一个内存泄露问题的定位

秦新良

2014年 3月 12日

目录

1 问题现象

USCDBV100R007C10版本TR5在即,性能、稳定性、安全测试全面启动,测试部的一套环境上连续业务呼叫一整天后,出现网元下的所有PGW进程从凌晨2点到5点陆续全部重启。

2 问题定位

2.1 初步分析

通过分析进程退出时产生的黑匣子发现,重启是因为当时PGW进程的虚拟内存已达到3G,后续的业务处理由于内存申请失败而导致进程异常退出。

重启的原因已非常明显,内存泄露导致。但是是哪种业务场景导致的内存泄露呢?这是整个定位过程中的重中之重。

2.2 问题重现

出现问题的环境是BEC组网的环境,业务运行在SP上,跑的是UPA的指令。这些指令的特殊性在于,客户端过来的一条UPA指令,会被PGW通过三方适配拆成一条UPA指令和一条HLR指令,分别发给远端UPA和HLR的SP上执行。除此之外,该环境上的全局数据刷新开关是打开的,PGW每隔一小时刷新一次全局数据。

通过分析系统每分钟的top日志发现,业务进程的虚拟内存每隔一小时上涨一次,与全局数据刷新的周期相吻合,所以重现及排查问题的重点就放在了全局数据刷新上[1]。

不幸的是,按照全局数据刷新这个思路来重现问题,问题一直没有重现。但由于该场景是重点怀疑对象,所以我们几个人还是想尽各种办法来排查、构造全局数据的异常场景,期待问题的重现。

经过了一天一夜后,问题还是没能稳定的重现。虽然运行过程中,进程的内存会有一些浮动,但整体来看是一个稳定的状态。最后没办法,只能换个方向,通过跑UPA的业务指令试着重现问题。

持续执行UPA的命令,发现内存是有一些增涨,但非常缓慢,而且也不是持续增涨。如果要等到这种场景把内存泄露完来确认问题,估计一天又要过去了。但看到内存有一些增涨,总归是给定

[1]还有一个原因是全局数据刷新在历史的版本中就有内存相关的疑难杂症,在171版本刚刚解决,而且在171版本中又加了一个按FE刷新,所以它的疑点最多。

位问题提供了一个新的方向,先只能按着这个方向继续往下定位。

2.3 问题定位

在重现问题的过程中,查了一些tcmalloc内存管理相关资料,发现tcmalloc在申请内存的时候,是一次申请大片内存,后续应用程序有内存申请的时候,直接从这块大内存中分配,如果当前申请的内存不够的话,才会向系统申请内存。通过分析malloc_stat的结果,验证了这一结论。既然这样,那如果是UPA的指令存在内存泄露,外在表现确实不会是连续的内存增涨,而是伴随着tcmalloc已申请的内存不足时再次向系统申请时才会出现内存上涨。有了这个分析结论后,问题重现及排查的重点就放在了UPA指令上。

最后通过valgrind将PGW拉起并执行UPA的指令,运行一段时间后,将进程退出,生成valgrind的报告如下:

```
==4401== Memcheck, a memory error detector
==4401== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
==4401== Using Valgrind-3.9.0 and LibVEX; rerun with -h for copyright info
==4401== Command: ./test
==4401==
==4401==
==4401== HEAP SUMMARY:
==4401==     in use at exit: 38 bytes in 1 blocks
==4401==   total heap usage: 2 allocs, 1 frees, 54 bytes allocated
==4401==
==4401== 38 bytes in 1 blocks are definitely lost in loss record 1 of 1
==4401==    at 0x4C2C3FB: operator new(unsigned long) (vg_replace_malloc.c:319)
==4401==    by 0x4EF0108: std::string::_Rep::_S_create(unsigned long, unsigned long, std::allocator<char> const&) (in /usr/lib/x86_64-linux-gnu/libstdc++.so.6.0.17)
==4401==    by 0x4EF0302: std::string::_M_mutate(unsigned long, unsigned long, unsigned long) (in /usr/lib/x86_64-linux-gnu/libstdc++.so.6.0.17)
==4401==    by 0x4EF049B: std::string::_M_replace_safe(unsigned long, unsigned long, char const*, unsigned long) (in /usr/lib/x86_64-linux-gnu/libstdc++.so.6.0.17)
==4401==    by 0x4009FA: D1::D1() (test.cpp:16)
==4401==    by 0x40093B: mem_leak_test() (test.cpp:27)
==4401==    by 0x400980: main (test.cpp:35)
==4401==
==4401== LEAK SUMMARY:
==4401==    definitely lost: 38 bytes in 1 blocks
==4401==    indirectly lost: 0 bytes in 0 blocks
==4401==      possibly lost: 0 bytes in 0 blocks
==4401==    still reachable: 0 bytes in 0 blocks
==4401==         suppressed: 0 bytes in 0 blocks
==4401==
==4401== For counts of detected and suppressed errors, rerun with: -v
==4401== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 2 from 2)
```

从报告中的LEAK SUMMARY可以看出,进程中存在38个字节(definitely lost)的内存泄露。继续分析报告,报告中指出,函数mem_leak_test()中存在内存泄露。函数mem_leak_test()的定义如代码[2]1所示。

```
1  #include <string>
2
3  class Base
4  {
5    public:
6      virtual void fun1() {}
7      virtual void fun2() {}
8
9    private:
10     std::string m_strData1;
11 };
12
13 class D1: public Base
14 {
15   public:
16     D1(){ m_strData2 = "Hello, world."; }
17
18     virtual void fun1() {}
19     virtual void fun2() {}
20
21   private:
22     std::string m_strData2;
23 };
24
25 void mem_leak_test()
26 {
27     Base *pObj = new D1;
28
29     delete pObj;
30     return;
31 }
32
33 int main(int argc, char *argv[])
34 {
35     mem_leak_test();
36
37     return 0;
38 }
```

代码 1: 示例代码

从代码1中可以看出,在函数mem_leak_test()中,申明了基类Base的指针pObj,其指向的是派生类D1的对象,然后通过pObj将动态申请的对象释放。

接着再看类Base和D1的定义,发现基类Base的析构函数未显示定义,那么编译器会自动为Base生成默认的析构函数,但默认生成的析构是非virtual的,这就导致了通过基类指针pObj释放派生类的对象时,派生类对象的析构函数不会被调用,如果派生类中有动态的内存申请的话,就会出现内存泄露。具体来说就是,D1的析构函数不会被调用,那么D1的成员m_strData2的析构函

[2]这段代码只是用来说明问题,实际的业务逻辑比这复杂的多,而且内存也不是在同一函数内申请和释放。

数就不会被调用,最终导致 m_strData2 里动态申请的内存得不到释放。

3 修改方案

修改方案很简单,将基类的析构函数定义为虚函数即可,如代码2所示。

```
class Base
{
  public:
    virtual ~Base() {}
    virtual void fun1() {}
    virtual void fun2() {}

  private:
   std::string m_strData1;
};
```

代码 2: 修改后的代码

基类 Base 的析构函数申明为虚函数后,Base 和 D1 的内存布局如图1所示。

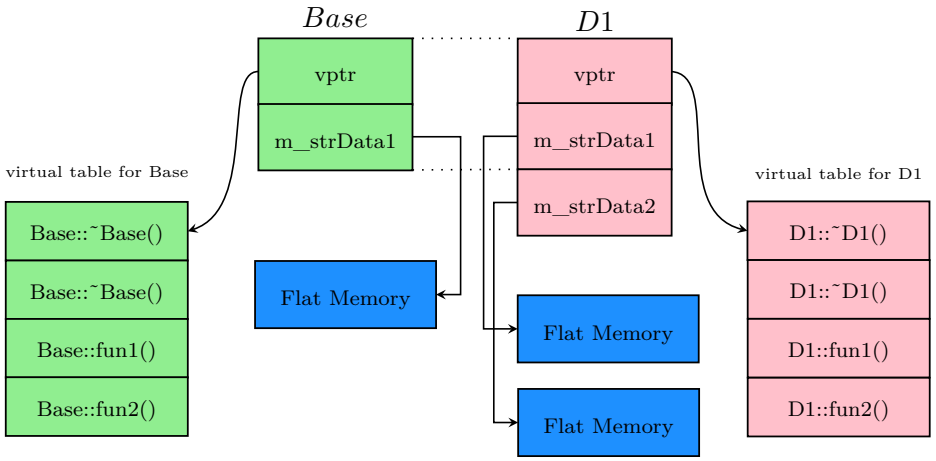

图 1: Base 和 D1 的内存布局

我们出问题的地方就在通过基类指针释放派生类的对象时,由于派生类的析构函数没有被调用,导致数据成员 m_strData2 指向的内存没有被释放。

4 问题总结

虽然我们知道动态内存的申请和释放一定要成对,但对该问题来说有一个地方比较隐晦,就是在派生类中并没有显示的动态申请内存,而是成员变量 string 在动态申请,这也导致了在代码检视或问题排查的时候容易被忽略,最终将问题遗留下来。

5 virtual 背后的原理

关于 C++ 的虚函数,之前也写过一篇文章分析过,参见这里。这里再谈一下,为什么基类的析构函数定义为虚函数,通过基类指针就可以释放派生类的对象且派生类的析构函数会被正确的调用,而非虚函数则达不到这一效果。

首先来看 Base 的析构函数为非虚函数的情况下,函数 mem_leak_test() 反汇编后的结果,如代码3所示:

```
1  0804871c <_Z13mem_leak_testv>:
2   804871c:        push   %ebp
3   804871d:        mov    %esp,%ebp
4   804871f:        push   %esi
5   8048720:        push   %ebx
6   8048721:        sub    $0x20,%esp
7   8048724:        movl   $0xc,(%esp)
8   804872b:        call   80485f0 <_Znwj@plt>
9   8048730:        mov    %eax,%ebx
10  8048732:        mov    %ebx,(%esp)
11  8048735:        call   80487d0 <_ZN2D1C1Ev>
12  804873a:        mov    %ebx,-0xc(%ebp)
13  804873d:        mov    -0xc(%ebp),%ebx
14  8048740:        test   %ebx,%ebx
15  8048742:        je     804876a <_Z13mem_leak_testv+0x4e>
16  8048744:        mov    %ebx,(%esp)
17  8048747:        call   80487b0 <_ZN4BaseD1Ev>
18  804874c:        mov    %ebx,(%esp)
19  804874f:        call   80485c0 <_ZdlPv@plt>
20  804876a:        nop
21  804876b:        add    $0x20,%esp
22  804876e:        pop    %ebx
23  804876f:        pop    %esi
24  8048770:        pop    %ebp
25  8048771:        ret
```

代码 3: 非虚析构

代码3中的第 8 行调用 new 来申请内存,在 11 行调用类 D1 的构造函数对新申请的对象初始化,第 17 行调用 Base 的析构函数,第 19 行调用 delete 释放内存。可见,在整个函数的执行过程中,并没有调用 D1 的析构函数。

将 Base 的析构函数定义为虚函数后,函数 mem_leak_test() 反汇编后的结果如代码4所示:

```
1  0804871c <_Z13mem_leak_testv>:
2   804871c:        push   %ebp
3   804871d:        mov    %esp,%ebp
4   804871f:        push   %esi
5   8048720:        push   %ebx
6   8048721:        sub    $0x20,%esp
7   8048724:        movl   $0xc,(%esp)
8   804872b:        call   80485f0 <_Znwj@plt>
9   8048730:        mov    %eax,%ebx
```

```
8048732:        mov    %ebx,(%esp)
8048735:        call   8048806 <_ZN2D1C1Ev>
804873a:        mov    %ebx,-0xc(%ebp)
804873d:        cmpl   $0x0,-0xc(%ebp)
8048741:        je     804876b <_Z13mem_leak_testv+0x4f>
8048743:        mov    -0xc(%ebp),%eax
8048746:        mov    (%eax),%eax
8048748:        add    $0x4,%eax
804874b:        mov    (%eax),%eax
804874d:        mov    -0xc(%ebp),%edx
8048750:        mov    %edx,(%esp)
8048753:        call   *%eax
8048755:        jmp    804876b <_Z13mem_leak_testv+0x4f>
8048757:        mov    %eax,%esi
8048759:        mov    %ebx,(%esp)
804875c:        call   80485c0 <_ZdlPv@plt>
8048761:        mov    %esi,%eax
8048763:        mov    %eax,(%esp)
8048766:        call   8048610 <_Unwind_Resume@plt>
804876b:        nop
804876c:        add    $0x20,%esp
804876f:        pop    %ebx
8048770:        pop    %esi
8048771:        pop    %ebp
8048772:        ret
```

代码 4: 虚析构

反汇编后代码中的第 8 行调用 new 申请内存,第 11 行调用类 D1 的构造函数对新申请的对象初始化,第 25 行调用 delete 释放内存,这和非虚析构函数的情况是一样的。不同之处在于第 21 行,这里不再是调用 Base 的析构函数,而是 call *%eax。call *%eax 表示什么意思呢,见代码5。

```
0804871c <_Z13mem_leak_testv>:
push   %ebp
mov    %esp,%ebp
push   %esi
push   %ebx
sub    $0x20,%esp
movl   $0xc,(%esp)
call   80485f0 <_Znwj@plt>   ; call new to allocate mem
mov    %eax,%ebx             ; save the return value of new
                             ; to register ebx
mov    %ebx,(%esp)           ; push "this" pointer on stack
                             ; as the first para to D1::D1()
call   8048806 <_ZN2D1C1Ev> ; call the constructor of D1
mov    %ebx,-0xc(%ebp)       ; save the "this" pointer on stack
cmpl   $0x0,-0xc(%ebp)       ; check the pointer
je     804876b <_Z13mem_leak_testv+0x4f>
mov    -0xc(%ebp),%eax       ; save the new object pointer to
                             ; register eax
mov    (%eax),%eax           ; save the virtual table address eax
add    $0x4,%eax             ; virtual talbe address + 4
mov    (%eax),%eax           ; get the second function address in
                             ; virtual table
mov    -0xc(%ebp),%edx       ; "this" pointer to register edx
mov    %edx,(%esp)           ; push "this" pointer on stack
call   *%eax                 ; call the second function in virtual table
```

```
26 jmp    804876b <_Z13mem_leak_testv+0x4f>
27 mov    %eax,%esi
28 mov    %ebx,(%esp)
29 call   80485c0 <_ZdlPv@plt>
30 mov    %esi,%eax
31 mov    %eax,(%esp)
```

代码 5: 代码注释

从代码5的注释中可以看出,call *%eax 实际上是调用了类 D1 虚函数表中的第二个函数。D1 虚函数表中的第二个函数即 D1::~D1(),如图1所示[3],这样就达到了通过基类指针正确的释放派生类对象的目的。

[3]关于虚函数表中为什么有两个虚析构函数,参见这里。