Subject: Re: [PATCH 0/7] memcg kernel memory tracking
Posted by Suleiman Souhlal on Wed, 22 Feb 2012 20:32:03 GMT
View Forum Message <> Reply to Message

On Wed, Feb 22, 2012 at 5:58 AM, Glauber Costa <glommer@parallels.com> wrote:
>>> As previously proposed, one has the option of keeping kernel memory
>>> accounted separatedly, or together with the normal userspace memory.
>>> However, this time I made the option to, in this later case, bill
>>> the memory directly to memcg->res. It has the disadvantage that it
>>> becomes
>>> complicated to know which memory came from user or kernel, but OTOH,
>>> it does not create any overhead of drawing from multiple res_counters
>>> at read time. (and if you want them to be joined, you probably don't
>>> care)
>>
>>
>> It would be nice to still keep a kernel memory counter (that gets
>> updated at the same time as memcg->res) even when the limits are not
>> independent, because sometimes it's important to know how much kernel
>> memory is being used by a cgroup.
>
>
> Can you clarify in this "sometimes" ? The way I see it, we either always use
> two counters - as did in my original proposal - or use a single counter for
> this case. Keeping a separated counter and still billing to the user memory
> is the worst of both worlds to me, since you get the performance hit of
> updating two resource counters.

By "sometimes", I mean pretty much any time we have to debug why a
cgroup is out of memory.
If there is no counter for how much kernel memory is used, it's pretty
much impossible to determine why the cgroup is full.

As for the performance, I do not think it is bad, as the accounting is
done in the slow path of slab allocation, when we allocate/free pages.

>
>
>>> Kernel memory is never tracked for the root memory cgroup. This means
>>> that a system where no memory cgroups exists other than the root, the
>>> time cost of this implementation is a couple of branches in the slub
>>> code - none of them in fast paths. At the moment, this works only
>>> with the slub.
>>>
>>> At cgroup destruction, memory is billed to the parent. With no hierarchy,
>>> this would mean the root memcg. But since we are not billing to that,
>>> it simply ceases to be tracked.
>>>

>>> The caches that we want to be tracked need to explicit register into
>>> the infrastructure.
>>
>>
>> Why not track every cache unless otherwise specified? If you don't,
>> you might end up polluting code all around the kernel to create
>> per-cgroup caches.
>> From what I've seen, there are a fair amount of different caches that
>> can end up using a significant amount of memory, and having to go
>> around and explicitly mark each one doesn't seem like the right thing
>> to do.
>>
> The registration code is quite simple, so I don't agree this is polluting
> code all around the kernel. It is just a couple of lines.
>
> Of course, in an opt-out system, this count would be zero. So is it better?
>
> Let's divide the caches in two groups: Ones that use shrinkers, and simple
> ones that won't do. I am assuming most of the ones we need to track use
> shrinkers somehow.
>
> So if they do use a shrinker, it is very unlikely that the normal shrinkers
> will work without being memcg-aware. We then end up in a scenario in which
> we track memory, we create a bunch of new caches, but we can't really force
> reclaim on that cache. We then depend on luck to have the objects reclaimed
> from the root shrinker. Note that this is a problem that did not exist
> before: a dcache shrinker would shrink dcache objects and that's it, but we
> didn't have more than one cache with those objects.
>
> So in this context, registering a cache explicitly is better IMHO, because
> what you are doing is telling: "I examined this cache, and I believe it will
> work okay with the memcg. It either does not need changes to the shrinker,
> or I made them already"
>
> Also, everytime we create a new cache, we're wasting some memory, as we
> duplicate state. That is fine, since we're doing this to prevent the usage
> to explode.
>
> But I am not sure it pays of in a lot of caches, even if they use a lot of
> pages: Like, quickly scanning slabinfo:
>
> task_struct   512   570   5920   5   8 : tunables   0   0 0 :
> slabdata   114   114   0
>
> Can only grow if # of processes grow. Likely to hit a limit on that first.
>
> Acpi-Namespace   4348   5304   40 102   1 : tunables   0   0 0 :
> slabdata   52   52   0

>
> I doubt we can take down a sane system by using this cache...
>
> and so on and so forth.
>
> What do you think?

Well, we've seen several slabs that don't have shrinkers use significant amounts of memory. For example, size-64, size-32, vm_area_struct, buffer_head, radix_tree_node, TCP, filp..

For example, consider this perl program (with high enough file descriptor limits):
use POSIX; use Socket; my $i; for ($i = 0; $i < 100000; $i++) {
socket($i, PF_INET, SOCK_STREAM, 0) || die "socket: $!"; }

One can make other simple programs like this that use significant amounts of slab memory.

Having to look at a running kernel, having to find out which caches are significant, and then going back and marking them for accounting, really doesn't seem the right approach to me.

-- Suleiman