

Re: [PATCH 00/28] Swap over NFS -v16

Source: <http://linux.derkeiler.com/Mailing-Lists/Kernel/2008-03/msg02951.html>

- *From:* Neil Brown <neilb@xxxxxxx>
 - *Date:* Fri, 7 Mar 2008 14:33:11 +1100
-

On Tuesday March 4, a.p.zijlstra@xxxxxxxxx wrote:

On Tue, 2008-03-04 at 10:41 +1100, Neil Brown wrote:

Those skbs we allocated – they are either sitting in the fragment cache, or have been attached to a SK_MEMALLOC socket, or have been freed – correct? If so, then there is already a limit to how much memory they can consume.

Not really, there is no natural limit to the amount of packets that can be in transit between RX and socket demux. So we need the extra (skb) accounting to impose that.

Isn't there? A brief look at the code suggests that (except for fragment handling) there is a fairly straight path from network-receive to socket demux. No queues along the way. That suggests the number of in-transit skbs should be limited by the number of CPUs. Did I miss something? Or is the number of CPUs potentially too large to be a suitable limit (seems unlikely).

While looking at the code it also occurred to me that:
1/ tcpdump could be sent incoming packets. Is there a limit to the number of packets that can be in-kernel waiting for tcpdump to collect them? Should this limit be added to the base reserve?

2/ If the host is routing network packets, then incoming packets might go on an outbound queue. Is this space limited? and included in the reserve?

Not major points, but I thought I would mention them.

I also don't see the value of tracking pages to see if they are 'reserve' pages or not. The decision to drop an skb that is not for an SK_MEMALLOC socket should be based on whether we are currently

Re: [PATCH 00/28] Swap over NFS -v16

short on memory. Not whether we were short on memory when the skb was allocated.

That comes from accounting, once you need to account data you need to know when to start accounting, and keep state so that you can properly un-account.

skbs in the main (only?) thing you do accounting on, so focusing on those:

Suppose that every time you allocate memory for an skb, you check if the allocation had to dip into emergency reserves, and account the memory if so – releasing the memory and dropping the packet if we are over the limit.

And any time you free memory associated with an skb, you check if the accounts currently say '0', and if not subtract the size of the allocation from the accounts.

Then you have quite workable accounting that doesn't need to tag every piece of memory with its 'reserve' status, and only pays the accounting cost (presumably a spinlock) when running out of memory, or just recovering.

This more relaxed approach to accounting reserved vs non-reserved memory has a strong parallel in your slub code (which I now understand). When sl[au]b first gets a ->reserve page, it sets the ->reserve flag on the memcache and leaves it set until it sometime later gets a non-"->reserve" page. Any memory freed in the mean time (whether originally reserved or not) is treated as reserve memory in that it will only be returned for ALLOC_NO_WATERMARKS allocations. I think this is a good way of working with reserved memory. It isn't precise, but it is low-cost and good enough to get you through the difficult patch.

Your netvm-skbuff-reserve.patch has some code to make sure that all the allocations in an skb have the same 'reserve' status. I don't think that is needed and just makes the code messy – plus it requires the 'overcommit' flag to mem_reserve_kmalloc_charge which is a bit of a wart on the interface.

I would suggest getting rid of that. Just flag the whole skb if any part gets a 'reserve' allocation, and use that flag to decide to drop packets arriving at non-SK_MEMALLOC sockets.

So: I think I now really understand what your code is doing, so I will try to explain it in terms that even I understand... This text in

explicitly available under GPLv2 in case you want it.

It actually describes something a bit different to what your code currently does, but I think it is very close to the spirit. Some differences follow from my observations above. Others the way that seemed to make sense while describing the problem and solution differed slightly from what I saw the code doing. Obviously the code and the description should be aligned one way or another before being finalised.

The description is a bit long ... sorry about that. But I wanted to make sure I included motivation and various assumptions. Some of my understanding may well be wrong, but I present it here anyway. It is easier for you to correct if it is clearly visible:-)

Problem:

When Linux needs to allocate memory it may find that there is insufficient free memory so it needs to reclaim space that is in use but not needed at the moment. There are several options:

- 1/ Shrink a kernel cache such as the inode or dentry cache. This is fairly easy but provides limited returns.
- 2/ Discard 'clean' pages from the page cache. This is easy, and works well as long as there are clean pages in the page cache. Similarly clean 'anonymous' pages can be discarded – if there are any.
- 3/ Write out some dirty page-cache pages so that they become clean. The VM limits the number of dirty page-cache pages to e.g. 40% of available memory so that (among other reasons) a "sync" will not take excessively long. So there should never be excessive amounts of dirty pagecache.
Writing out dirty page-cache pages involves work by the filesystem which may need to allocate memory itself. To avoid deadlock, filesystems use GFP_NOFS when allocating memory on the write-out path. When this is used, cleaning dirty page-cache pages is not an option so if the filesystem finds that memory is tight, another option must be found.
- 4/ Write out dirty anonymous pages to the "Swap" partition/file. This is the most interesting for a couple of reasons.
 - a/ Unlike dirty page-cache pages, there is no need to write anon pages out unless we are actually short of memory. Thus they tend to be left to last.
 - b/ Anon pages tend to be updated randomly and unpredictably, and flushing them out of memory can have a very significant performance impact on the process using them. This contrasts with page-cache pages which are often written sequentially and often treated as "write-once, read-many".
So anon pages tend to be left until last to be cleaned, and may be the only cleanable pages while there are still some dirty page-cache pages (which are waiting on a GFP_NOFS allocation).

[I don't find the above wholly satisfying. There seems to be too much

hand-waving. If someone can provide better text explaining why swapout is a special case, that would be great.]

So we need to be able to write to the swap file/partition without needing to allocate any memory ... or only a small well controlled amount.

The VM reserves a small amount of memory that can only be allocated for use as part of the swap-out procedure. It is only available to processes with the PF_MEMALLOC flag set, which is typically just the memory cleaner.

Traditionally swap-out is performed directly to block devices (swap files on block-device filesystems are supported by examining the mapping from file offset to device offset in advance, and then using the device offsets to write directly to the device). Block devices are (required to be) written to pre-allocate any memory that might be needed during write-out, and to block when the pre-allocated memory is exhausted and no other memory is available. They can be sure not to block forever as the pre-allocated memory will be returned as soon as the data it is being used for has been written out. The primary mechanism for pre-allocating memory is called "mempools".

This approach does not work for writing anonymous pages (i.e. swapping) over a network, using e.g NFS or NBD or iSCSI.

The main reason that it does not work is that when data from an anon page is written to the network, we must wait for a reply to confirm the data is safe. Receiving that reply will consume memory and, significantly, we need to allocate memory to an incoming packet before we can tell if it is the reply we are waiting for or not.

The secondary reason is that the network code is not written to use mempools and in most cases does not need to use them. Changing all allocations in the networking layer to use mempools would be quite intrusive, and would waste memory, and probably cause a slow-down in the common case of not swapping over the network.

These problems are addressed by enhancing the system of memory reserves used by PF_MEMALLOC and requiring any in-kernel networking client that is used for swap-out to indicate which sockets are used for swapout so they can be handled specially in low memory situations.

There are several major parts to this enhancement:

1/ PG_emergency, GFP_MEMALLOC

To handle low memory conditions we need to know when those conditions exist. Having a global "low on memory" flag seems easy, but its implementation is problematic. Instead we make it possible

to tell if a recent memory allocation required use of the emergency memory pool.

For pages returned by `alloc_page`, the new page flag `PG_emergency` can be tested. If this is set, then a low memory condition was current when the page was allocated, so the memory should be used carefully.

For memory allocated using slab/slub: If a page that is added to a `kmem_cache` is found to have `PG_emergency` set, then a `->reserve` flag is set for the whole `kmem_cache`. Further allocations will only be returned from that page (or any other page in the cache) if they are emergency allocation (i.e. `PF_MEMALLOC` or `GFP_MEMALLOC` is set). Non-emergency allocations will block in `alloc_page` until a non-reserve page is available. Once a non-reserve page has been added to the cache, the `->reserve` flag on the cache is removed. When memory is returned by slab/slub, `PG_emergency` is set on the page holding the memory to match the `->reserve` flag on that cache.

After memory has been returned by `kmem_cache_alloc` or `kmalloc`, the page's `PG_emergency` flag can be tested. If it is set, then the most recent allocation from that cache required reserve memory, so this allocation should be used with care.

It is not safe to test the cache's `->reserve` flag immediately after an allocation as that flag is in per-cpu data, and the process could have been rescheduled to a different cpu if preemption is enabled. Thus the use of `PG_emergency` to carry this information.

This allows us to

- a/ request use of the emergency pool when allocating memory (`GFP_MEMALLOC`), and
- b/ to find out if the emergency pool was used.

2/ `SK_MEMALLOC`, `sk_buff->emergency`.

When memory from the reserve is used to store incoming network packets, the memory must be freed (and the packet dropped) as soon as we find out that the packet is not for a socket that is used for swap-out.

To achieve this we have an `->emergency` flag for skbs, and an `SK_MEMALLOC` flag for sockets.

When memory is allocated for an skb, it is allocated with `GFP_MEMALLOC` (if we are currently swapping over the network at all). If a subsequent test shows that the emergency pool was used, `->emergency` is set.

When the skb is finally attached to its destination socket, the `SK_MEMALLOC` flag on the socket is tested. If the skb has `->emergency` set, but the socket does not have `SK_MEMALLOC` set, then the skb is immediately freed and the packet is dropped.

This ensures that reserve memory is never queued on a socket that is not used for swapout.

Similarly, if an skb is ever queued for deliver to user-space for example by netfilter, the `->emergency` flag is tested and the skb is released if `->emergency` is set.

This ensures that memory from the emergency reserve can be used to allow swapout to proceed, but will not get caught up in any other network queue.

3/ pages_emergency

The above would be sufficient if the total memory below the lowest memory watermark (i.e the size of the emergency reserve) were known to be enough to hold all transient allocations needed for writeout. I'm a little blurry on how big the current emergency pool is, but it isn't big and certainly hasn't been sized to allow network traffic to consume any.

We could simply make the size of the reserve bigger. However in the common case that we are not swapping over the network, that would be a waste of memory.

So a new "watermark" is defined: `pages_emergency`. This is effectively added to the current low water marks, so that pages from this emergency pool can only be allocated if one of `PF_MEMALLOC` or `GFP_MEMALLOC` are set.

`pages_emergency` can be changed dynamically based on need. When swapout over the network is required, `pages_emergency` is increased to cover the maximum expected load. When network swapout is disabled, `pages_emergency` is decreased.

To determine how much to increase it by, we introduce reservation groups....

3a/ reservation groups

The memory used transiently for swapout can be in a number of different places. e.g. the network route cache, the network fragment cache, in transit between network card and socket, or (in the case of NFS) in sunrpc data structures awaiting a reply. We need to ensure each of these is limited in the amount of memory they use, and that the maximum is included in the reserve.

The memory required by the network layer only needs to be reserved once, even if there are multiple swapout paths using the network (e.g. NFS and NDB and iSCSI, though using all three for swapout at the same time would be unusual).

So we create a tree of reservation groups. The network might

register a collection of reservations, but not mark them as being in use. NFS and sunrpc might similarly register a collection of reservations, and attach it to the network reservations as it depends on them.

When swapout over NFS is requested, the NFS/sunrpc reservations are activated which implicitly activates the network reservations.

The total new reservation is added to pages_emergency.

Provided each memory usage stays beneath the registered limit (at least when allocating memory from reserves), the system will never run out of emergency memory, and swapout will not deadlock.

It is worth noting here that it is not critical that each usage stays beneath the limit 100% of the time. Occasional excess is acceptable provided that the memory will be freed again within a short amount of time that does **not** require waiting for any event that itself might require memory.

This is because, at all stages of transmit and receive, it is acceptable to discard all transient memory associated with a particular writeout and try again later. On transmit, the page can be re-queued for later transmission. On receive, the packet can be dropped assuming that the peer will resend after a timeout.

Thus allocations that are truly transient and will be freed without blocking do not strictly need to be reserved for. Doing so might still be a good idea to ensure forward progress doesn't take too long.

4/ lo-mem accounting

Most places that might hold on to emergency memory (e.g. route cache, fragment cache etc) already place a limit on the amount of memory that they can use. This limit can simply be reserved using the above mechanism and no more needs to be done.

However some memory usage might not be accounted with sufficient firmness to allow an appropriate emergency reservation. The in-flight skbs for incoming packets is (claimed to be) on such example.

To support this, a low-overhead mechanism for accounting memory usage against the reserves is provided. This mechanism uses the same data structure that is used to store the emergency memory reservations through the addition of a 'usage' field.

When memory allocation for a particular purpose succeeds, the memory is checked to see if it is 'reserve' memory. If it is, the size of the allocation is added to the 'usage'. If this exceeds the reservation, the usage is reduced again and the memory that was allocated is free.

When memory that was allocated for that purpose is freed, the 'usage' field is checked again. If it is non-zero, then the size of the freed memory is subtracted from the usage, making sure the usage never becomes less than zero.

This provides adequate accounting with minimal overheads when not in a low memory condition. When a low memory condition is encountered it does add the cost of a spin lock necessary to serialise updates to 'usage'.

5/ swapfile/swap_out/swap_in

So that a filesystem (e.g. NFS) can know when to set SK_MEMALLOC on any network socket that it uses, and can know when to account reserve memory carefully, new address_space_operations are available.

"swapfile" requests that an address space (i.e a file) be make ready for swapout. swap_out and swap_in request the actual IO. They together must ensure that each swap_out request can succeed without allocating more emergency memory that was reserved by swapfile.

Thanks for reading this far. I hope it made sense :-)

NeilBrown

--

To unsubscribe from this list: send the line "unsubscribe linux-kernel" in the body of a message to majordomo@xxxxxxxxxxxxxxxxx

More majordomo info at <http://vger.kernel.org/majordomo-info.html>

Please read the FAQ at <http://www.tux.org/lkml/>