Subject: [PATCH 0/3] Fix problem with static_key decrement
Posted by Glauber Costa on Thu, 19 Apr 2012 22:49:15 GMT
View Forum Message <> Reply to Message

Hi,

This is my proposed fix for the sock memcg static_key
problem raised by Kamezawa. It works for me, but I would
Kame, please confirm.

For that to work, I am dependent on two cgroup patches
that goes attached. The rationale behind it, is that we
can't do static_key updates with the cgroup_mutex held,
or we risk deadlocking.

Looking closely, there seem to be no particular reason
to hold the cgroup_mutex during destruction. Subsystems
that really need it, can hold it themselves.

Tejun, let me know if this is acceptable from your PoV.

Glauber Costa (3):
  don't attach a task to a dead cgroup
  don't take cgroup_mutex in destroy()
  decrement static keys on real destroy time

```
 block/blk-cgroup.c         |    2 +
 include/net/sock.h         |    9 +++++++
 kernel/cgroup.c            |   12 ++++++----
 kernel/cpuset.c            |    2 +
 mm/memcontrol.c            |   20 ++++++++++++++-
 net/ipv4/tcp_memcontrol.c  |   52 +++++++++++++++++++++++++++++++++++++++------
 6 files changed, 83 insertions(+), 14 deletions(-)
```

--
1.7.7.6

Subject: [PATCH 1/3] don't attach a task to a dead cgroup
Posted by Glauber Costa on Thu, 19 Apr 2012 22:49:16 GMT
View Forum Message <> Reply to Message

Not all external callers of cgroup_attach_task() test to
see if the cgroup is still live - the internal callers at
cgroup.c does.

With this test in cgroup_attach_task, we can assure that
no tasks are ever moved to a cgroup that is past its

destruction point and was already marked as dead.

Signed-off-by: Glauber Costa <glommer@parallels.com>
CC: Tejun Heo <tj@kernel.org>
CC: Li Zefan <lizefan@huawei.com>
CC: Kamezawa Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com>
---
 kernel/cgroup.c | 3 +++
 1 files changed, 3 insertions(+), 0 deletions(-)

diff --git a/kernel/cgroup.c b/kernel/cgroup.c
index b61b938..932c318 100644
--- a/kernel/cgroup.c
+++ b/kernel/cgroup.c
@@ -1927,6 +1927,9 @@ int cgroup_attach_task(struct cgroup *cgrp, struct task_struct *tsk)
  struct cgroup_taskset tset = { };
  struct css_set *newcg;

+ if (cgroup_is_removed(cgrp))
+ return -ENODEV;
+
  /* @tsk either already exited or can't exit until the end */
  if (tsk->flags & PF_EXITING)
   return -ESRCH;
--
1.7.7.6

Subject: [PATCH 2/3] don't take cgroup_mutex in destroy()
Posted by Glauber Costa on Thu, 19 Apr 2012 22:49:17 GMT
View Forum Message <> Reply to Message

Most of the destroy functions are only doing very simple things
like freeing memory.

The ones who goes through lists and such, already use its own
locking for those.

* The cgroup itself won't go away until we free it, (after destroy)
* The parent won't go away because we hold a reference count
* There are no more tasks in the cgroup, and the cgroup is declared
  dead (cgroup_is_removed() == true)

For the blk-cgroup and the cpusets, I got the impression that the mutex
is still necessary.

For those, I grabbed it from within the destroy function itself.

If the maintainer for those subsystems consider it safe to remove
it, we can discuss it separately.

Signed-off-by: Glauber Costa <glommer@parallels.com>
CC: Tejun Heo <tj@kernel.org>
CC: Li Zefan <lizefan@huawei.com>
CC: Kamezawa Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com>
CC: Vivek Goyal <vgoyal@redhat.com>
---
 block/blk-cgroup.c | 2 ++
 kernel/cgroup.c | 9 +++++-----
 kernel/cpuset.c | 2 ++
 3 files changed, 8 insertions(+), 5 deletions(-)

diff --git a/block/blk-cgroup.c b/block/blk-cgroup.c
index 126c341..477463f 100644
--- a/block/blk-cgroup.c
+++ b/block/blk-cgroup.c
@@ -1527,6 +1527,7 @@ static void blkiocg_destroy(struct cgroup *cgroup)
  struct blkio_policy_type *blkiop;
  struct blkio_policy_node *pn, *pntmp;

+ cgroup_lock();
  rcu_read_lock();
  do {
   spin_lock_irqsave(&blkcg->lock, flags);
@@ -1566,6 +1567,7 @@ static void blkiocg_destroy(struct cgroup *cgroup)
  rcu_read_unlock();
  if (blkcg != &blkio_root_cgroup)
   kfree(blkcg);
+ cgroup_unlock();
 }

 static struct cgroup_subsys_state *blkiocg_create(struct cgroup *cgroup)
diff --git a/kernel/cgroup.c b/kernel/cgroup.c
index 932c318..976d332 100644
--- a/kernel/cgroup.c
+++ b/kernel/cgroup.c
@@ -869,13 +869,13 @@ static void cgroup_diput(struct dentry *dentry, struct inode *inode)
    * agent */
   synchronize_rcu();

- mutex_lock(&cgroup_mutex);
   /*
    * Release the subsystem state objects.
    */
   for_each_subsys(cgrp->root, ss)
    ss->destroy(cgrp);

```
+  mutex_lock(&cgroup_mutex);
   cgrp->root->number_of_cgroups--;
   mutex_unlock(&cgroup_mutex);

@@ -3994,13 +3994,12 @@ static long cgroup_create(struct cgroup *parent, struct dentry
*dentry,

  err_destroy:

+ mutex_unlock(&cgroup_mutex);
  for_each_subsys(root, ss) {
   if (cgrp->subsys[ss->subsys_id])
    ss->destroy(cgrp);
  }

- mutex_unlock(&cgroup_mutex);
-
  /* Release the reference count that we took on the superblock */
  deactivate_super(sb);

@@ -4349,9 +4348,9 @@ int __init_or_module cgroup_load_subsys(struct cgroup_subsys *ss)
   int ret = cgroup_init_idr(ss, css);
   if (ret) {
    dummytop->subsys[ss->subsys_id] = NULL;
+   mutex_unlock(&cgroup_mutex);
    ss->destroy(dummytop);
    subsys[i] = NULL;
-   mutex_unlock(&cgroup_mutex);
    return ret;
   }
  }
@@ -4447,10 +4446,10 @@ void cgroup_unload_subsys(struct cgroup_subsys *ss)
   * pointer to find their state. note that this also takes care of
   * freeing the css_id.
   */
+ mutex_unlock(&cgroup_mutex);
  ss->destroy(dummytop);
  dummytop->subsys[ss->subsys_id] = NULL;

- mutex_unlock(&cgroup_mutex);
 }
 EXPORT_SYMBOL_GPL(cgroup_unload_subsys);

diff --git a/kernel/cpuset.c b/kernel/cpuset.c
index 8c8bd65..3cd4916 100644
--- a/kernel/cpuset.c
+++ b/kernel/cpuset.c
```

```
@@ -1862,10 +1862,12 @@ static void cpuset_destroy(struct cgroup *cont)
 {
  struct cpuset *cs = cgroup_cs(cont);

+ cgroup_lock();
  if (is_sched_load_balance(cs))
   update_flag(CS_SCHED_LOAD_BALANCE, cs, 0);

  number_of_cpusets--;
+ cgroup_unlock();
  free_cpumask_var(cs->cpus_allowed);
  kfree(cs);
 }
--
1.7.7.6
```

Subject: [PATCH 3/3] decrement static keys on real destroy time
Posted by Glauber Costa on Thu, 19 Apr 2012 22:49:18 GMT
View Forum Message <> Reply to Message

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;
   }
   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;
 }
--
1.7.7.6
```

Subject: Re: [PATCH 1/3] don't attach a task to a dead cgroup
Posted by Tejun Heo on Thu, 19 Apr 2012 22:53:22 GMT
View Forum Message <> Reply to Message

On Thu, Apr 19, 2012 at 07:49:16PM -0300, Glauber Costa wrote:
> Not all external callers of cgroup_attach_task() test to
> see if the cgroup is still live - the internal callers at
> cgroup.c does.
>
> With this test in cgroup_attach_task, we can assure that
> no tasks are ever moved to a cgroup that is past its
> destruction point and was already marked as dead.
>
> Signed-off-by: Glauber Costa <glommer@parallels.com>
> CC: Tejun Heo <tj@kernel.org>
> CC: Li Zefan <lizefan@huawei.com>
> CC: Kamezawa Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com>
> ---
> kernel/cgroup.c | 3 +++

> 1 files changed, 3 insertions(+), 0 deletions(-)
>
> diff --git a/kernel/cgroup.c b/kernel/cgroup.c
> index b61b938..932c318 100644
> --- a/kernel/cgroup.c
> +++ b/kernel/cgroup.c
> @@ -1927,6 +1927,9 @@ int cgroup_attach_task(struct cgroup *cgrp, struct task_struct *tsk)
>   struct cgroup_taskset tset = { };
>   struct css_set *newcg;
>
> + if (cgroup_is_removed(cgrp))
> + return -ENODEV;
> +

Isn't the test in cgroup_lock_live_group() enough?

--
tejun

Subject: Re: [PATCH 0/3] Fix problem with static_key decrement
Posted by Tejun Heo on Thu, 19 Apr 2012 22:54:41 GMT
View Forum Message <> Reply to Message

On Thu, Apr 19, 2012 at 07:49:15PM -0300, Glauber Costa wrote:
> Hi,
>
> This is my proposed fix for the sock memcg static_key
> problem raised by Kamezawa. It works for me, but I would
> Kame, please confirm.

Please detail the problem. I don't follow what's the purpose here.

Thanks.

--
tejun

Subject: Re: [PATCH 2/3] don't take cgroup_mutex in destroy()
Posted by Tejun Heo on Thu, 19 Apr 2012 22:57:04 GMT
View Forum Message <> Reply to Message

On Thu, Apr 19, 2012 at 07:49:17PM -0300, Glauber Costa wrote:
> Most of the destroy functions are only doing very simple things
> like freeing memory.
>

> The ones who goes through lists and such, already use its own
> locking for those.
>
> * The cgroup itself won't go away until we free it, (after destroy)
> * The parent won't go away because we hold a reference count
> * There are no more tasks in the cgroup, and the cgroup is declared
>   dead (cgroup_is_removed() == true)
>
> For the blk-cgroup and the cpusets, I got the impression that the mutex
> is still necessary.
>
> For those, I grabbed it from within the destroy function itself.
>
> If the maintainer for those subsystems consider it safe to remove
> it, we can discuss it separately.

I really don't like cgroup_lock() usage spreading more. It's
something which should be contained in cgroup.c proper. I looked at
the existing users a while ago and they seemed to be compensating
deficencies in API, so, if at all possible, let's not spread the
disease.

Thanks.

--
tejun

Subject: Re: [PATCH 2/3] don't take cgroup_mutex in destroy()
Posted by Li Zefan on Fri, 20 Apr 2012 00:30:23 GMT
View Forum Message <> Reply to Message

Tejun Heo wrote:

> On Thu, Apr 19, 2012 at 07:49:17PM -0300, Glauber Costa wrote:
>> Most of the destroy functions are only doing very simple things
>> like freeing memory.
>>
>> The ones who goes through lists and such, already use its own
>> locking for those.
>>
>> * The cgroup itself won't go away until we free it, (after destroy)
>> * The parent won't go away because we hold a reference count
>> * There are no more tasks in the cgroup, and the cgroup is declared
>>   dead (cgroup_is_removed() == true)
>>
>> For the blk-cgroup and the cpusets, I got the impression that the mutex
>> is still necessary.

>>
>> For those, I grabbed it from within the destroy function itself.
>>
>> If the maintainer for those subsystems consider it safe to remove
>> it, we can discuss it separately.
>
> I really don't like cgroup_lock() usage spreading more. It's
> something which should be contained in cgroup.c proper. I looked at
> the existing users a while ago and they seemed to be compensating
> deficencies in API, so, if at all possible, let's not spread the
> disease.
>

Agreed. I used to do cleanups to remove cgroup_lock()s in subsystems
which are really not necessary.

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;
> }

Subject: Re: [PATCH 0/3] Fix problem with static_key decrement
Posted by Glauber Costa on Fri, 20 Apr 2012 15:01:50 GMT
View Forum Message <> Reply to Message

On 04/19/2012 07:54 PM, Tejun Heo wrote:
> On Thu, Apr 19, 2012 at 07:49:15PM -0300, Glauber Costa wrote:
>> Hi,
>>
>> This is my proposed fix for the sock memcg static_key
>> problem raised by Kamezawa. It works for me, but I would
>> Kame, please confirm.
>
> Please detail the problem. I don't follow what's the purpose here.
>

Ok.

1) Kame found the following bug: we were decrementing the jump label
when the socket limit was set back to unlimited. The problem is that the
sockets outlive the memcg, so we can only do that when the last
reference count is dropped. It is worth mentioning that kmem controller
for memcg will have the exact same problem - I am actually updating my
series with all the results of this discussion here.

2) If, however, there are no sockets in flight, mem_cgroup_put() during
->destroy() will be the last one, and the decrementing will happen there.

3) static_key updates cannot happen with the cgroup_mutex held. This is
because cpusets hold it from within the cpu_hotplug.lock - that
static_keys take through get_online_cpus() in its cpu hotplug handler.

4) Looking at the cpusets code, it really seems necessary, at least by now.

5) Deferring all this to worker threads as you suggested in the cpu
thread - that has a similar problem - can solve this problem, but in
general, will create tons of others, like windows of inconsistent
information.

That's basically it.

Subject: Re: [PATCH 2/3] don't take cgroup_mutex in destroy()
Posted by Glauber Costa on Fri, 20 Apr 2012 15:04:11 GMT

On 04/19/2012 07:57 PM, Tejun Heo wrote:
> On Thu, Apr 19, 2012 at 07:49:17PM -0300, Glauber Costa wrote:
>> Most of the destroy functions are only doing very simple things
>> like freeing memory.
>>
>> The ones who goes through lists and such, already use its own
>> locking for those.
>>
>> * The cgroup itself won't go away until we free it, (after destroy)
>> * The parent won't go away because we hold a reference count
>> * There are no more tasks in the cgroup, and the cgroup is declared
>>   dead (cgroup_is_removed() == true)
>>
>> For the blk-cgroup and the cpusets, I got the impression that the mutex
>> is still necessary.
>>
>> For those, I grabbed it from within the destroy function itself.
>>
>> If the maintainer for those subsystems consider it safe to remove
>> it, we can discuss it separately.
>
> I really don't like cgroup_lock() usage spreading more. It's
> something which should be contained in cgroup.c proper. I looked at
> the existing users a while ago and they seemed to be compensating
> deficencies in API, so, if at all possible, let's not spread the
> disease.

Well, I can dig deeper and see if they are really needed. I don't know
cpusets and blkcg *that* well, that's why I took them there, hoping that
someone could enlighten me, maybe they aren't really needed even now.

I agree with the compensating: As I mentioned, most of them are already
taking other kinds of lock to protect their structures, which is the
right thing to do.

There were only two or three spots in cpusets and blkcg where I wasn't
that sure that we could drop the lock... What do you say about that ?

Subject: Re: [PATCH 1/3] don't attach a task to a dead cgroup
Posted by Glauber Costa on Fri, 20 Apr 2012 15:05:31 GMT

On 04/19/2012 07:53 PM, Tejun Heo wrote:
> On Thu, Apr 19, 2012 at 07:49:16PM -0300, Glauber Costa wrote:
>> Not all external callers of cgroup_attach_task() test to

>> see if the cgroup is still live - the internal callers at
>> cgroup.c does.
>>
>> With this test in cgroup_attach_task, we can assure that
>> no tasks are ever moved to a cgroup that is past its
>> destruction point and was already marked as dead.
>>
>> Signed-off-by: Glauber Costa<glommer@parallels.com>
>> CC: Tejun Heo<tj@kernel.org>
>> CC: Li Zefan<lizefan@huawei.com>
>> CC: Kamezawa Hiroyuki<kamezawa.hiroyu@jp.fujitsu.com>
>> ---
>> kernel/cgroup.c | 3 +++
>> 1 files changed, 3 insertions(+), 0 deletions(-)
>>
>> diff --git a/kernel/cgroup.c b/kernel/cgroup.c
>> index b61b938..932c318 100644
>> --- a/kernel/cgroup.c
>> +++ b/kernel/cgroup.c
>> @@ -1927,6 +1927,9 @@ int cgroup_attach_task(struct cgroup *cgrp, struct task_struct *tsk)
>> struct cgroup_taskset tset = { };
>> struct css_set *newcg;
>>
>> + if (cgroup_is_removed(cgrp))
>> + return -ENODEV;
>> +
>
> Isn't the test in cgroup_lock_live_group() enough?
>
Yes, when it is done.

Not all callers take it, specially the external ones.

Subject: Re: [PATCH 3/3] decrement static keys on real destroy time
Posted by Glauber Costa on Fri, 20 Apr 2012 19:39:00 GMT
View Forum Message <> Reply to Message

On 04/20/2012 04:38 AM, KAMEZAWA Hiroyuki wrote:
>> 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
>

Kame,

You are right.

The first update needs to be done after the jump label activation as
well. I got myself confused with the two flags =(

I will repost with this fixed once I get into agreement with Tejun and
Li about the lockless ->destroy()

Subject: Re: [PATCH 2/3] don't take cgroup_mutex in destroy()
Posted by Li Zefan on Sat, 21 Apr 2012 06:47:16 GMT
View Forum Message <> Reply to Message

Glauber Costa wrote:

> On 04/19/2012 07:57 PM, Tejun Heo wrote:
>> On Thu, Apr 19, 2012 at 07:49:17PM -0300, Glauber Costa wrote:
>>> Most of the destroy functions are only doing very simple things
>>> like freeing memory.
>>>
>>> The ones who goes through lists and such, already use its own
>>> locking for those.
>>>
>>> * The cgroup itself won't go away until we free it, (after destroy)
>>> * The parent won't go away because we hold a reference count
>>> * There are no more tasks in the cgroup, and the cgroup is declared
>>>   dead (cgroup_is_removed() == true)
>>>
>>> For the blk-cgroup and the cpusets, I got the impression that the mutex
>>> is still necessary.
>>>
>>> For those, I grabbed it from within the destroy function itself.
>>>
>>> If the maintainer for those subsystems consider it safe to remove
>>> it, we can discuss it separately.
>>
>> I really don't like cgroup_lock() usage spreading more. It's
>> something which should be contained in cgroup.c proper. I looked at

>> the existing users a while ago and they seemed to be compensating
>> deficencies in API, so, if at all possible, let's not spread the
>> disease.
>
> Well, I can dig deeper and see if they are really needed. I don't know cpusets and blkcg *that* well, that's why I took them there, hoping that someone could enlighten me, maybe they aren't really needed even now.
>
> I agree with the compensating: As I mentioned, most of them are already taking other kinds of lock to protect their structures, which is the right thing to do.
>
> There were only two or three spots in cpusets and blkcg where I wasn't that sure that we could drop the lock... What do you say about that ?
> .

We can drop cgroup_mutex for cpusets with changes like this:

(Note: as I'm not able to get the latest code at this momment, this patch is based on 3.0.)

There are several places reading number_of_cpusets, but no one holds cgroup_mutex, except the one in generate_sched_domains(). With this patch, both cpuset_create() and generate_sched_domains() are still holding cgroup_mutex, so it's safe.

```
--- linux-kernel/kernel/cpuset.c.orig 2012-04-21 01:55:57.000000000 -0400
+++ linux-kernel/kernel/cpuset.c 2012-04-21 02:30:53.000000000 -0400
@@ -1876,7 +1876,9 @@ static struct cgroup_subsys_state *cpuse
  cs->relax_domain_level = -1;

  cs->parent = parent;
+ mutex_lock(&callback_mutex);
  number_of_cpusets++;
+ mutex_unlock(&callback_mutex);
  return &cs->css ;
 }

@@ -1890,10 +1892,18 @@ static void cpuset_destroy(struct cgroup
 {
  struct cpuset *cs = cgroup_cs(cont);

- if (is_sched_load_balance(cs))
+ if (is_sched_load_balance(cs)) {
+  /*
+   * This cpuset is under destruction, so no one else can
+   * modify it, so it's safe to call update_flag() without
+   * cgroup_lock.
+   */
   update_flag(CS_SCHED_LOAD_BALANCE, cs, 0);
+ }
```

```
+ mutex_lock(&callback_mutex);
  number_of_cpusets--;
+ mutex_lock(&callback_mutex);
  free_cpumask_var(cs->cpus_allowed);
  kfree(cs);
 }
```

Subject: Re: [PATCH 2/3] don't take cgroup_mutex in destroy()
Posted by Glauber Costa on Mon, 23 Apr 2012 16:36:46 GMT
View Forum Message <> Reply to Message

On 04/21/2012 03:47 AM, Li Zefan wrote:
> Glauber Costa wrote:
>
>> On 04/19/2012 07:57 PM, Tejun Heo wrote:
>>> On Thu, Apr 19, 2012 at 07:49:17PM -0300, Glauber Costa wrote:
>>>> Most of the destroy functions are only doing very simple things
>>>> like freeing memory.
>>>>
>>>> The ones who goes through lists and such, already use its own
>>>> locking for those.
>>>>
>>>> * The cgroup itself won't go away until we free it, (after destroy)
>>>> * The parent won't go away because we hold a reference count
>>>> * There are no more tasks in the cgroup, and the cgroup is declared
>>>>   dead (cgroup_is_removed() == true)
>>>>
>>>> For the blk-cgroup and the cpusets, I got the impression that the mutex
>>>> is still necessary.
>>>>
>>>> For those, I grabbed it from within the destroy function itself.
>>>>
>>>> If the maintainer for those subsystems consider it safe to remove
>>>> it, we can discuss it separately.
>>>
>>> I really don't like cgroup_lock() usage spreading more. It's
>>> something which should be contained in cgroup.c proper. I looked at
>>> the existing users a while ago and they seemed to be compensating
>>> deficencies in API, so, if at all possible, let's not spread the
>>> disease.
>>
>> Well, I can dig deeper and see if they are really needed. I don't know cpusets and blkcg *that* well, that's why I took them there, hoping that someone could enlighten me, maybe they aren't really needed even now.
>>
>> I agree with the compensating: As I mentioned, most of them are already taking other kinds of

lock to protect their structures, which is the right thing to do.
>>
>> There were only two or three spots in cpusets and blkcg where I wasn't that sure that we could drop the lock... What do you say about that ?
>> .
>
> We can drop cgroup_mutex for cpusets with changes like this:
>
> (Note: as I'm not able to get the latest code at this momment, this patch is based on 3.0.)
>
> There are several places reading number_of_cpusets, but no one holds cgroup_mutex, except
> the one in generate_sched_domains(). With this patch, both cpuset_create() and
> generate_sched_domains() are still holding cgroup_mutex, so it's safe.
>
> --- linux-kernel/kernel/cpuset.c.orig 2012-04-21 01:55:57.000000000 -0400
> +++ linux-kernel/kernel/cpuset.c 2012-04-21 02:30:53.000000000 -0400
> @@ -1876,7 +1876,9 @@ static struct cgroup_subsys_state *cpuse
> cs->relax_domain_level = -1;
>
> cs->parent = parent;
> + mutex_lock(&callback_mutex);
> number_of_cpusets++;
> + mutex_unlock(&callback_mutex);
> return&cs->css ;
> }
>
> @@ -1890,10 +1892,18 @@ static void cpuset_destroy(struct cgroup
> {
> struct cpuset *cs = cgroup_cs(cont);
>
> - if (is_sched_load_balance(cs))
> + if (is_sched_load_balance(cs)) {
> + /*
> + * This cpuset is under destruction, so no one else can
> + * modify it, so it's safe to call update_flag() without
> + * cgroup_lock.
> + */
> update_flag(CS_SCHED_LOAD_BALANCE, cs, 0);
> + }
>
> + mutex_lock(&callback_mutex);
> number_of_cpusets--;
> + mutex_lock(&callback_mutex);
> free_cpumask_var(cs->cpus_allowed);
> kfree(cs);
> }

I'll see if I can work something out.