Subject: Re: [PATCH v3 2/2] decrement static keys on real destroy time
Posted by Glauber Costa on Thu, 26 Apr 2012 21:58:37 GMT
View Forum Message <> Reply to Message

```
On 04/26/2012 06:39 PM, Tejun Heo wrote:
> Hello, Glauber.
>
> Overall, I like this approach much better.  Just some nits below.
>
> On Thu, Apr 26, 2012 at 06:24:23PM -0300, Glauber Costa wrote:
>> @@ -4836,6 +4851,18 @@ static void free_work(struct work_struct *work)
>>    int size = sizeof(struct mem_cgroup);
>>
>>    memcg = container_of(work, struct mem_cgroup, work_freeing);
>> + /*
>> +  * We need to make sure that (at least for now), the jump label
>> +  * destruction code runs outside of the cgroup lock. It is in theory
>> +  * possible to call the cgroup destruction function outside of that
>> +  * lock, but it is not yet done. rate limiting plus the deferred
>> +  * interface for static_branch destruction guarantees that it will
>> +  * run through schedule_work(), therefore, not holding any cgroup
>> +  * related lock (this is, of course, until someone decides to write
>> +  * a schedule_work cgroup :p )
>> +  */
>
> Isn't the above a bit too verbose?  Wouldn't just stating the locking
> dependency be enough?

I used a lot of verbosity here because it is a tricky and racy issue.
I am fine with trimming the comments if this is considered too much.

>>    cg_proto->sysctl_mem = tcp->tcp_prot_mem;
>>    cg_proto->memory_allocated =&tcp->tcp_memory_allocated;
>>    cg_proto->sockets_allocated =&tcp->tcp_sockets_allocated;
>> + cg_proto->active = false;
>> + cg_proto->activated = false;
>
> Isn't the memory zallocd?  I find 0 / NULL / false inits unnecessary
> and even misleading (can the memory be non-zero here?).  Another side
> effect is that it tends to get out of sync as more fields are added.

I can take them off.

>
>> +/*
>> + * This is to prevent two writes arriving at the same time
>> + * at kmem.tcp.limit_in_bytes.
>> + *
>> + * There is a race at the first time we write to this file:
>> + *
>> + * - cg_proto->activated == false for all writers.
>> + * - They all do a static_key_slow_inc().
>> + * - When we are finally read to decrement the static_keys,
>                            ^
>                            ready
```

Thanks.

```
>> + *   we'll do it only once per activated cgroup. So we won't
>> + *   be able to disable it.
>> + *
>> + *   Also, after the first caller increments the static_branch
>> + *   counter, all others will return right away. That does not mean,
>> + *   however, that the update is finished.
>> + *
>> + *   Without this mutex, it would then be possible for a second writer
>> + *   to get to the update site, return
>
> I kinda don't follow the above sentence.
```

I will try to rephrase it for more clarity. But this is the thing behind
this patchset coming and going with so many attempts:

jump label updates are atomic given a single patch site. But they are
not atomic given multiple patch sites.

In our case, they are pretty spread around. Which means that while some
of them are already patched, some are not. If the socket marking in
sock_update_memcg is done last, we're fine, because all the accounters
test for that. Otherwise, we can misaccount.

To protect against that, we use the "activated" field. But it need to be
lock-protected, otherwise a second writer can arrive here before the
update is finished, update the accounted field, and we're down to the
same problem as before.

```
>> + *   When a user updates limit of 2 cgroups at once, following happens.
>> + *
>> + *    CPU A    CPU B
>> + *
>> + * if (cg_proto->activated) if (cg->proto_activated)
>> + *  static_key_inc()  static_key_inc()
>> + *   =>  set counter 0->1  =>  set counter 1->2,
>> + *       return immediately.
>> + *   =>  hold mutex   =>  cg_proto->activated = true.
>> + *   =>  overwrite jmps.
```

>
> Isn't this something which should be solved from static_keys API? Why
> is this being worked around from memcg? Also, I again hope that the
> explanation is slightly more concise.
>

At first I though that we could get rid of all this complication by
calling stop machine from the static_branch API. This would all
magically go away. I actually even tried it.

However, reading the code for other architectures (other than x86), I
found that they usually rely on the fixed instruction size to just patch
an instruction atomically and go home happy.

Using stop machine and the like would slow them down considerably. Not
only slow down the static branch update (which is acceptable), but
everybody else (which is horrible). It seemed to defeat the purpose of
static branches a bit.

The other users of static branches seems to be fine coping with the fact
that in cases with multiple-sites, they will spread in time.