Subject: [PATCH v2 0/5] Fix problem with static_key decrement
Posted by Glauber Costa on Mon, 23 Apr 2012 19:37:42 GMT
View Forum Message <> Reply to Message

This is my proposed fix for the sock memcg static_key
problem raised by Kamezawa. In a summary, the problem is
as follows:

We are decrementing the jump label when the socket limit is 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.

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

But 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.

Removing the cgroup_lock() dependency from cpusets is a lot harder, since the
code for generate_sched_domain() rely on that lock to be held, and it interact
with the cgroup core code by quite a bit.

The aim of this series is to make ->destroy() a stable point for jump label
updating, by calling it without the cgroup_mutex held. I believe it to be a
good thing in itself, since it removes a bit the reach of the almighty
cgroup_mutex.

I am ready to make any further modifications on this that you guys
deem necessary.

Thanks

Glauber Costa (5):
  don't attach a task to a dead cgroup
  blkcg: protect blkcg->policy_list
  change number_of_cpusets to an atomic
  don't take cgroup_mutex in destroy()
  decrement static keys on real destroy time

 block/blk-cgroup.c | 2 +
 include/linux/cpuset.h | 6 ++--
 include/net/sock.h | 9 ++++++++
 kernel/cgroup.c | 12 ++++++++----
 kernel/cpuset.c | 10 ++++++----
 mm/memcontrol.c | 20 ++++++++++++++++++++-

```
 net/ipv4/tcp_memcontrol.c |   50 ++++++++++++++++++++++++++++++++++++++------
 7 files changed, 87 insertions(+), 22 deletions(-)

--
1.7.7.6
```

Subject: [PATCH v2 1/5] don't attach a task to a dead cgroup
Posted by Glauber Costa on Mon, 23 Apr 2012 19:37:43 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 v2 2/5] blkcg: protect blkcg->policy_list
Posted by Glauber Costa on Mon, 23 Apr 2012 19:37:44 GMT

policy_list walks are protected with blkcg->lock everywhere else in the code.
In destroy(), they are not. Because destroy is usually protected with the
cgroup_mutex(), this is usually not a problem. But it would be a lot better
not to assume this.

Signed-off-by: Glauber Costa <glommer@parallels.com>
---
 block/blk-cgroup.c | 2 ++
 1 files changed, 2 insertions(+), 0 deletions(-)

```
diff --git a/block/blk-cgroup.c b/block/blk-cgroup.c
index 126c341..35fd701 100644
--- a/block/blk-cgroup.c
+++ b/block/blk-cgroup.c
@@ -1557,10 +1557,12 @@ static void blkiocg_destroy(struct cgroup *cgroup)
   spin_unlock(&blkio_list_lock);
  } while (1);

+ spin_lock_irqsave(&blkcg->lock, flags);
  list_for_each_entry_safe(pn, pntmp, &blkcg->policy_list, node) {
   blkio_policy_delete_node(pn);
   kfree(pn);
  }
+ spin_unlock_irqrestore(&blkcg->lock, flags);

  free_css_id(&blkio_subsys, &blkcg->css);
  rcu_read_unlock();
--
1.7.7.6
```

Subject: [PATCH v2 3/5] change number_of_cpusets to an atomic
Posted by Glauber Costa on Mon, 23 Apr 2012 19:37:45 GMT

This will allow us to call destroy() without holding the
cgroup_mutex(). Other important updates inside update_flags()
are protected by the callback_mutex.

We could protect this variable with the callback_mutex as well,
as suggested by Li Zefan, but we need to make sure we are protected
by that mutex at all times, and some of its updates happen inside the
cgroup_mutex - which means we would deadlock.

An atomic variable is not expensive, since it is seldom updated,
and protect us well.

```
Signed-off-by: Glauber Costa <glommer@parallels.com>
---
 include/linux/cpuset.h |    6 +++---
 kernel/cpuset.c        |   10 +++++++-----
 2 files changed, 8 insertions(+), 8 deletions(-)

diff --git a/include/linux/cpuset.h b/include/linux/cpuset.h
index 668f66b..9b3d468 100644
--- a/include/linux/cpuset.h
+++ b/include/linux/cpuset.h
@@ -16,7 +16,7 @@

 #ifdef CONFIG_CPUSETS

-extern int number_of_cpusets; /* How many cpusets are defined in system? */
+extern atomic_t number_of_cpusets; /* How many cpusets are defined in system? */

 extern int cpuset_init(void);
 extern void cpuset_init_smp(void);
@@ -33,13 +33,13 @@ extern int __cpuset_node_allowed_hardwall(int node, gfp_t gfp_mask);

 static inline int cpuset_node_allowed_softwall(int node, gfp_t gfp_mask)
 {
- return number_of_cpusets <= 1 ||
+ return atomic_read(&number_of_cpusets) <= 1 ||
   __cpuset_node_allowed_softwall(node, gfp_mask);
 }

 static inline int cpuset_node_allowed_hardwall(int node, gfp_t gfp_mask)
 {
- return number_of_cpusets <= 1 ||
+ return atomic_read(&number_of_cpusets) <= 1 ||
   __cpuset_node_allowed_hardwall(node, gfp_mask);
 }

diff --git a/kernel/cpuset.c b/kernel/cpuset.c
index 8c8bd65..65bfd6d 100644
--- a/kernel/cpuset.c
+++ b/kernel/cpuset.c
@@ -73,7 +73,7 @@ static struct workqueue_struct *cpuset_wq;
  * When there is only one cpuset (the root cpuset) we can
  * short circuit some hooks.
  */
-int number_of_cpusets __read_mostly;
+atomic_t number_of_cpusets __read_mostly;

 /* Forward declare cgroup structures */
```

```
 struct cgroup_subsys cpuset_subsys;
@@ -583,7 +583,7 @@ static int generate_sched_domains(cpumask_var_t **domains,
   goto done;
  }

- csa = kmalloc(number_of_cpusets * sizeof(cp), GFP_KERNEL);
+ csa = kmalloc(atomic_read(&number_of_cpusets) * sizeof(cp), GFP_KERNEL);
  if (!csa)
   goto done;
  csn = 0;
@@ -1848,7 +1848,7 @@ static struct cgroup_subsys_state *cpuset_create(struct cgroup *cont)
  cs->relax_domain_level = -1;

  cs->parent = parent;
- number_of_cpusets++;
+ atomic_inc(&number_of_cpusets);
  return &cs->css ;
 }

@@ -1865,7 +1865,7 @@ static void cpuset_destroy(struct cgroup *cont)
  if (is_sched_load_balance(cs))
   update_flag(CS_SCHED_LOAD_BALANCE, cs, 0);

- number_of_cpusets--;
+ atomic_dec(&number_of_cpusets);
  free_cpumask_var(cs->cpus_allowed);
  kfree(cs);
 }
@@ -1909,7 +1909,7 @@ int __init cpuset_init(void)
  if (!alloc_cpumask_var(&cpus_attach, GFP_KERNEL))
   BUG();

- number_of_cpusets = 1;
+ atomic_set(&number_of_cpusets, 1);
  return 0;
 }

--
1.7.7.6
```

Subject: [PATCH v2 4/5] don't take cgroup_mutex in destroy()
Posted by Glauber Costa on Mon, 23 Apr 2012 19:37:46 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)

[v2: don't cgroup_lock the freezer and blkcg]

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>
---
 kernel/cgroup.c | 9 +++++-----
 1 files changed, 4 insertions(+), 5 deletions(-)

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);

--
1.7.7.6
```

Subject: [PATCH v2 5/5] decrement static keys on real destroy time
Posted by Glauber Costa on Mon, 23 Apr 2012 19:37:47 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]
[v3: update the current active flag only after the static_key update]

Signed-off-by: Glauber Costa <glommer@parallels.com>
---
 include/net/sock.h | 9 ++++++++
 mm/memcontrol.c | 20 ++++++++++++++++++-
 net/ipv4/tcp_memcontrol.c | 50 ++------
 3 files changed, 70 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..e9c2710 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 +120,33 @@ 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) {
+  /*
+   * ->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);
+  cg_proto->active = true;
+ }

  return 0;
 }
--
1.7.7.6
```

Subject: Re: [PATCH v2 1/5] don't attach a task to a dead cgroup
Posted by KAMEZAWA Hiroyuki on Tue, 24 Apr 2012 02:20:45 GMT
View Forum Message <> Reply to Message

(2012/04/24 4:37), 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>

Reviewed-by: 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;

Subject: Re: [PATCH v2 3/5] change number_of_cpusets to an atomic
Posted by KAMEZAWA Hiroyuki on Tue, 24 Apr 2012 02:25:46 GMT
View Forum Message <> Reply to Message

(2012/04/24 4:37), Glauber Costa wrote:

> This will allow us to call destroy() without holding the
> cgroup_mutex(). Other important updates inside update_flags()
> are protected by the callback_mutex.

>
> We could protect this variable with the callback_mutex as well,
> as suggested by Li Zefan, but we need to make sure we are protected
> by that mutex at all times, and some of its updates happen inside the
> cgroup_mutex - which means we would deadlock.
>
> An atomic variable is not expensive, since it is seldom updated,
> and protect us well.
>
> Signed-off-by: Glauber Costa <glommer@parallels.com>

Reviewed-by: KAMEZAWA Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com>

Subject: Re: [PATCH v2 4/5] don't take cgroup_mutex in destroy()
Posted by KAMEZAWA Hiroyuki on Tue, 24 Apr 2012 02:31:54 GMT
View Forum Message <> Reply to Message

(2012/04/24 4:37), 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)
>
> [v2: don't cgroup_lock the freezer and blkcg]
>
> 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>
> ---
> kernel/cgroup.c | 9 +++++-----
> 1 files changed, 4 insertions(+), 5 deletions(-)
>
> 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;
```

>

I'm not fully sure but...dummytop->subsys[] update can be done without locking ?

Thanks,
-Kame

Subject: Re: [PATCH v2 5/5] decrement static keys on real destroy time
Posted by KAMEZAWA Hiroyuki on Tue, 24 Apr 2012 02:40:57 GMT
View Forum Message <> Reply to Message

(2012/04/24 4:37), 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]
> [v3: update the current active flag only after the static_key update]
>
> Signed-off-by: Glauber Costa <glommer@parallels.com>

Acked-by: KAMEZAWA Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com>

A small request below.

<snip>

```
> +   * ->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);
```

Could you explain for what this mutex is in above comment ?

Thanks,
-Kame

```
> +  if (!cg_proto->activated) {
> +   static_key_slow_inc(&memcg_socket_limit_enabled);
> +   cg_proto->activated = true;
> +  }
> +  mutex_unlock(&tcp_set_limit_mutex);
> +  cg_proto->active = true;
> + }
>
>   return 0;
>  }
```

Subject: Re: [PATCH v2 5/5] decrement static keys on real destroy time
Posted by Glauber Costa on Tue, 24 Apr 2012 11:41:42 GMT
View Forum Message <> Reply to Message

On 04/23/2012 11:40 PM, KAMEZAWA Hiroyuki wrote:
> (2012/04/24 4:37), 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]
>> [v3: update the current active flag only after the static_key update]
>>
>> Signed-off-by: Glauber Costa<glommer@parallels.com>
>
>
> Acked-by: KAMEZAWA Hiroyuki<kamezawa.hiroyu@jp.fujitsu.com>
>
> A small request below.
>
> <snip>
>
>
>> + * ->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);
>
>
> Could you explain for what this mutex is in above comment ?
>
This is explained at the site where the mutex is defined.
If you still want me to mention it here, or maybe expand the explanation
there, I surely can.

Subject: Re: [PATCH v2 4/5] don't take cgroup_mutex in destroy()
Posted by Glauber Costa on Tue, 24 Apr 2012 11:42:32 GMT
View Forum Message <> Reply to Message

On 04/23/2012 11:31 PM, KAMEZAWA Hiroyuki wrote:
> (2012/04/24 4:37), 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)
>>
>> [v2: don't cgroup_lock the freezer and blkcg]
>>
>> 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>
>> ---
>>   kernel/cgroup.c | 9 ++++-----
>>   1 files changed, 4 insertions(+), 5 deletions(-)
>>
>> 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;
>>
>
> I'm not fully sure but...dummytop->subsys[] update can be done without locking ?
>
I don't see a reason why updates to subsys[] after destruction shouldn't
be safe. But maybe I am wrong.

Tejun? Li?

Subject: Re: [PATCH v2 3/5] change number_of_cpusets to an atomic
Posted by Christoph Lameter on Tue, 24 Apr 2012 15:02:02 GMT
View Forum Message <> Reply to Message

On Mon, 23 Apr 2012, Glauber Costa wrote:

> This will allow us to call destroy() without holding the
> cgroup_mutex(). Other important updates inside update_flags()
> are protected by the callback_mutex.
>
> We could protect this variable with the callback_mutex as well,
> as suggested by Li Zefan, but we need to make sure we are protected
> by that mutex at all times, and some of its updates happen inside the
> cgroup_mutex - which means we would deadlock.

Would this not also be a good case to introduce static branching?

number_of_cpusets is used to avoid going through unnecessary processing
should there be no cpusets in use.

Subject: Re: [PATCH v2 3/5] change number_of_cpusets to an atomic
Posted by Glauber Costa on Tue, 24 Apr 2012 16:15:36 GMT
View Forum Message <> Reply to Message

On 04/24/2012 12:02 PM, Christoph Lameter wrote:
> On Mon, 23 Apr 2012, Glauber Costa wrote:
>
>> This will allow us to call destroy() without holding the
>> cgroup_mutex(). Other important updates inside update_flags()
>> are protected by the callback_mutex.
>>
>> We could protect this variable with the callback_mutex as well,

>> as suggested by Li Zefan, but we need to make sure we are protected
>> by that mutex at all times, and some of its updates happen inside the
>> cgroup_mutex - which means we would deadlock.
>
> Would this not also be a good case to introduce static branching?
>
> number_of_cpusets is used to avoid going through unnecessary processing
> should there be no cpusets in use.
>
Well,

static branches comes with a set of problems themselves, so I usually
prefer to use them only in places where we don't want to pay even a
cache miss if we can avoid, or a function call, or anything like that -
like the slub cache alloc as you may have seen in my kmem memcg series.

It doesn't seem to be the case here.

Subject: Re: [PATCH v2 3/5] change number_of_cpusets to an atomic
Posted by Christoph Lameter on Tue, 24 Apr 2012 16:24:14 GMT
View Forum Message <> Reply to Message

On Tue, 24 Apr 2012, Glauber Costa wrote:

> > Would this not also be a good case to introduce static branching?
> >
> > number_of_cpusets is used to avoid going through unnecessary processing
> > should there be no cpusets in use.
>
> static branches comes with a set of problems themselves, so I usually prefer
> to use them only in places where we don't want to pay even a cache miss if we
> can avoid, or a function call, or anything like that - like the slub cache
> alloc as you may have seen in my kmem memcg series.
>
> It doesn't seem to be the case here.

How did you figure that? number_of_cpusets was introduced exactly because
the functions are used in places where we do not pay the cost of calling
__cpuset_node_allowed_soft/hardwall. Have a look at these. They may take
locks etc etc in critical allocation paths

Subject: Re: [PATCH v2 3/5] change number_of_cpusets to an atomic
Posted by Glauber Costa on Tue, 24 Apr 2012 16:30:05 GMT
View Forum Message <> Reply to Message

On 04/24/2012 01:24 PM, Christoph Lameter wrote:
> On Tue, 24 Apr 2012, Glauber Costa wrote:
>
>>> Would this not also be a good case to introduce static branching?
>>>
>>> number_of_cpusets is used to avoid going through unnecessary processing
>>> should there be no cpusets in use.
>>
>> static branches comes with a set of problems themselves, so I usually prefer
>> to use them only in places where we don't want to pay even a cache miss if we
>> can avoid, or a function call, or anything like that - like the slub cache
>> alloc as you may have seen in my kmem memcg series.
>>
>> It doesn't seem to be the case here.
>
> How did you figure that? number_of_cpusets was introduced exactly because
> the functions are used in places where we do not pay the cost of calling
> __cpuset_node_allowed_soft/hardwall. Have a look at these. They may take
> locks etc etc in critical allocation paths
I am not arguing that.

You want to avoid the cost of processing a function, that's fair.
(Note that by "function call cost" I don't mean the cost of processing a
function, but the cost of a (potentially empty) function call.)
The real question is: Are you okay with the cost of a branch + a global
variable (which is almost read only) fetch?

The test of a global variable can - and do as of right now - avoid all
the expensive operations like locking, sleeping, etc, and if you don't
need to squeeze every nanosecond you can, they are often simpler - and
therefore better - than static branching.

Just to mention one point I am coming across these days - that initiated
all this: static patching holds the cpu_hotplug.lock. So it can't be
called if you hold any lock that has been already held under the
cpu_hotplug.lock. This will probably mean any lock the cpuset cgroup
needs to take, because it is called - and to do a lot of things - from
the cpu hotplug handler, that holds the cpu_hotplug.lock.

So if if were a case of simple static branch usage, I am not opposed to
it. But I foresee it getting so complicated, that a global variable
seems to do the job we need just fine.

Subject: Re: [PATCH v2 3/5] change number_of_cpusets to an atomic
Posted by Christoph Lameter on Tue, 24 Apr 2012 18:27:09 GMT
View Forum Message <> Reply to Message

On Tue, 24 Apr 2012, Glauber Costa wrote:

> > > It doesn't seem to be the case here.
> >
> > How did you figure that? number_of_cpusets was introduced exactly because
> > the functions are used in places where we do not pay the cost of calling
> > __cpuset_node_allowed_soft/hardwall. Have a look at these. They may take
> > locks etc etc in critical allocation paths
> I am not arguing that.
>
> You want to avoid the cost of processing a function, that's fair.
> (Note that by "function call cost" I don't mean the cost of processing a
> function, but the cost of a (potentially empty) function call.)
> The real question is: Are you okay with the cost of a branch + a global
> variable (which is almost read only) fetch?

No and that is why the static branching comes in. It takes away the global
read of the number_of_cpusets variable in the critical paths.

> The test of a global variable can - and do as of right now - avoid all the
> expensive operations like locking, sleeping, etc, and if you don't need to
> squeeze every nanosecond you can, they are often simpler - and therefore
> better - than static branching.

Better than static branching? This is in critical VM functions and
reducing the cache footprint there is good for everyone.

> Just to mention one point I am coming across these days - that initiated all
> this: static patching holds the cpu_hotplug.lock. So it can't be called if you
> hold any lock that has been already held under the cpu_hotplug.lock. This will
> probably mean any lock the cpuset cgroup needs to take, because it is called -
> and to do a lot of things - from the cpu hotplug handler, that holds the
> cpu_hotplug.lock.

Transitions from one to two cpusets are rare and are only done when a
cpuset is created in the /dev/cpuset hierachy). You could move the
code modification outside of locks or defer action into an event
thread if there are locks in the way.

Subject: Re: [PATCH v2 5/5] decrement static keys on real destroy time
Posted by KAMEZAWA Hiroyuki on Wed, 25 Apr 2012 00:22:37 GMT
View Forum Message <> Reply to Message

(2012/04/24 20:41), Glauber Costa wrote:

> On 04/23/2012 11:40 PM, KAMEZAWA Hiroyuki wrote:
>> (2012/04/24 4:37), 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]
>>> [v3: update the current active flag only after the static_key update]
>>>
>>> Signed-off-by: Glauber Costa<glommer@parallels.com>
>>
>>
>> Acked-by: KAMEZAWA Hiroyuki<kamezawa.hiroyu@jp.fujitsu.com>
>>
>> A small request below.
>>
>> <snip>
>>
>>
>>> +   * ->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);
>>
>>
>> Could you explain for what this mutex is in above comment ?
>>
> This is explained at the site where the mutex is defined.
> If you still want me to mention it here, or maybe expand the explanation
> there, I surely can.
>

Ah, I think it's better to mention one more complicated race.
Let me explain.

Assume we don't have tcp_set_limit_mutex. And jump_label is not activated yet
i.e. memcg_socket_limit_enabled->count == 0.

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.
```

Then, without mutex, activated/active may be set 'true' before the end
of jump_label modification.

Thanks,
-Kame

Subject: Re: [PATCH v2 4/5] don't take cgroup_mutex in destroy()
Posted by Li Zefan on Wed, 25 Apr 2012 08:01:03 GMT
View Forum Message <> Reply to Message

Glauber Costa wrote:

> On 04/23/2012 11:31 PM, KAMEZAWA Hiroyuki wrote:

>> (2012/04/24 4:37), 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)
>>>
>>> [v2: don't cgroup_lock the freezer and blkcg]
>>>
>>> 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>
>>> ---
>>>   kernel/cgroup.c | 9 +++++-----
>>>   1 files changed, 4 insertions(+), 5 deletions(-)
>>>
>>> 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;
>>>
>>
>> I'm not fully sure but...dummytop->subsys[] update can be done without locking ?
>>
> I don't see a reason why updates to subsys[] after destruction shouldn't
> be safe. But maybe I am wrong.
>
> Tejun? Li?
>
```

It's safe for dummpytop->subsys[], but it makes the code a bit subtle.

The worst part is, it's not safe to NULLify subsys[i] without cgroup_mutex. It should be ok to do that before calling ->destroy(), but again the code becomes a bit subtler.