Subject: [PATCH v4 0/3] fix problem with static_branch() for sock memcg
Posted by Glauber Costa on Thu, 26 Apr 2012 22:51:04 GMT
View Forum Message <> Reply to Message

Hi,

While trying to fulfill's Christoph's request for using static_branches
to do part of the role of number_of_cpusets in the cpuset cgroup, I took
a much more extensive look at the cpuset code (Thanks Christoph).

I started to feel that removing the cgroup_lock() from cpuset's
destroy is not as safe as I first imagined. At the very best, is not safe
enough to be bundled in a bugfix and deserves its own analysis.

I started then to consider another approach. While I voiced many times
that I would not like to do deferred updates for the static_branches, doing
that during destroy time would be perfectly acceptable IMHO (creation is
another story). In a summary, we are effectively calling the static_branch
updates only when the last reference to the memcg is gone. And that is
already asynchronous by nature, and we cope well with that.

In memcg, it turns out that we already do deferred freeing of the memcg
structure depending on the size of struct mem_cgroup.

My proposal is to always do that, and then we get a worker more or less
for free. Patch 3 is basically the same I had posted before, but without
the mutex lock protection, now in the static branch guaranteed interface.

Let me know if this is acceptable.

Thanks

Glauber Costa (3):
  make jump_labels wait while updates are in place
  Always free struct memcg through schedule_work()
  decrement static keys on real destroy time

 include/net/sock.h | 9 ++++++++
 kernel/jump_label.c | 13 ++++++++---
 mm/memcontrol.c | 50 ++++++++++++++++++++++++++++++++++++++-------
 net/ipv4/tcp_memcontrol.c | 34 ++++++++++++++++++++++++++++++-
 4 files changed, 82 insertions(+), 24 deletions(-)

--
1.7.7.6

Subject: [PATCH v4 1/3] make jump_labels wait while updates are in place
Posted by Glauber Costa on Thu, 26 Apr 2012 22:51:05 GMT
View Forum Message <> Reply to Message

In mem cgroup, we need to guarantee that two concurrent updates
of the jump_label interface wait for each other. IOW, we can't have
other updates returning while the first one is still patching the
kernel around, otherwise we'll race.

I believe this is something that can fit well in the static branch
API, without noticeable disadvantages:

* in the common case, it will be a quite simple lock/unlock operation
* Every context that calls static_branch_slow* already expects to be
  in sleeping context because it will mutex_lock the unlikely case.
* static_key_slow_inc is not expected to be called in any fast path,
  otherwise it would be expected to have quite a different name. Therefore
  the mutex + atomic combination instead of just an atomic should not kill
  us.

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: Johannes Weiner <hannes@cmpxchg.org>
CC: Michal Hocko <mhocko@suse.cz>
CC: Ingo Molnar <mingo@elte.hu>
CC: Jason Baron <jbaron@redhat.com>
---
 kernel/jump_label.c | 21 ++++++++++++----------
 1 files changed, 11 insertions(+), 10 deletions(-)

diff --git a/kernel/jump_label.c b/kernel/jump_label.c
index 4304919..5d09cb4 100644
--- a/kernel/jump_label.c
+++ b/kernel/jump_label.c
@@ -57,17 +57,16 @@ static void jump_label_update(struct static_key *key, int enable);

 void static_key_slow_inc(struct static_key *key)
 {
+ jump_label_lock();
  if (atomic_inc_not_zero(&key->enabled))
- return;
+ goto out;

- jump_label_lock();
- if (atomic_read(&key->enabled) == 0) {
- if (!jump_label_get_branch_default(key))
-   jump_label_update(key, JUMP_LABEL_ENABLE);

```
-  else
-   jump_label_update(key, JUMP_LABEL_DISABLE);
- }
+ if (!jump_label_get_branch_default(key))
+  jump_label_update(key, JUMP_LABEL_ENABLE);
+ else
+  jump_label_update(key, JUMP_LABEL_DISABLE);
  atomic_inc(&key->enabled);
+out:
  jump_label_unlock();
 }
 EXPORT_SYMBOL_GPL(static_key_slow_inc);
@@ -75,10 +74,11 @@ EXPORT_SYMBOL_GPL(static_key_slow_inc);
 static void __static_key_slow_dec(struct static_key *key,
   unsigned long rate_limit, struct delayed_work *work)
 {
- if (!atomic_dec_and_mutex_lock(&key->enabled, &jump_label_mutex)) {
+ jump_label_lock();
+ if (atomic_dec_and_test(&key->enabled)) {
   WARN(atomic_read(&key->enabled) < 0,
        "jump label: negative count!\n");
-  return;
+  goto out;
  }

  if (rate_limit) {
@@ -90,6 +90,7 @@ static void __static_key_slow_dec(struct static_key *key,
   else
    jump_label_update(key, JUMP_LABEL_ENABLE);
  }
+out:
  jump_label_unlock();
 }

--
1.7.7.6
```

Subject: [PATCH v4 2/3] Always free struct memcg through schedule_work()
Posted by Glauber Costa on Thu, 26 Apr 2012 22:51:06 GMT
View Forum Message <> Reply to Message

Right now we free struct memcg with kfree right after a
rcu grace period, but defer it if we need to use vfree() to get
rid of that memory area. We do that by need, because we need vfree
to be called in a process context.

This patch unifies this behavior, by ensuring that even kfree will

happen in a separate thread. The goal is to have a stable place to
call the upcoming jump label destruction function outside the realm
of the complicated and quite far-reaching cgroup lock (that can't be
held when calling neither the cpu_hotplug.lock nor the jump_label_mutex)

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: Johannes Weiner <hannes@cmpxchg.org>
CC: Michal Hocko <mhocko@suse.cz>
---
 mm/memcontrol.c | 24 ++++++++++++++++----------
 1 files changed, 13 insertions(+), 11 deletions(-)

diff --git a/mm/memcontrol.c b/mm/memcontrol.c
index 7832b4d..b0076cc 100644
--- a/mm/memcontrol.c
+++ b/mm/memcontrol.c
@@ -245,8 +245,8 @@ struct mem_cgroup {
    */
   struct rcu_head rcu_freeing;
   /*
-   * But when using vfree(), that cannot be done at
-   * interrupt time, so we must then queue the work.
+   * We also need some space for a worker in deferred freeing.
+   * By the time we call it, rcu_freeing is not longer in use.
    */
   struct work_struct work_freeing;
 };
@@ -4826,23 +4826,28 @@ out_free:
 }

 /*
- * Helpers for freeing a vzalloc()ed mem_cgroup by RCU,
+ * Helpers for freeing a kmalloc()ed/vzalloc()ed mem_cgroup by RCU,
  * but in process context. The work_freeing structure is overlaid
  * on the rcu_freeing structure, which itself is overlaid on memsw.
  */
-static void vfree_work(struct work_struct *work)
+static void free_work(struct work_struct *work)
 {
  struct mem_cgroup *memcg;
+ int size = sizeof(struct mem_cgroup);

  memcg = container_of(work, struct mem_cgroup, work_freeing);
- vfree(memcg);
+ if (size < PAGE_SIZE)

```
+  kfree(memcg);
+ else
+  vfree(memcg);
 }
-static void vfree_rcu(struct rcu_head *rcu_head)
+
+static void free_rcu(struct rcu_head *rcu_head)
 {
  struct mem_cgroup *memcg;

  memcg = container_of(rcu_head, struct mem_cgroup, rcu_freeing);
- INIT_WORK(&memcg->work_freeing, vfree_work);
+ INIT_WORK(&memcg->work_freeing, free_work);
  schedule_work(&memcg->work_freeing);
 }

@@ -4868,10 +4873,7 @@ static void __mem_cgroup_free(struct mem_cgroup *memcg)
   free_mem_cgroup_per_zone_info(memcg, node);

  free_percpu(memcg->stat);
- if (sizeof(struct mem_cgroup) < PAGE_SIZE)
-  kfree_rcu(memcg, rcu_freeing);
- else
-  call_rcu(&memcg->rcu_freeing, vfree_rcu);
+ call_rcu(&memcg->rcu_freeing, free_rcu);
 }

 static void mem_cgroup_get(struct mem_cgroup *memcg)
--
1.7.7.6
```

Subject: [PATCH v4 3/3] decrement static keys on real destroy time
Posted by Glauber Costa on Thu, 26 Apr 2012 22:51:07 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]
[v4: disarm_static_keys() inside free_work]
[v5: got rid of tcp_limit_mutex, now in the static_key interface]

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: Johannes Weiner <hannes@cmpxchg.org>
CC: Michal Hocko <mhocko@suse.cz>
---
 include/net/sock.h | 9 +++++++++
 mm/memcontrol.c | 26 ++++++++++++++++++++++++--
 net/ipv4/tcp_memcontrol.c | 34 ++++++++++++++++++++++++++++++++--
 3 files changed, 60 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 b0076cc..3d004ee 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);
@@ -4836,6 +4851,13 @@ 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. schedule_work()
+  * will guarantee this happens. Be careful if you need to move this
+  * disarm_static_keys around
+  */
+ disarm_static_keys(memcg);
  if (size < PAGE_SIZE)
   kfree(memcg);
  else
diff --git a/net/ipv4/tcp_memcontrol.c b/net/ipv4/tcp_memcontrol.c
index 1517037..81004df 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;
@@ -74,9 +76,6 @@ 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);

@@ -107,10 +106,31 @@ 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.
+   */
+  if (!cg_proto->activated) {
+   static_key_slow_inc(&memcg_socket_limit_enabled);
+   cg_proto->activated = true;
+  }
+  cg_proto->active = true;
+ }

  return 0;
 }
--
1.7.7.6
```

Subject: Re: [PATCH v4 1/3] make jump_labels wait while updates are in place
Posted by Steven Rostedt on Fri, 27 Apr 2012 00:43:06 GMT
View Forum Message <> Reply to Message

On Thu, Apr 26, 2012 at 07:51:05PM -0300, Glauber Costa wrote:
> In mem cgroup, we need to guarantee that two concurrent updates
> of the jump_label interface wait for each other. IOW, we can't have
> other updates returning while the first one is still patching the
> kernel around, otherwise we'll race.

But it shouldn't. The code as is should prevent that.

>
> I believe this is something that can fit well in the static branch
> API, without noticeable disadvantages:
>
> * in the common case, it will be a quite simple lock/unlock operation
> * Every context that calls static_branch_slow* already expects to be
> in sleeping context because it will mutex_lock the unlikely case.
> * static_key_slow_inc is not expected to be called in any fast path,
> otherwise it would be expected to have quite a different name. Therefore
> the mutex + atomic combination instead of just an atomic should not kill
> us.
>
> 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: Johannes Weiner <hannes@cmpxchg.org>
> CC: Michal Hocko <mhocko@suse.cz>
> CC: Ingo Molnar <mingo@elte.hu>
> CC: Jason Baron <jbaron@redhat.com>
> ---
> kernel/jump_label.c | 21 ++++++++++++----------
> 1 files changed, 11 insertions(+), 10 deletions(-)
>
> diff --git a/kernel/jump_label.c b/kernel/jump_label.c
> index 4304919..5d09cb4 100644
> --- a/kernel/jump_label.c
> +++ b/kernel/jump_label.c
> @@ -57,17 +57,16 @@ static void jump_label_update(struct static_key *key, int enable);
>
> void static_key_slow_inc(struct static_key *key)
> {
> + jump_label_lock();
> if (atomic_inc_not_zero(&key->enabled))
> - return;

If key->enabled is not zero, there's nothing to be done. As the jump
label has already been enabled. Note, the key->enabled doesn't get set
until after the jump label is updated. Thus, if two tasks were to come
in, they both would be locked on the jump_label_lock().

> + goto out;
>
> - jump_label_lock();
> - if (atomic_read(&key->enabled) == 0) {
> - if (!jump_label_get_branch_default(key))

> -   jump_label_update(key, JUMP_LABEL_ENABLE);
> - else
> -   jump_label_update(key, JUMP_LABEL_DISABLE);
> - }
> + if (!jump_label_get_branch_default(key))
> + jump_label_update(key, JUMP_LABEL_ENABLE);
> + else
> + jump_label_update(key, JUMP_LABEL_DISABLE);
>   atomic_inc(&key->enabled);
> +out:
>   jump_label_unlock();
> }
> EXPORT_SYMBOL_GPL(static_key_slow_inc);
> @@ -75,10 +74,11 @@ EXPORT_SYMBOL_GPL(static_key_slow_inc);
> static void __static_key_slow_dec(struct static_key *key,
>   unsigned long rate_limit, struct delayed_work *work)
> {
> - if (!atomic_dec_and_mutex_lock(&key->enabled, &jump_label_mutex)) {
> + jump_label_lock();
> + if (atomic_dec_and_test(&key->enabled)) {
>   WARN(atomic_read(&key->enabled) < 0,
>   "jump label: negative count!\n");
> - return;

Here, it is similar. If enabled is > 1, it wouldn't need to do anything,
thus it would dec the counter and return. But if it were one, then the
lock would be taken. and set to zero. There shouldn't be a case where
two tasks came in to set it less than zero (then something is
unbalanced).

Are you hitting the WARN_ON?

-- Steve

> + goto out;
>   }
>
>   if (rate_limit) {
> @@ -90,6 +90,7 @@ static void __static_key_slow_dec(struct static_key *key,
>   else
>   jump_label_update(key, JUMP_LABEL_ENABLE);
>   }
> +out:
>   jump_label_unlock();
> }
>
> --
> 1.7.7.6

Subject: Re: [PATCH v4 1/3] make jump_labels wait while updates are in place
Posted by KAMEZAWA Hiroyuki on Fri, 27 Apr 2012 01:05:02 GMT
View Forum Message <> Reply to Message

(2012/04/27 9:43), Steven Rostedt wrote:

> On Thu, Apr 26, 2012 at 07:51:05PM -0300, Glauber Costa wrote:
>> In mem cgroup, we need to guarantee that two concurrent updates
>> of the jump_label interface wait for each other. IOW, we can't have
>> other updates returning while the first one is still patching the
>> kernel around, otherwise we'll race.
>
> But it shouldn't. The code as is should prevent that.
>
>>
>> I believe this is something that can fit well in the static branch
>> API, without noticeable disadvantages:
>>
>> * in the common case, it will be a quite simple lock/unlock operation
>> * Every context that calls static_branch_slow* already expects to be
>> in sleeping context because it will mutex_lock the unlikely case.
>> * static_key_slow_inc is not expected to be called in any fast path,
>> otherwise it would be expected to have quite a different name. Therefore
>> the mutex + atomic combination instead of just an atomic should not kill
>> us.
>>
>> 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: Johannes Weiner <hannes@cmpxchg.org>
>> CC: Michal Hocko <mhocko@suse.cz>
>> CC: Ingo Molnar <mingo@elte.hu>
>> CC: Jason Baron <jbaron@redhat.com>
>> ---
>> kernel/jump_label.c | 21 ++++++++++++----------
>> 1 files changed, 11 insertions(+), 10 deletions(-)
>>
>> diff --git a/kernel/jump_label.c b/kernel/jump_label.c
>> index 4304919..5d09cb4 100644
>> --- a/kernel/jump_label.c
>> +++ b/kernel/jump_label.c
>> @@ -57,17 +57,16 @@ static void jump_label_update(struct static_key *key, int enable);
>>
>> void static_key_slow_inc(struct static_key *key)
>> {
>> + jump_label_lock();
>> if (atomic_inc_not_zero(&key->enabled))
>> - return;

>
> If key->enabled is not zero, there's nothing to be done. As the jump
> label has already been enabled. Note, the key->enabled doesn't get set
> until after the jump label is updated. Thus, if two tasks were to come
> in, they both would be locked on the jump_label_lock().
>

Ah, sorry, I misunderstood somthing. I'm sorry, Glauber.

-Kame

Subject: Re: [PATCH v4 1/3] make jump_labels wait while updates are in place
Posted by Jason Baron on Fri, 27 Apr 2012 13:53:21 GMT
View Forum Message <> Reply to Message

On Thu, Apr 26, 2012 at 08:43:06PM -0400, Steven Rostedt wrote:
> On Thu, Apr 26, 2012 at 07:51:05PM -0300, Glauber Costa wrote:
> > In mem cgroup, we need to guarantee that two concurrent updates
> > of the jump_label interface wait for each other. IOW, we can't have
> > other updates returning while the first one is still patching the
> > kernel around, otherwise we'll race.
>
> But it shouldn't. The code as is should prevent that.
>
> >
> > I believe this is something that can fit well in the static branch
> > API, without noticeable disadvantages:
> >
> > * in the common case, it will be a quite simple lock/unlock operation
> > * Every context that calls static_branch_slow* already expects to be
> >   in sleeping context because it will mutex_lock the unlikely case.
> > * static_key_slow_inc is not expected to be called in any fast path,
> >   otherwise it would be expected to have quite a different name. Therefore
> >   the mutex + atomic combination instead of just an atomic should not kill
> >   us.
> >
> > 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: Johannes Weiner <hannes@cmpxchg.org>
> > CC: Michal Hocko <mhocko@suse.cz>
> > CC: Ingo Molnar <mingo@elte.hu>
> > CC: Jason Baron <jbaron@redhat.com>
> > ---
> > kernel/jump_label.c | 21 ++++++++++++----------
> > 1 files changed, 11 insertions(+), 10 deletions(-)

> >
> > diff --git a/kernel/jump_label.c b/kernel/jump_label.c
> > index 4304919..5d09cb4 100644
> > --- a/kernel/jump_label.c
> > +++ b/kernel/jump_label.c
> > @@ -57,17 +57,16 @@ static void jump_label_update(struct static_key *key, int enable);
> >
> > void static_key_slow_inc(struct static_key *key)
> > {
> > + jump_label_lock();
> >   if (atomic_inc_not_zero(&key->enabled))
> > - return;
>
> If key->enabled is not zero, there's nothing to be done. As the jump
> label has already been enabled. Note, the key->enabled doesn't get set
> until after the jump label is updated. Thus, if two tasks were to come
> in, they both would be locked on the jump_label_lock().
>

Right, for x86 which uses stop_machine currently, we guarantee that all
cpus are going to see the updated code, before the inc of key->enabled.
However, other arches (sparc, mips, powerpc, for example), seem to be
using much lighter weight updates, which I hope are ok :)

Thanks,

-Jason

Subject: Re: [PATCH v4 1/3] make jump_labels wait while updates are in place
Posted by Steven Rostedt on Fri, 27 Apr 2012 14:07:10 GMT
View Forum Message <> Reply to Message

On Fri, 2012-04-27 at 09:53 -0400, Jason Baron wrote:

> Right, for x86 which uses stop_machine currently, we guarantee that all
> cpus are going to see the updated code, before the inc of key->enabled.
> However, other arches (sparc, mips, powerpc, for example), seem to be
> using much lighter weight updates, which I hope are ok :)

And x86 will soon be removing stop_machine() from its path too. But all
archs should perform some kind of memory sync after patching code. Thus
the update should be treated as if a memory barrier was added after it,
and before the inc.

-- Steve

Subject: Re: [PATCH v4 1/3] make jump_labels wait while updates are in place
Posted by Glauber Costa on Fri, 27 Apr 2012 14:59:37 GMT
View Forum Message <> Reply to Message

On 04/27/2012 10:53 AM, Jason Baron wrote:
> On Thu, Apr 26, 2012 at 08:43:06PM -0400, Steven Rostedt wrote:
>> On Thu, Apr 26, 2012 at 07:51:05PM -0300, Glauber Costa wrote:
>>> In mem cgroup, we need to guarantee that two concurrent updates
>>> of the jump_label interface wait for each other. IOW, we can't have
>>> other updates returning while the first one is still patching the
>>> kernel around, otherwise we'll race.
>>
>> But it shouldn't. The code as is should prevent that.
>>
>>>
>>> I believe this is something that can fit well in the static branch
>>> API, without noticeable disadvantages:
>>>
>>> * in the common case, it will be a quite simple lock/unlock operation
>>> * Every context that calls static_branch_slow* already expects to be
>>>   in sleeping context because it will mutex_lock the unlikely case.
>>> * static_key_slow_inc is not expected to be called in any fast path,
>>>   otherwise it would be expected to have quite a different name. Therefore
>>>   the mutex + atomic combination instead of just an atomic should not kill
>>>   us.
>>>
>>> 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: Johannes Weiner<hannes@cmpxchg.org>
>>> CC: Michal Hocko<mhocko@suse.cz>
>>> CC: Ingo Molnar<mingo@elte.hu>
>>> CC: Jason Baron<jbaron@redhat.com>
>>> ---
>>> kernel/jump_label.c | 21 ++++++++++++----------
>>> 1 files changed, 11 insertions(+), 10 deletions(-)
>>>
>>> diff --git a/kernel/jump_label.c b/kernel/jump_label.c
>>> index 4304919..5d09cb4 100644
>>> --- a/kernel/jump_label.c
>>> +++ b/kernel/jump_label.c
>>> @@ -57,17 +57,16 @@ static void jump_label_update(struct static_key *key, int enable);
>>>
>>> void static_key_slow_inc(struct static_key *key)
>>> {
>>> + jump_label_lock();
>>>   if (atomic_inc_not_zero(&key->enabled))
>>> - return;

>>
>> If key->enabled is not zero, there's nothing to be done. As the jump
>> label has already been enabled. Note, the key->enabled doesn't get set
>> until after the jump label is updated. Thus, if two tasks were to come
>> in, they both would be locked on the jump_label_lock().
>>
>

Okay, we seem to have been tricked by the usage of atomic while analyzing this. The fact that the atomic update happens after the code is patched seems enough to guarantee what we need, now that I read it again (and it seems so obvious =p)
