e 48 c7 c2 78 68 b6 b6 be 00 04 00 00 48 c7 c7 60
[ 4993.027793] RSP: 0018:ff6a361b00863cd8 EFLAGS: 00010202
[ 4993.027794] RAX: 0000000000000000 RBX: 0000000000000001 RCX: 000000000000000b
```

```
[ 4993.027795] RDX: 000000000000000b RSI: 00000000000003e8 RDI: ff43684dbfb20610
[ 4993.027796] RBP: ff43684c76244d80 R08: 0000000000000000 R09: ff6a361b00863b20
[ 4993.027796] R10: ff6a361b00863b18 R11: ffffffffb7385b20 R12: 0000000000000000
[ 4993.027797] R13: 0000000000000008 R14: ff43684c76244dd8 R15: ff43684d44cf57a0
[ 4993.027798] FS:  0000000000000000(0000) GS:ff43684dbfb00000(0000)
knlGS:0000000000000000
[ 4993.027799] CS:  0010 DS: 0000 ES: 0000 CR0: 0000000080050033
[ 4993.027800] CR2: 0000000000000050 CR3: 000000011ac06006 CR4: 0000000000771ee0
[ 4993.027800] DR0: 0000000000000000 DR1: 0000000000000000 DR2: 0000000000000000
[ 4993.027801] DR3: 0000000000000000 DR6: 00000000fffe0ff0 DR7: 0000000000000400
[ 4993.027802] PKRU: 55555554
[ 4993.027802] Kernel panic - not syncing: Fatal exception
[ 4993.027803] kernel fault(0x5) notification starting on CPU 2
[ 4993.027806] kernel fault(0x5) notification finished on CPU 2
[ 4993.028838] Kernel Offset: 0x34800000 from 0xffffffff81000000 (relocation
range: 0xffffffff80000000-0xffffffffbfffffff)
[ 4993.029615] kexec: Bye!
```

2、解析 `vmcore` 文件，并收集故障现场信息

```
$ crash vmcore vmlinux
$ crash> bt
PID: 56829    TASK: ff43684d40253080  CPU: 2    COMMAND: "kworker/2:3"
 #0 [ff6a361b00863aa8] panic at ffffffffb6262cce
 #1 [ff6a361b00863b48] no_context at ffffffffb587d79c
 #2 [ff6a361b00863b80] __bad_area_nosemaphore at ffffffffb587d8a2
 #3 [ff6a361b00863bc8] exc_page_fault at ffffffffb62ad55c
 #4 [ff6a361b00863c20] asm_exc_page_fault at ffffffffb6400afe
    [exception RIP: do_show_dentry_tree+353]
    RIP: ffffffffb6272a4c  RSP: ff6a361b00863cd8  RFLAGS: 00010202
    RAX: 0000000000000000  RBX: 0000000000000001  RCX: 000000000000000b
    RDX: 000000000000000b  RSI: 00000000000003e8  RDI: ff43684dbfb20610
    RBP: ff43684c76244d80   R8: 0000000000000000   R9: ff6a361b00863b20
    R10: ff6a361b00863b18  R11: ffffffffb7385b20  R12: 0000000000000000
    R13: 0000000000000008  R14: ff43684c76244dd8  R15: ff43684d44cf57a0
    ORIG_RAX: ffffffffffffffff  CS: 0010  SS: 0018
 #5 [ff6a361b00863d28] do_show_dentry_tree at ffffffffb6272b6a
 #6 [ff6a361b00863d80] show_file_info_by_name.cold at ffffffffb6272ce0
 #7 [ff6a361b00863db0] iterate_supers at ffffffffb5b92a1c
 #8 [ff6a361b00863de8] show_memfs_info.cold at ffffffffb6272ea2
 #9 [ff6a361b00863df0] oom_show_debug_info.cold at ffffffffb6272f4d
#10 [ff6a361b00863e10] out_of_memory at ffffffffb5aae6db
#11 [ff6a361b00863e48] moom_callback at ffffffffb5ec5b89
#12 [ff6a361b00863e98] process_one_work at ffffffffb590975d
#13 [ff6a361b00863ed8] worker_thread at ffffffffb5909cf9
#14 [ff6a361b00863f10] kthread at ffffffffb590ea3b
#15 [ff6a361b00863f50] ret_from_fork at ffffffffb580356f

######
# 故障点在 RIP: do_show_dentry_tree+353，查看故障点上下文信息
######
$ crash> dis -s (do_show_dentry_tree+353)
FILE: mm/oom_enhance_info.c
LINE: 272

  267                                     }
```

```
  268                                     printk(KERN_CONT "%s", str_print);
  269                                     touch_nmi_watchdog();
  270                                     mdelay(10);
  271                             }
* 272                             dir_size += dentry->d_inode->i_size;     # 故障点代
码，发生空指针引用
  273                     }
  274             }
  275             /* second iterate list for showing subdirs after showing files
*/
  276             list_for_each_entry(dentry, &tree->d_subdirs, d_child) {
  277                     if (dentry == NULL || dentry->d_inode == NULL) {
... ... ... ... ... ...


######
# 反汇编故障点代码，明确是谁（dentry / d_inode）引用了空指针
######
$ crash> dis -l (do_show_dentry_tree+353)
/usr/src/debug/kernel-5.10.0-136.75.0.155.u112.fos23.x86_64/linux-5.10.0-
136.75.0.155.u112.fos23.x86_64/mm/oom_enhance_info.c: 272
0xffffffffb6272a4c <do_show_dentry_tree+353>:   add    0x50(%rax),%r12
# 0xffffffffb6272a4c <do_show_dentry_tree+353>:   add    0x50(%rax),%r12 解析：
# 1、对应于源文件 oom_enhance_info.c 中的第 272
# 2、指令的内存地址是 0xffffffffb6272a4c，在函数 do_show_dentry_tree 中相对于函数开始处
的偏移量为 353 字节
# 3、add 0x50(%rax), %r12 的含义是将内存地址 (%rax + 0x50) 处的值加到寄存器 %r12 中，计
算结果存储在 %r12 中
#
# 通过反汇编结果可以说明  d_inode 保存在 %rax 寄存器， i_size 为 %rax 寄存器中地址偏移
0x50 处， dir_size 保存 %r12 中
# 此时我们想获取 dentry 变量所在的寄存器地址，获取 dentry 的值，以便进一步确认 d_inode 是否
存在空指针引用。
# $ objdump -Sd vmlinux > oops_fault.txt
# $ vim oops_fault.txt   # 查看 do_show_dentry_tree 函数对应的汇编代码
# ffffffff81a728eb <do_show_dentry_tree>:
# {
# ... ... ... ... ... ...
#                         dir_size += dentry->d_inode->i_size;
#                         # %r15 寄存器保存 dentry 变量地址，偏移 0x30 为 d_inode 变
量，将其保存至 %rax 寄存器
# ffffffff81a72a48:       49 8b 47 30             mov    0x30(%r15),%rax
#                     # %rax 寄存器保存 d_inode 变量地址，偏移 0x50 为 i_size 变量
# ffffffff81a72a4c:       4c 03 60 50             add    0x50(%rax),%r12
# ffffffff81a72a50:       eb 5e                   jmp    ffffffff81a72ab0
<do_show_dentry_tree+0x1c5>
# ... ... ... ... ... ...
# }

######
# 由上述反汇编可得 dentry 变量保存在 %r15 寄存器，验证 d_inode 是否为空
######
$ crash> struct dentry -x ff43684d44cf57a0   # ff43684d44cf57a0 由上述 bt 可以查到
%r15 保存值
struct dentry {
    ... ... ... ... ... ...
    d_inode = 0x0,  # 可以观察到 d_inode 值为 NULL
    ... ... ... ... ... ...
```

```
}
```

结果推测：为什么 `d_inode` 为空?

1、测试用例执行约 20 min，问题复现。说明故障在特定场景下会诱发。

2、`dir_size += dentry->d_inode->i_size;` 过程 `d_inode` 被置空，逻辑上此处不会是空指针。

3、由于并行执行过程中 `d_inode` 可被多个线程访问，存在 A 在执行 `d_inode->i_size` 计数前，B 将 `d_inode` 释放。

4、结合测试用例 `file.sh` 存在删除目录行为，很可能在遍历 `dentry->d_inode` 过程中，该目录被删除掉，导致 `d_inode` 被释放。

3、解析代码，分析问题成因

```
// 代码处理逻辑如下（伪代码）：
do_show_dentry_tree(directory):
    dir_size = 0

    lock(dentry->d_lock)                          ---> 对 dentry->d_lock 加锁，防止竞态情况（原处理逻辑）
    for each item in directory:
        if item is a file:
            dir_size += getSize(item)
    for each item in directory:
        if item is a sub_directory:
            do_show_dentry_tree(item)
    unlock(dentry->d_lock)                        ---> 对 dentry->d_lock 解锁，处理结束 （原处理逻辑）
```

漏洞：在进行目录的一层遍历时，获取遍历到的文件信息前，需要对 `dentry->d_lock` 加锁，以防止其他进程对 `d_inode` 数值修改。但此处加锁是针对父 `dentry` 加锁，循环遍历子 `dentry` 并没有加锁，子 `dentry` 存在被其他进程修改的风险。

4、修改方案

```
// 代码处理逻辑如下（伪代码）：
do_show_dentry_tree(directory):
    dir_size = 0

    for each item in directory:
        lock(dentry->d_lock)                      ---> 对 dentry->d_lock 加锁，防止竞态情况（现处理逻辑）
        if item is a file:
            dir_size += getSize(item)
        unlock(dentry->d_lock)                    ---> 对 dentry->d_lock 解锁，处理结束 （现处理逻辑）
    for each item in directory:
        lock(dentry->d_lock)        ---> 对 dentry 加锁，在遍历 d_subdirs 时，d_subdirs 的 dentry 结构体不会被销毁
        if item is a directory:
            do_show_dentry_tree(item)
         unlock(dentry->d_lock)      ---> 递归操作完，对 dentry 解锁。（ 阅读背景知识 - dentry销毁机制）
```

通过上述处理，加锁的 `dentry->d_lock` 是保护当前 `directory` 的 `d_node` 不会被清理或者修改，防止本故障中 `d_node` 被清理的场景。对 `d_parent` 加锁，防止遍历子目录时 `denttry` 被清理掉。

`Oom_enhance` 特性涉及的其他代码处理逻辑也同步做了修改，测试回归无问题。