**【转载】【堆调试工具】pageheap的使用和原理分析&Linux下相似的功能实现**

(2011-07-27 14:15:15)

[*[http://simg.sinajs.cn/blog7style/images/common/sg_trans.gif](javascript:;)转载▼*](javascript:;)

|  |  |
| --- | --- |
| 标签：  [**pageheap**](http://search.sina.com.cn/?c=blog&q=pageheap&by=tag)  [**杂谈**](http://search.sina.com.cn/?c=blog&q=%D4%D3%CC%B8&by=tag) |  |

偶然的机会，为了找一个内存方面的崩溃，接触到了pageheap，感觉有些情况下还是挺好用的，收集了一些资料，分享一下。

Ps：

1. win7下貌似pageheap不能用，也懒得找能用的版本了，有兴趣的可以自己研究下。

2. 不要在被调试的程序前加路径，直接到程序当前目录执行命令。

**【转载1】**[**堆调试工具——pageheap的使用和原理分析**](http://blog.csdn.net/xiaohyy/article/details/4174493)

今天调试一个bug，用pageheap解决，在此记录一下。

bug症状如下：  
1：不确定性崩溃，用vs调试启动每次崩溃地点都在crt分配或者释放堆的位置  
2：崩溃时vs看到的调用栈可能不同  
3：output输出HEAP: Free Heap block 388c58 modified at 388c88 after it was freed

问题分析：  
根据vs的输出，确定问题是在一块堆上分配的内存在释放后被改写了。由于CRT只能在下次做堆操作检查时才会暴露出问题，所以程序崩溃的调用栈是不确定的。  
折腾了2个小时后，启用pageheap缩小了程序出错到崩溃之间的距离，解决了问题。过程如下：  
1:启动pageheap  
pageheap /enable mybug.exe 0x01  
2:调试启动mybug.exe  
现在程序崩溃的调用栈每次都相同，并且都在相同的线程中，根据调用栈信息很轻松的锁定了bug。

由于上面的例子过于复杂，下面写了一些小程序分析了pageheap的原理

char\* buffer = new char[19];        // 1

buffer[19] = 0;                     // 2

delete [] buffer;                   // 3

这是一个很简单的堆内存越界的例子，在未启动pageheap的情况下，我们来看看buffer的内存情况：  
buffer = 0x00388C80  
第一行执行后,buffer的内存

0x00388C80  cd cd cd cd cd cd cd cd cd cd cd cd cd cd cd cd  ................

0x00388C90  cd cd cd fd fd fd fd ab ab ab ab ab ab ab ab fe  ................

简单说明一下，调试模式下堆上未初始化的内存为cd，并且在内存结束处有4个fd的边界，用于debug模式下crt做内存检查，执行第2行之后，buffer的内存为

0x00388C80  cd cd cd cd cd cd cd cd cd cd cd cd cd cd cd cd  ................  
0x00388C90  cd cd cd 00 fd fd fd ab ab ab ab ab ab ab ab fe  ................

可以看到4个fd的内存边界中第一个fd被破坏了。但这个时候程序并没有崩溃，继续执行第3行，程序崩溃，提示堆错误，可以看到，如果第2行和第3行之间有很长的代码逻辑，那么也只能在第3行执行之后程序才会崩溃。这给调式程序带来了极大的不便。  
如果第2行改为：buffer[24] = 0 程序同样不会崩溃  
如果启用了pageheap，再来看看在debug模式下buffer的内存分配情况：  
第一行分配内存后,buffer的内存情况：

0x01675FE8  cd cd cd cd cd cd cd cd cd cd cd cd cd cd cd cd  ................

0x01675FF8  cd cd cd fd fd fd fd d0 ?? ?? ?? ?? ?? ?? ?? ??  ................

可以看到，和上面一样，在内存结束加上了4个fd的边界，d0是用于填补4字节对齐，注意buffer后面的地址(第一个??)为0x01675FF8+8 = 0x01676000,这是一个4k对齐的PAGE\_NOACCESS页面,这个时候我们执行第2行代码  
buffer[19] = 0; 同样不会崩溃，即使是修改buffer[19-23]的值(4个fd边界和1个对齐d0)，和未启动pageheap一样，程序都只会在执行第3行的时候崩溃。如果修改buffer[24]则程序会崩溃。

通过这个例子，可以得出一个结论：启用pageheap后，堆内存分配在页面的末尾，后面紧跟了一个4k的PAGE\_NOACCESS属性的页面,这种情况下，启用pageheap的好处是能在一定程度上检查内存越界。

再来看一个例子

 char\* buffer = new char[20];  // 1

 delete [] buffer;             // 2

 buffer[1] = 1;                // 3

这个例子演示了操作delete释放后的内存，在未启动pageheap的情况下，程序不会崩溃，原因同上一个例子，启用pageheap后，buffer内存为：  
第一行执行后:

0x01675FE8  cd cd cd cd cd cd cd cd cd cd cd cd cd cd cd cd  ................  
0x01675FF8  cd cd cd cd fd fd fd fd ?? ?? ?? ?? ?? ?? ?? ??  ................

第2行执行后：

0x01675FE8  ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ??  ................  
0x01675FF8  ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ??  ................

可以看到，启用pageheap后delete内存，分配该内存的整个页面都被设置为PAGE\_NOACCESS属性，这样操作delete后的任何内存程序马上就会崩溃。

结论2：启用pageheap很容易检查操作delete后的内存的错误（包括2次delete）

总结：  
1：启用pageheap后，系统的堆管理器会把内存分配到4k页面的末尾（注意需要4字节对齐，debug模式下还存在边界检查的4字节fd）  
2：紧随着的下一个页面被设置为PAGE\_NOACCESS属性  
3：启用pageheap后，释放内存把整个页面设置为PAGE\_NOACCESS属性  
4：内存越界和非法操作依靠非法访问PAGE\_NOACCESS属性的页面暴露问题  
5：由于每块内存都至少需要2个页面（1个页面分配，1个页面PAGE\_NOACCESS），在内存消耗较大的环境下会占用极大的内存资源。  
6：把pageheap和crt的堆检查函数结合起来，能够更好的暴露堆相关bug

ps.pageheap的作用是在注册表位置HKLM/SOFTWARE/Microsoft/Windows NT/CurrentVersion/Image File Execution Options下生成一个项

【转载2】MSDN？

[Full-Page Heap & Normal Page Heap](http://www.cnblogs.com/xiaxi/archive/2011/01/26/1945382.html)

**Full-Page Heap**

Full-page heap should be enabled for individual processes, or under limited parameters for large processes, because of its high memory requirements. It cannot be enabled system-wide, because it is difficult to evaluate the required page file size. Using a page file that is too small with system-wide full-page heap renders the system unbootable.  
  
The advantage of full-page heap is that it causes a process to access violate (AV) exactly at the point of failure. This makes the failure easy to debug. In order to be able to pinpoint failures, first use normal page heap to determine the range where a process is failing, and then use full-page heap on individual large-scale processes for that restricted class of allocations (that is, a specific size range or a specific library).

**Normal Page Heap**

Normal page heap can be used for the testing of large-scale processes without the high memory consumption that full-page heap requires. However, normal page heap delays detection until blocks are freed, thus making failures more difficult to debug.  
  
In general, use normal page heap for initial large-scale processes testing. Then, if problems are detected, enable full-page heap for a restricted class of allocations in those processes.  
  
Normal page heap can be safely enabled system-wide for all processes. This is very useful on test benches that perform general system validation, rather than component focused testing. Normal page heap can also be enabled for a single process.

Interested Link:

<http://support.microsoft.com/kb/264471>

<http://support.microsoft.com/kb/286470>

**【转载3】**[**Linux下PageHeap**](http://www.cppblog.com/feixuwu/archive/2011/05/14/146395.html)

**问题**

   最近游戏开始技术封测了，不过刚刚上线3个小时，Server就挂了，挂在框架代码里，一个不可能挂的地方。  
从CallStack看，是在获取数据时发送请求包的时候挂的，由于框架部分是其他部门的同事开发的，所以查问题的时候就拉上他们了，  
大家折腾了2天，没有实质性的进展，服务器还是基本上每3个小时宕机一次。由于上层逻辑大部分都在我那，所以压力比较大，宕机的直接原因是hashtable的一个桶的指针异常，  
这个hashtable是框架代码的一个内部成员，按道理我们是无从破坏的，只有可能是多线程环境下迭代器损坏导致的。  
但是框架代码在这个地方确实无懈可击，所以真正的原因应该还是上层代码破坏了堆内存，很可能是一个memcpy越界导致的。这毕竟是个猜想，如何找到证据呢，这是个问题。  
把所有代码里的memcpy浏览了一遍，没有发现明显问题。

**猜测**

  一般游戏中比较容易出现但是不好查的问题很多时候都是脚本（lua）导致的，我们的脚本部分是一个同事几年前写的，在几个产品中都使用过，按道理没这么脆弱，不过老大还是和最初开发这个模块的部门沟通了下，  
还真发现问题了，赶紧拿了新的版本更新上去。经过一天的观察，服务器没有宕机了，OK，问题碰巧解决了,背了这么久的黑锅，终于放下来了。

**PageHeap**

   假如没有碰巧解决了这个问题，正常的思路该如何解决这个问题呢，这个时候我怀念windows了，在windows下有PageHeap来解决这类写越界的问题。基本思路就是每次分配内存的时候，都将内存的结尾放在页的边缘，紧接着这块内存分配一块不能写的内存，这样，一旦写越界，就会写异常，导致宕机。linux下没有现成的工具，但是linux提供了mmap功能，我们可以自己实现这样一个功能，当然，这一切都不用自己动手了，tcmalloc已经包含了  
这个功能了，不过在文档里基本没有介绍，我也是在阅读tcmalloc代码时看到的，这个功能默认是关闭的，打开这个开关需要改写代码：  
  
这个代码在debugallocation.cc里：  
  
DEFINE\_bool(malloc\_page\_fence,  
            EnvToBool("TCMALLOC\_PAGE\_FENCE", false),  
            "Enables putting of memory allocations at page boundaries "  
            "with a guard page following the allocation (to catch buffer "  
            "overruns right when they happen).");  
把false改成true就可以了。  
想要在项目里加入PageHeap功能，只需要链接的时候加上 -ltcmalloc\_debug即可。把它加入项目中，试着运行下，直接挂了，  
仔细一看，原来是项目中很多成员变量没有初始化导致的，tcmalloc\_debug会自动将new 和malloc出来的内存初始化为指定值，这样，一旦变量没有初始化，很容易就暴露了。  
修改完这个问题后，编译，再运行，还是挂，这个是mprotect的时候挂的，错误是内存不够，这怎么可能呢，其实是达到了资源限制了。  
echo 128000 > /proc/sys/vm/max\_map\_count  
把map数量限制加大,再运行，OK了！  
   
  但是游戏Server启动后，发现一个问题，CPU长期处于100%，导致登陆一个玩家都很困难，gdb中断后，info thread，发现大部分的操作都在mmap和mprotect,最开始  
怀疑我的linux版本有问题，导致这2个AP慢，写了测试程序试了下，发现其实API不慢，估计是频繁调用导致的。  
所以得换种思路优化下才可以，其实大部分情况下，我们free的时候，无需将页面munmap掉，可以先cache进来，下次分配的时候，如果有，直接拿来用就可以了。  
最简单的cache算法就是定义一个void\* s\_pageCache[50000]数组，页面数相同的内存组成一个链表，挂在一个数组项下，这个很像STL的小内存处理，我们可以将mmap出来的内存的  
前面几个字节(一个指针大小)用于索引下一个freePage。当然这个过程需要加锁，不能用pthread的锁（因为他们会调用malloc等内存分配函数），必须用spinlock，从linux源码里直接抄一个过来即可。  
static void\*   s\_pagePool[MAX\_PAGE\_ALLOC]={0};  
  
malloc的时候，先从pagePool里面获取:  
// 先从pagePool找  
 void\* pFreePage = NULL;  
 spin\_lock(&s\_pageHeapLock);  
 assert(nPageNum < MAX\_PAGE\_ALLOC);  
 if(s\_pagePool[nPageNum])  
 {  
   pFreePage = s\_pagePool[nPageNum];  
   void\* pNextFreePage = \*((void\*\*)pFreePage);  
   s\_pagePool[nPageNum] = pNextFreePage;  
 }  
 spin\_unlock(&s\_pageHeapLock);  
  
free内存的时候，直接放到pagePoll里:  
spin\_lock(&s\_pageHeapLock);  
 assert(nPageNum < MAX\_PAGE\_ALLOC);  
 void\* pNextFree = s\_pagePool[nPageNum];  
 \*(void\*\*)pAddress = pNextFree;  
 s\_pagePool[nPageNum] = pAddress;  
   
 spin\_unlock(&s\_pageHeapLock);  
  
编译、运行,OK了，CPU迅速降下来了，空载的时候不到1%,而且也能达到检测写溢出的问题。