The OOM killer may be called even when there is still plenty of memory available

Adrien Kunysz, Tue, 01 Mar 2011 03:48:02

Many people are confused by memory management on Linux. This is not very surprising considering it is pretty counter intuitive in many ways. A specific element that seems to cause much head scratching is the Out Of Memory killer. The general idea is that Linux overcommits memory in such a way that applications may end up attempting to use more memory than is actually available. If this becomes too much of a problem, the OOM killer will kick in and try to find a memory-hogging process and kill it in an attempt to get the system back to a mangeable state. However there are other cases when the OOM killer may be called and this article describes such a case.

This all began when a user (let's call him yodaz) complained on some forum that his system running 2.6.12.6-arm1 was subject to OOM messages. Yodaz further noticed there was still plenty of swap available to the system and he was obviously confused as to why the OOM was being triggered in his case. He had the good idea of pasting a lot of log messages, most of them being irrelevant but he still managed to include the actual OOM message:

```
oom-killer: gfp_mask=0x2d0
DMA per-cpu:
cpu 0 hot: low 2, high 6, batch 1
cpu 0 cold: low 0, high 2, batch 1
Normal per-cpu: empty
HighMem per-cpu: empty

Free pages:        6524kB (0kB HighMem)
Active:20 inactive:23 dirty:0 writeback:0 unstable:0 free:1631 slab:874 mapped:20 pagetables:90
DMA free:6524kB min:1200kB low:1500kB high:1800kB active:80kB inactive:92kB present:16384kB pages_scanned:41 all_unreclaimable? no
lowmem_reserve[]: 0 0 0
Normal free:0kB min:0kB low:0kB high:0kB active:0kB inactive:0kB present:0kB pages_scanned:0 all_unreclaimable? no
lowmem_reserve[]: 0 0 0
HighMem free:0kB min:128kB low:160kB high:192kB active:0kB inactive:0kB present:0kB pages_scanned:0 all_unreclaimable? no
lowmem_reserve[]: 0 0 0
DMA: 411*4kB 252*8kB 113*16kB 27*32kB 1*64kB 1*128kB 0*256kB 0*512kB 0*1024kB 0*2048kB 0*4096kB = 6524kB
Normal: empty
HighMem: empty
Swap cache: add 24282, delete 24269, find 7705/11790, race 0+0
Free swap  = 124768kB
Total swap = 128448kB
Out of Memory: Killed process 453 (smbd).
```

As we can see, the memory is split into three zones: DMA, Normal and HighMem. Close observation shows the Normal and HighMem zones are actually completely empty. The system is only using the DMA zone. It's the first time I see this but then it's the first time I see an OOM message from an ARM. Anyway, this bit:

```
DMA free:6524kB min:1200kB low:1500kB
```

Clearly shows there is still over 6 MB of memory which is way above both the low mark value of 1500 kB and the minimum (1200 kB) the system will accept before taking drastic action (calling the OOM killer). A while ago, Dominic Duval wrote a short webpage with a nice graphic explaining what these limits are for. That page is better than any explanation I could give about this.

So, there is still memory available. How come the OOM killer is triggered? Another common cause is memory fragmentation. In some case you may have plenty of memory available but it is split in many small blocks. Hence, if something attempts to allocate a relatively large chunk of contiguous memory, it may still fail and when memory allocation fails, you may end up calling the OOM killer. Now, we actually have information about how fragmented the memory is in the dump that was provided:

```
DMA: 411*4kB 252*8kB 113*16kB 27*32kB 1*64kB 1*128kB 0*256kB 0*512kB 0*1024kB 0*2048kB 0*4096kB = 6524kB
```

See? There is 6525 kB of free memory split in 411 chunks of 4 kB, 252 of 8 kB and so on. Notice how the largest available chunk is 128 kB. The memory fragmentation hypothesis seems to match but we still have no idea what is actually attempting to allocate memory and triggering the OOM. At this stage there are various ways to instrument the kernel to obtain a backtrace that would give us some indications as to where that failed allocation comes from but maybe there are other messages before the OOM that could give us a clue?

After sharing my observation with yodaz, he did notice some interesting messages before the OOM:

```
XFS: possible memory allocation deadlock in kmem_alloc (mode:0x2d0)
```

Notice how the `mode` value is the same than the `gfp_mask` in the OOM message. Some digging inside the kernel source tells us where this message comes from:

```
47void *
48kmem_alloc(size_t size, int flags)
49{
50        int     retries = 0;
51        int     lflags = kmem_flags_convert(flags);
52        void    *ptr;
53
54        do {
55                if (size < MAX_SLAB_SIZE || retries > MAX_VMALLOCS)
56                        ptr = kmalloc(size, lflags);
57                else
58                        ptr = __vmalloc(size, lflags, PAGE_KERNEL);
59                if (ptr || (flags & (KM_MAYFAIL|KM_NOSLEEP)))
60                        return ptr;
61                if (!(++retries % 100))
62                        printk(KERN_ERR "XFS: possible memory allocation "
63                                        "deadlock in %s (mode:0x%x)\n",
64                                        __FUNCTION__, lflags);
65                blk_congestion_wait(WRITE, HZ/50);
66        } while (1);
67}
```

This doesn't look very nice. What this means is that when XFS tries to allocate chunks of more than `MAX_SLAB_SIZE` bytes, it first tries to get contiguous memory then if it failed `MAX_VMALLOCS` times it will fall back to non contiguous memory. So, this means nothing is actually failing: XFS will still gets the memory it needs but meanwhile, it will have caused the OOM killer to be triggered and an (arguably) innocent Samba daemon will have been forcibly terminated.

Now, how do we fix that? Well, 2.6.12.6 is pretty old so upgrading is probably a good idea as recent kernels have many improvements in memory management (thus reducing risks of high fragmentation) and in the XFS driver (potentially reducing the need for large chunks needing to be allocated at inopportune time). It is also possible that some careful tuning of the XFS driver may avoid the problematic situation as is some serious memory profiling of the system.

Another more obvious fix is to change `kmem_alloc()` and its callers as to attempt to allocate contiguous memory only when strictly necessary. This is what commit bdfb04301fa5 does. After that patch, the function looks like this:

```
47void *
48kmem_alloc(size_t size, unsigned int __nocast flags)
49{
50        int     retries = 0;
51        gfp_t   lflags = kmem_flags_convert(flags);
52        void    *ptr;
53
54        do {
55                ptr = kmalloc(size, lflags);
56                if (ptr || (flags & (KM_MAYFAIL|KM_NOSLEEP)))
57                        return ptr;
58                if (!(++retries % 100))
59                        printk(KERN_ERR "XFS: possible memory allocation "
60                                        "deadlock in %s (mode:0x%x)\n",
61                                        __func__, lflags);
62                congestion_wait(BLK_RW_ASYNC, HZ/50);
63        } while (1);
64}
```

We are still looping forever to get our chunk of memory but we are not trying `__vmalloc()` first then changing our mind if it fails. This means we are less likely to trigger an OOM when the memory is highly fragmented and our smbd daemon will not die (well, not for the same reason at least).

As far as I know, after a `fsck^Wxfs_repair` that obviously didn't help at all, yodaz is now trying to rebuild a newer version of his XFS module.

Back to all articles.