Subject: Re: [PATCH 3/3] decrement static keys on real destroy time
Posted by KAMEZAWA Hiroyuki on Fri, 20 Apr 2012 07:38:49 GMT
View Forum Message <> Reply to Message

(2012/04/20 7:49), Glauber Costa wrote:

> We call the destroy function when a cgroup starts to be removed,
> such as by a rmdir event.
>
> However, because of our reference counters, some objects are still
> inflight. Right now, we are decrementing the static_keys at destroy()
> time, meaning that if we get rid of the last static_key reference,
> some objects will still have charges, but the code to properly
> uncharge them won't be run.
>
> This becomes a problem specially if it is ever enabled again, because
> now new charges will be added to the staled charges making keeping
> it pretty much impossible.
>
> We just need to be careful with the static branch activation:
> since there is no particular preferred order of their activation,
> we need to make sure that we only start using it after all
> call sites are active. This is achieved by having a per-memcg
> flag that is only updated after static_key_slow_inc() returns.
> At this time, we are sure all sites are active.
>
> This is made per-memcg, not global, for a reason:
> it also has the effect of making socket accounting more
> consistent. The first memcg to be limited will trigger static_key()
> activation, therefore, accounting. But all the others will then be
> accounted no matter what. After this patch, only limited memcgs
> will have its sockets accounted.
>
> [v2: changed a tcp limited flag for a generic proto limited flag]
>
> Signed-off-by: Glauber Costa <glommer@parallels.com>

> ---
> include/net/sock.h | 9 +++++++
> mm/memcontrol.c | 20 ++++++++++++++++-
> net/ipv4/tcp_memcontrol.c | 52 ++-
> 3 files changed, 72 insertions(+), 9 deletions(-)
>
> diff --git a/include/net/sock.h b/include/net/sock.h
> index b3ebe6b..c5a2010 100644
> --- a/include/net/sock.h
> +++ b/include/net/sock.h
> @@ -914,6 +914,15 @@ struct cg_proto {

> int *memory_pressure;
> long *sysctl_mem;
> /*
> + * active means it is currently active, and new sockets should
> + * be assigned to cgroups.
> + *
> + * activated means it was ever activated, and we need to
> + * disarm the static keys on destruction
> + */
> + bool activated;
> + bool active;
> + /*
>   * memcg field is used to find which memcg we belong directly
>   * Each memcg struct can hold more than one cg_proto, so container_of
>   * won't really cut.
> diff --git a/mm/memcontrol.c b/mm/memcontrol.c
> index 7832b4d..01d25a0 100644
> --- a/mm/memcontrol.c
> +++ b/mm/memcontrol.c
> @@ -404,6 +404,7 @@ void sock_update_memcg(struct sock *sk)
> {
> if (mem_cgroup_sockets_enabled) {
>   struct mem_cgroup *memcg;
> + struct cg_proto *cg_proto;
>
>   BUG_ON(!sk->sk_prot->proto_cgroup);
>
> @@ -423,9 +424,10 @@ void sock_update_memcg(struct sock *sk)
>
>   rcu_read_lock();
>   memcg = mem_cgroup_from_task(current);
> - if (!mem_cgroup_is_root(memcg)) {
> + cg_proto = sk->sk_prot->proto_cgroup(memcg);
> + if (!mem_cgroup_is_root(memcg) && cg_proto->active) {

>

>   mem_cgroup_get(memcg);
> -   sk->sk_cgrp = sk->sk_prot->proto_cgroup(memcg);
> +   sk->sk_cgrp = cg_proto;
>   }

Is this correct ? cg_proto->active can be true before all jump_labels are patched, then we can loose accounting. That will cause underflow of res_countner.

cg_proto->active should be set after jump_label modification.
Then, things will work, I guess.

Thanks,
-Kame

> rcu_read_unlock();
> }
> @@ -442,6 +444,14 @@ void sock_release_memcg(struct sock *sk)
> }
> }
>
> +static void disarm_static_keys(struct mem_cgroup *memcg)
> +{
> +#ifdef CONFIG_INET
> + if (memcg->tcp_mem.cg_proto.activated)
> + static_key_slow_dec(&memcg_socket_limit_enabled);
> +#endif
> +}
> +
> #ifdef CONFIG_INET
> struct cg_proto *tcp_proto_cgroup(struct mem_cgroup *memcg)
> {
> @@ -452,6 +462,11 @@ struct cg_proto *tcp_proto_cgroup(struct mem_cgroup *memcg)
> }
> EXPORT_SYMBOL(tcp_proto_cgroup);
> #endif /* CONFIG_INET */
> +#else
> +static inline void disarm_static_keys(struct mem_cgroup *memcg)
> +{
> +}
> +
> #endif /* CONFIG_CGROUP_MEM_RES_CTLR_KMEM */
>
> static void drain_all_stock_async(struct mem_cgroup *memcg);
> @@ -4883,6 +4898,7 @@ static void __mem_cgroup_put(struct mem_cgroup *memcg, int count)
> {
> if (atomic_sub_and_test(count, &memcg->refcnt)) {
> struct mem_cgroup *parent = parent_mem_cgroup(memcg);
> + disarm_static_keys(memcg);
> __mem_cgroup_free(memcg);
> if (parent)
> mem_cgroup_put(parent);
> diff --git a/net/ipv4/tcp_memcontrol.c b/net/ipv4/tcp_memcontrol.c

> index 1517037..d02573a 100644
> --- a/net/ipv4/tcp_memcontrol.c
> +++ b/net/ipv4/tcp_memcontrol.c
> @@ -54,6 +54,8 @@ int tcp_init_cgroup(struct mem_cgroup *memcg, struct cgroup_subsys
*ss)
> 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;
> cg_proto->memcg = memcg;
>
> return 0;
> @@ -74,12 +76,23 @@ void tcp_destroy_cgroup(struct mem_cgroup *memcg)
> percpu_counter_destroy(&tcp->tcp_sockets_allocated);
>
> val = res_counter_read_u64(&tcp->tcp_memory_allocated, RES_LIMIT);
> -
> - if (val != RESOURCE_MAX)
> - static_key_slow_dec(&memcg_socket_limit_enabled);
> }
> EXPORT_SYMBOL(tcp_destroy_cgroup);
>
> +/*
> + * 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,
> + * we'll do it only once per activated cgroup. So we won't
> + * be able to disable it.
> + */
> +static DEFINE_MUTEX(tcp_set_limit_mutex);
> +
> static int tcp_update_limit(struct mem_cgroup *memcg, u64 val)
> {
> struct net *net = current->nsproxy->net_ns;
> @@ -107,10 +120,35 @@ static int tcp_update_limit(struct mem_cgroup *memcg, u64 val)
> tcp->tcp_prot_mem[i] = min_t(long, val >> PAGE_SHIFT,
> net->ipv4.sysctl_tcp_mem[i]);
>
> - if (val == RESOURCE_MAX && old_lim != RESOURCE_MAX)
> - static_key_slow_dec(&memcg_socket_limit_enabled);
> - else if (old_lim == RESOURCE_MAX && val != RESOURCE_MAX)
> - static_key_slow_inc(&memcg_socket_limit_enabled);

```
> + if (val == RESOURCE_MAX)
> +  cg_proto->active = false;
> + else if (val != RESOURCE_MAX) {
> +  cg_proto->active = true;
> +
> +
> +  /*
> +   * ->activated needs to be written after the static_key update.
> +   *  This is what guarantees that the socket activation function
> +   *  is the last one to run. See sock_update_memcg() for details,
> +   *  and note that we don't mark any socket as belonging to this
> +   *  memcg until that flag is up.
> +   *
> +   *  We need to do this, because static_keys will span multiple
> +   *  sites, but we can't control their order. If we mark a socket
> +   *  as accounted, but the accounting functions are not patched in
> +   *  yet, we'll lose accounting.
> +   *
> +   *  We never race with the readers in sock_update_memcg(), because
> +   *  when this value change, the code to process it is not patched in
> +   *  yet.
> +   */
> +  mutex_lock(&tcp_set_limit_mutex);
> +  if (!cg_proto->activated) {
> +   static_key_slow_inc(&memcg_socket_limit_enabled);
> +   cg_proto->activated = true;
> +  }
> +  mutex_unlock(&tcp_set_limit_mutex);
> + }
>
>   return 0;
>  }
```