Subject: [RFC][-mm] Memory controller hierarchy support (v1)
Posted by Balbir Singh on Sat, 19 Apr 2008 05:35:51 GMT
View Forum Message <> Reply to Message

This applies on top of 2.6.25-rc8-mm2. The next version will be applied on top of 2.5.25-mm1.

This code is built on top of Pavel's hierarchy patches.

1. It propagates the charges upwards. A charge incurred on a cgroup is propagated to root. If any of the counters along the hierarchy is over limit, reclaim is initiated from the parent. We reclaim pages from the parent and the children below it. We also keep track of the last child from whom reclaim was done and start from there in the next reclaim.

TODO's/Open Questions

1. We need to hold cgroup_mutex while walking through the children in reclaim. We need to figure out the best way to do so. Should cgroups provide a helper function/macro for it?
2. Do not allow children to have a limit greater than their parents.
3. Allow the user to select if hierarchial support is required
4. Fine tune reclaim from children logic

Testing

This code was tested on a UML instance, where it compiled and worked well.

Signed-off-by: Pavel Emelyanov <xemul@openvz.org>
Signed-off-by: Balbir Singh <balbir@linux.vnet.ibm.com>

```
 include/linux/res_counter.h |   14 ++++
 kernel/res_counter.c        |   42 +++++++++++---
 mm/memcontrol.c             |  128 +++++++++++++++++++++++++++++++++++---------
 3 files changed, 148 insertions(+), 36 deletions(-)

diff -puN include/linux/res_counter.h~memory-controller-hierarchy-support
include/linux/res_counter.h
--- linux-2.6.25-rc8/include/linux/res_counter.h~memory-controller-hierarchy-support 2008-04-19
11:00:28.000000000 +0530
+++ linux-2.6.25-rc8-balbir/include/linux/res_counter.h 2008-04-19 11:00:28.000000000 +0530
@@ -43,6 +43,10 @@ struct res_counter {
   * the routines below consider this to be IRQ-safe
   */
  spinlock_t lock;
+ /*
```

```
+  * the parent counter. used for hierarchical resource accounting
+  */
+ struct res_counter *parent;
 };

 /**
@@ -82,7 +86,12 @@ enum {
  * helpers for accounting
  */

-void res_counter_init(struct res_counter *counter);
+/*
+ * the parent pointer is set only once - during the counter
+ * initialization. caller then must itself provide that this
+ * pointer is valid during the new counter lifetime
+ */
+void res_counter_init(struct res_counter *counter, struct res_counter *parent);

 /*
  * charge - try to consume more resource.
@@ -96,7 +105,8 @@ void res_counter_init(struct res_counter
  */

 int res_counter_charge_locked(struct res_counter *counter, unsigned long val);
-int res_counter_charge(struct res_counter *counter, unsigned long val);
+int res_counter_charge(struct res_counter *counter, unsigned long val,
+    struct res_counter **limit_exceeded_at);

 /*
  * uncharge - tell that some portion of the resource is released
diff -puN kernel/res_counter.c~memory-controller-hierarchy-support kernel/res_counter.c
--- linux-2.6.25-rc8/kernel/res_counter.c~memory-controller-hierarchy-support 2008-04-19
11:00:28.000000000 +0530
+++ linux-2.6.25-rc8-balbir/kernel/res_counter.c 2008-04-19 11:00:28.000000000 +0530
@@ -14,10 +14,11 @@
 #include <linux/res_counter.h>
 #include <linux/uaccess.h>

-void res_counter_init(struct res_counter *counter)
+void res_counter_init(struct res_counter *counter, struct res_counter *parent)
 {
  spin_lock_init(&counter->lock);
  counter->limit = (unsigned long long)LLONG_MAX;
+ counter->parent = parent;
 }

 int res_counter_charge_locked(struct res_counter *counter, unsigned long val)
@@ -33,14 +34,34 @@ int res_counter_charge_locked(struct res
```

```
  return 0;
 }

-int res_counter_charge(struct res_counter *counter, unsigned long val)
+int res_counter_charge(struct res_counter *counter, unsigned long val,
+    struct res_counter **limit_exceeded_at)
 {
  int ret;
  unsigned long flags;
+ struct res_counter *c, *unroll_c;

- spin_lock_irqsave(&counter->lock, flags);
- ret = res_counter_charge_locked(counter, val);
- spin_unlock_irqrestore(&counter->lock, flags);
+ *limit_exceeded_at = NULL;
+ local_irq_save(flags);
+ for (c = counter; c != NULL; c = c->parent) {
+   spin_lock(&c->lock);
+   ret = res_counter_charge_locked(c, val);
+   spin_unlock(&c->lock);
+   if (ret < 0) {
+     *limit_exceeded_at = c;
+     goto unroll;
+   }
+ }
+ local_irq_restore(flags);
+ return 0;
+
+unroll:
+ for (unroll_c = counter; unroll_c != c; unroll_c = unroll_c->parent) {
+   spin_lock(&unroll_c->lock);
+   res_counter_uncharge_locked(unroll_c, val);
+   spin_unlock(&unroll_c->lock);
+ }
+ local_irq_restore(flags);
  return ret;
 }

@@ -55,10 +76,15 @@ void res_counter_uncharge_locked(struct
 void res_counter_uncharge(struct res_counter *counter, unsigned long val)
 {
  unsigned long flags;
+ struct res_counter *c;

- spin_lock_irqsave(&counter->lock, flags);
- res_counter_uncharge_locked(counter, val);
- spin_unlock_irqrestore(&counter->lock, flags);
+ local_irq_save(flags);
```

```
+ for (c = counter; c != NULL; c = c->parent) {
+  spin_lock(&c->lock);
+  res_counter_uncharge_locked(c, val);
+  spin_unlock(&c->lock);
+ }
+ local_irq_restore(flags);
 }


diff -puN mm/memcontrol.c~memory-controller-hierarchy-support mm/memcontrol.c
--- linux-2.6.25-rc8/mm/memcontrol.c~memory-controller-hierarchy-support 2008-04-19
11:00:28.000000000 +0530
+++ linux-2.6.25-rc8-balbir/mm/memcontrol.c 2008-04-19 11:00:28.000000000 +0530
@@ -138,6 +138,13 @@ struct mem_cgroup {
   * statistics.
   */
  struct mem_cgroup_stat stat;
+
+ /*
+  * When reclaiming in a hierarchy, we need to know, which child
+  * we reclaimed last from. This helps us avoid hitting the first
+  * child over and over again
+  */
+ struct mem_cgroup *last_scanned_child;
 };
 static struct mem_cgroup init_mem_cgroup;

@@ -244,6 +251,12 @@ struct mem_cgroup *mem_cgroup_from_task(
     struct mem_cgroup, css);
 }

+static struct mem_cgroup*
+mem_cgroup_from_res_counter(struct res_counter *counter)
+{
+ return container_of(counter, struct mem_cgroup, res);
+}
+
 static inline int page_cgroup_locked(struct page *page)
 {
  return bit_spin_is_locked(PAGE_CGROUP_LOCK_BIT, &page->page_cgroup);
@@ -508,6 +521,86 @@ unsigned long mem_cgroup_isolate_pages(u
 }

 /*
+ * Charge mem and check if it is over it's limit. If so, reclaim from
+ * mem. This function can call itself recursively (as we walk up the
+ * hierarchy).
+ */
```

```
+static int mem_cgroup_charge_and_reclaim(struct mem_cgroup *mem, gfp_t gfp_mask)
+{
+ int ret = 0;
+ unsigned long nr_retries = MEM_CGROUP_RECLAIM_RETRIES;
+ struct res_counter *counter_over_limit;
+ struct mem_cgroup *mem_over_limit;
+ struct cgroup *cgroup, *cgrp, *curr_cgroup;
+
+ while (res_counter_charge(&mem->res, PAGE_SIZE, &counter_over_limit)) {
+  if (!(gfp_mask & __GFP_WAIT))
+   goto out;
+
+  /*
+   * Is one of our ancestors over limit ?
+   */
+  if (counter_over_limit) {
+   mem_over_limit =
+    mem_cgroup_from_res_counter(counter_over_limit);
+
+   if (mem != mem_over_limit)
+    ret = mem_cgroup_charge_and_reclaim(
+      mem_over_limit, gfp_mask);
+  }
+
+  if (try_to_free_mem_cgroup_pages(mem, gfp_mask))
+   continue;
+
+  /*
+   * try_to_free_mem_cgroup_pages() might not give us a full
+   * picture of reclaim. Some pages are reclaimed and might be
+   * moved to swap cache or just unmapped from the cgroup.
+   * Check the limit again to see if the reclaim reduced the
+   * current usage of the cgroup before giving up
+   */
+  if (res_counter_check_under_limit(&mem->res))
+   continue;
+
+  /*
+   * Now scan all children under the group. This is required
+   * to support hierarchies
+   */
+  if (!mem->last_scanned_child)
+   cgroup = list_first_entry(&mem->css.cgroup->children,
+      struct cgroup, sibling);
+  else
+   cgroup = mem->last_scanned_child->css.cgroup;
+
+  curr_cgroup = mem->css.cgroup;
```

```
+
+  /*
+   * Ideally we need to hold cgroup_mutex here
+   */
+  list_for_each_entry_safe_from(cgroup, cgrp,
+    &curr_cgroup->children, sibling) {
+   struct mem_cgroup *mem_child;
+
+   mem_child = mem_cgroup_from_cont(cgroup);
+   ret = try_to_free_mem_cgroup_pages(mem_child,
+         gfp_mask);
+   mem->last_scanned_child = mem_child;
+   if (ret == 0)
+    break;
+  }
+
+  if (!nr_retries--) {
+   mem_cgroup_out_of_memory(mem, gfp_mask);
+   ret = -ENOMEM;
+   break;
+  }
+ }
+
+out:
+ return ret;
+}
+
+/*
  * Charge the memory controller for page usage.
  * Return
  * 0 if the charge was successful
@@ -519,7 +612,6 @@ static int mem_cgroup_charge_common(stru
  struct mem_cgroup *mem;
  struct page_cgroup *pc;
  unsigned long flags;
- unsigned long nr_retries = MEM_CGROUP_RECLAIM_RETRIES;
  struct mem_cgroup_per_zone *mz;

  if (mem_cgroup_subsys.disabled)
@@ -570,28 +662,8 @@ retry:
  css_get(&mem->css);
  rcu_read_unlock();

- while (res_counter_charge(&mem->res, PAGE_SIZE)) {
-  if (!(gfp_mask & __GFP_WAIT))
-   goto out;
-
-  if (try_to_free_mem_cgroup_pages(mem, gfp_mask))
```

```
-   continue;
-
-  /*
-   * try_to_free_mem_cgroup_pages() might not give us a full
-   * picture of reclaim. Some pages are reclaimed and might be
-   * moved to swap cache or just unmapped from the cgroup.
-   * Check the limit again to see if the reclaim reduced the
-   * current usage of the cgroup before giving up
-   */
-  if (res_counter_check_under_limit(&mem->res))
-   continue;
-
-  if (!nr_retries--) {
-   mem_cgroup_out_of_memory(mem, gfp_mask);
-   goto out;
-  }
- }
+ if (mem_cgroup_charge_and_reclaim(mem, gfp_mask))
+  goto out;

  pc->ref_cnt = 1;
  pc->mem_cgroup = mem;
@@ -986,19 +1058,23 @@ static void free_mem_cgroup_per_zone_inf
 static struct cgroup_subsys_state *
 mem_cgroup_create(struct cgroup_subsys *ss, struct cgroup *cont)
 {
- struct mem_cgroup *mem;
+ struct mem_cgroup *mem, *parent;
  int node;

  if (unlikely((cont->parent) == NULL)) {
   mem = &init_mem_cgroup;
   page_cgroup_cache = KMEM_CACHE(page_cgroup, SLAB_PANIC);
- } else
+  parent = NULL;
+ } else {
   mem = kzalloc(sizeof(struct mem_cgroup), GFP_KERNEL);
+  parent = mem_cgroup_from_cont(cont->parent);
+ }

  if (mem == NULL)
   return ERR_PTR(-ENOMEM);

- res_counter_init(&mem->res);
+ res_counter_init(&mem->res, parent ? &parent->res : NULL);
+ mem->last_scanned_child = NULL;

  memset(&mem->info, 0, sizeof(mem->info));
```

_

--
Warm Regards,
Balbir Singh
Linux Technology Center
IBM, ISTL

_____
Containers mailing list
Containers@lists.linux-foundation.org
https://lists.linux-foundation.org/mailman/listinfo/containers

---

Subject: Re: [RFC][-mm] Memory controller hierarchy support (v1)
Posted by yamamoto on Sat, 19 Apr 2008 06:56:24 GMT
View Forum Message <> Reply to Message

```
> -int res_counter_charge(struct res_counter *counter, unsigned long val)
> +int res_counter_charge(struct res_counter *counter, unsigned long val,
> +   struct res_counter **limit_exceeded_at)
>  {
>   int ret;
>   unsigned long flags;
> + struct res_counter *c, *unroll_c;
>
> - spin_lock_irqsave(&counter->lock, flags);
> - ret = res_counter_charge_locked(counter, val);
> - spin_unlock_irqrestore(&counter->lock, flags);
> + *limit_exceeded_at = NULL;
> + local_irq_save(flags);
> + for (c = counter; c != NULL; c = c->parent) {
> +  spin_lock(&c->lock);
> +  ret = res_counter_charge_locked(c, val);
> +  spin_unlock(&c->lock);
> +  if (ret < 0) {
> +   *limit_exceeded_at = c;
> +   goto unroll;
> +  }
> + }
> + local_irq_restore(flags);
> + return 0;
> +
> +unroll:
> + for (unroll_c = counter; unroll_c != c; unroll_c = unroll_c->parent) {
> +  spin_lock(&unroll_c->lock);
> +  res_counter_uncharge_locked(unroll_c, val);
> +  spin_unlock(&unroll_c->lock);
```

> + }
> + local_irq_restore(flags);
>   return ret;
> }

i wonder how much performance impacts this involves.

it increases the number of atomic ops per charge/uncharge and
makes the common case (success) of every charge/uncharge in a system
touch a global (ie. root cgroup's) cachelines.

> + /*
> +   * Ideally we need to hold cgroup_mutex here
> +   */
> + list_for_each_entry_safe_from(cgroup, cgrp,
> +   &curr_cgroup->children, sibling) {
> +   struct mem_cgroup *mem_child;
> +
> +   mem_child = mem_cgroup_from_cont(cgroup);
> +   ret = try_to_free_mem_cgroup_pages(mem_child,
> +   gfp_mask);
> +   mem->last_scanned_child = mem_child;
> +   if (ret == 0)
> +   break;
> + }

if i read it correctly, it makes us hit the last child again and again.

i think you want to reclaim from all cgroups under the curr_cgroup
including eg. children's children.

YAMAMOTO Takashi

_____
Containers mailing list
Containers@lists.linux-foundation.org
https://lists.linux-foundation.org/mailman/listinfo/containers

---

Subject: Re: [RFC][-mm] Memory controller hierarchy support (v1)
Posted by Balbir Singh on Sat, 19 Apr 2008 08:34:00 GMT
View Forum Message <> Reply to Message

YAMAMOTO Takashi wrote:
>> -int res_counter_charge(struct res_counter *counter, unsigned long val)
>> +int res_counter_charge(struct res_counter *counter, unsigned long val,
>> +   struct res_counter **limit_exceeded_at)
>> {
>>   int ret;

```
>>   unsigned long flags;
>> + struct res_counter *c, *unroll_c;
>>
>> - spin_lock_irqsave(&counter->lock, flags);
>> - ret = res_counter_charge_locked(counter, val);
>> - spin_unlock_irqrestore(&counter->lock, flags);
>> + *limit_exceeded_at = NULL;
>> + local_irq_save(flags);
>> + for (c = counter; c != NULL; c = c->parent) {
>> +  spin_lock(&c->lock);
>> +  ret = res_counter_charge_locked(c, val);
>> +  spin_unlock(&c->lock);
>> +  if (ret < 0) {
>> +   *limit_exceeded_at = c;
>> +   goto unroll;
>> +  }
>> + }
>> + local_irq_restore(flags);
>> + return 0;
>> +
>> +unroll:
>> + for (unroll_c = counter; unroll_c != c; unroll_c = unroll_c->parent) {
>> +  spin_lock(&unroll_c->lock);
>> +  res_counter_uncharge_locked(unroll_c, val);
>> +  spin_unlock(&unroll_c->lock);
>> + }
>> + local_irq_restore(flags);
>>   return ret;
>>  }
>
> i wonder how much performance impacts this involves.
>
> it increases the number of atomic ops per charge/uncharge and
> makes the common case (success) of every charge/uncharge in a system
> touch a global (ie. root cgroup's) cachelines.
>
```

Yes, it does. I'll run some tests to see what the overhead looks like. The multi-hierarchy feature is very useful though and one of the TODOs is to make the feature user selectable (possibly at run-time)

```
>> +  /*
>> +   * Ideally we need to hold cgroup_mutex here
>> +   */
>> +  list_for_each_entry_safe_from(cgroup, cgrp,
>> +    &curr_cgroup->children, sibling) {
>> +   struct mem_cgroup *mem_child;
>> +
```

>> + mem_child = mem_cgroup_from_cont(cgroup);
>> + ret = try_to_free_mem_cgroup_pages(mem_child,
>> +   gfp_mask);
>> + mem->last_scanned_child = mem_child;
>> + if (ret == 0)
>> +   break;
>> + }
>
> if i read it correctly, it makes us hit the last child again and again.
>

Hmm.. it should probably be set at the beginining of the loop. I'll retest

> i think you want to reclaim from all cgroups under the curr_cgroup
> including eg. children's children.
>

Yes, good point, I should break out the function, so that we can work around the
recursion problem. Charging can cause further recursion, since we check for
last_counter.

> YAMAMOTO Takashi

--
Warm Regards,
Balbir Singh
Linux Technology Center
IBM, ISTL

_____
Containers mailing list
Containers@lists.linux-foundation.org
https://lists.linux-foundation.org/mailman/listinfo/containers

---

Subject: Re: [RFC][-mm] Memory controller hierarchy support (v1)
Posted by Pavel Emelianov on Sat, 19 Apr 2008 10:47:19 GMT
View Forum Message <> Reply to Message

Balbir Singh wrote:
> This applies on top of 2.6.25-rc8-mm2. The next version will be applied
> on top of 2.5.25-mm1.
>
> This code is built on top of Pavel's hierarchy patches.
>
> 1. It propagates the charges upwards. A charge incurred on a cgroup
>   is propagated to root. If any of the counters along the hierarchy

> is over limit, reclaim is initiated from the parent. We reclaim
> pages from the parent and the children below it. We also keep track
> of the last child from whom reclaim was done and start from there in
> the next reclaim.

Are you going to split this patch? As is it looks rather huge :)

> TODO's/Open Questions
>
> 1. We need to hold cgroup_mutex while walking through the children
> in reclaim. We need to figure out the best way to do so. Should
> cgroups provide a helper function/macro for it?
> 2. Do not allow children to have a limit greater than their parents.
> 3. Allow the user to select if hierarchial support is required
> 4. Fine tune reclaim from children logic
>
> Testing
>
> This code was tested on a UML instance, where it compiled and worked well.
>
> Signed-off-by: Pavel Emelyanov <xemul@openvz.org>
> Signed-off-by: Balbir Singh <balbir@linux.vnet.ibm.com>
> ---
>
> include/linux/res_counter.h | 14 ++++
> kernel/res_counter.c | 42 +++++++++++---
> mm/memcontrol.c | 128 ++++++++++++++++++++++++++++++++++++---------
> 3 files changed, 148 insertions(+), 36 deletions(-)
>
> diff -puN include/linux/res_counter.h~memory-controller-hierarchy-support include/linux/res_counter.h
> --- linux-2.6.25-rc8/include/linux/res_counter.h~memory-controller-hierarchy-support 2008-04-19 11:00:28.000000000 +0530
> +++ linux-2.6.25-rc8-balbir/include/linux/res_counter.h 2008-04-19 11:00:28.000000000 +0530
> @@ -43,6 +43,10 @@ struct res_counter {
> * the routines below consider this to be IRQ-safe
> */
> spinlock_t lock;
> + /*
> + * the parent counter. used for hierarchical resource accounting
> + */
> + struct res_counter *parent;
> };
>
> /**
> @@ -82,7 +86,12 @@ enum {
> * helpers for accounting
> */

>
> -void res_counter_init(struct res_counter *counter);
> +/*
> + * the parent pointer is set only once - during the counter
> + * initialization. caller then must itself provide that this
> + * pointer is valid during the new counter lifetime
> + */
> +void res_counter_init(struct res_counter *counter, struct res_counter *parent);
>
> /*
>   * charge - try to consume more resource.
> @@ -96,7 +105,8 @@ void res_counter_init(struct res_counter
>   */
>
> int res_counter_charge_locked(struct res_counter *counter, unsigned long val);
> -int res_counter_charge(struct res_counter *counter, unsigned long val);
> +int res_counter_charge(struct res_counter *counter, unsigned long val,
> +   struct res_counter **limit_exceeded_at);
>
> /*
>   * uncharge - tell that some portion of the resource is released
> diff -puN kernel/res_counter.c~memory-controller-hierarchy-support kernel/res_counter.c
> --- linux-2.6.25-rc8/kernel/res_counter.c~memory-controller-hierarchy-support 2008-04-19
11:00:28.000000000 +0530
> +++ linux-2.6.25-rc8-balbir/kernel/res_counter.c 2008-04-19 11:00:28.000000000 +0530
> @@ -14,10 +14,11 @@
> #include <linux/res_counter.h>
> #include <linux/uaccess.h>
>
> -void res_counter_init(struct res_counter *counter)
> +void res_counter_init(struct res_counter *counter, struct res_counter *parent)
> {
>   spin_lock_init(&counter->lock);
>   counter->limit = (unsigned long long)LLONG_MAX;
> + counter->parent = parent;
> }
>
> int res_counter_charge_locked(struct res_counter *counter, unsigned long val)
> @@ -33,14 +34,34 @@ int res_counter_charge_locked(struct res
>   return 0;
> }
>
> -int res_counter_charge(struct res_counter *counter, unsigned long val)
> +int res_counter_charge(struct res_counter *counter, unsigned long val,
> +   struct res_counter **limit_exceeded_at)
> {
>   int ret;
>   unsigned long flags;

```
> + struct res_counter *c, *unroll_c;
>
> - spin_lock_irqsave(&counter->lock, flags);
> - ret = res_counter_charge_locked(counter, val);
> - spin_unlock_irqrestore(&counter->lock, flags);
> + *limit_exceeded_at = NULL;
> + local_irq_save(flags);
> + for (c = counter; c != NULL; c = c->parent) {
> +  spin_lock(&c->lock);
> +  ret = res_counter_charge_locked(c, val);
> +  spin_unlock(&c->lock);
> +  if (ret < 0) {
> +   *limit_exceeded_at = c;
> +   goto unroll;
> +  }
> + }
> + local_irq_restore(flags);
> + return 0;
> +
> +unroll:
> + for (unroll_c = counter; unroll_c != c; unroll_c = unroll_c->parent) {
> +  spin_lock(&unroll_c->lock);
> +  res_counter_uncharge_locked(unroll_c, val);
> +  spin_unlock(&unroll_c->lock);
> + }
> + local_irq_restore(flags);
>   return ret;
>  }
>
> @@ -55,10 +76,15 @@ void res_counter_uncharge_locked(struct
>  void res_counter_uncharge(struct res_counter *counter, unsigned long val)
>  {
>   unsigned long flags;
> + struct res_counter *c;
>
> - spin_lock_irqsave(&counter->lock, flags);
> - res_counter_uncharge_locked(counter, val);
> - spin_unlock_irqrestore(&counter->lock, flags);
> + local_irq_save(flags);
> + for (c = counter; c != NULL; c = c->parent) {
> +  spin_lock(&c->lock);
> +  res_counter_uncharge_locked(c, val);
> +  spin_unlock(&c->lock);
> + }
> + local_irq_restore(flags);
>  }
>
>
```

> diff -puN mm/memcontrol.c~memory-controller-hierarchy-support mm/memcontrol.c
> --- linux-2.6.25-rc8/mm/memcontrol.c~memory-controller-hierarchy-support 2008-04-19 11:00:28.000000000 +0530
> +++ linux-2.6.25-rc8-balbir/mm/memcontrol.c 2008-04-19 11:00:28.000000000 +0530
> @@ -138,6 +138,13 @@ struct mem_cgroup {
> * statistics.
> */
> struct mem_cgroup_stat stat;
> +
> + /*
> + * When reclaiming in a hierarchy, we need to know, which child
> + * we reclaimed last from. This helps us avoid hitting the first
> + * child over and over again
> + */
> + struct mem_cgroup *last_scanned_child;
> };
> static struct mem_cgroup init_mem_cgroup;
>
> @@ -244,6 +251,12 @@ struct mem_cgroup *mem_cgroup_from_task(
> struct mem_cgroup, css);
> }
>
> +static struct mem_cgroup*
> +mem_cgroup_from_res_counter(struct res_counter *counter)
> +{
> + return container_of(counter, struct mem_cgroup, res);
> +}
> +
> static inline int page_cgroup_locked(struct page *page)
> {
> return bit_spin_is_locked(PAGE_CGROUP_LOCK_BIT, &page->page_cgroup);
> @@ -508,6 +521,86 @@ unsigned long mem_cgroup_isolate_pages(u
> }
>
> /*
> + * Charge mem and check if it is over it's limit. If so, reclaim from
> + * mem. This function can call itself recursively (as we walk up the
> + * hierarchy).
> + */
> +static int mem_cgroup_charge_and_reclaim(struct mem_cgroup *mem, gfp_t gfp_mask)
> +{
> + int ret = 0;
> + unsigned long nr_retries = MEM_CGROUP_RECLAIM_RETRIES;
> + struct res_counter *counter_over_limit;
> + struct mem_cgroup *mem_over_limit;
> + struct cgroup *cgroup, *cgrp, *curr_cgroup;
> +
> + while (res_counter_charge(&mem->res, PAGE_SIZE, &counter_over_limit)) {

```
> +  if (!(gfp_mask & __GFP_WAIT))
> +   goto out;
> +
> +  /*
> +   * Is one of our ancestors over limit ?
> +   */
> +  if (counter_over_limit) {
> +   mem_over_limit =
> +    mem_cgroup_from_res_counter(counter_over_limit);
> +
> +   if (mem != mem_over_limit)
> +    ret = mem_cgroup_charge_and_reclaim(
> +      mem_over_limit, gfp_mask);
> +  }
> +
> +  if (try_to_free_mem_cgroup_pages(mem, gfp_mask))
> +   continue;
> +
> +  /*
> +   * try_to_free_mem_cgroup_pages() might not give us a full
> +   * picture of reclaim. Some pages are reclaimed and might be
> +   * moved to swap cache or just unmapped from the cgroup.
> +   * Check the limit again to see if the reclaim reduced the
> +   * current usage of the cgroup before giving up
> +   */
> +  if (res_counter_check_under_limit(&mem->res))
> +   continue;
> +
> +  /*
> +   * Now scan all children under the group. This is required
> +   * to support hierarchies
> +   */
> +  if (!mem->last_scanned_child)
> +   cgroup = list_first_entry(&mem->css.cgroup->children,
> +      struct cgroup, sibling);
> +  else
> +   cgroup = mem->last_scanned_child->css.cgroup;
> +
> +  curr_cgroup = mem->css.cgroup;
> +
> +  /*
> +   * Ideally we need to hold cgroup_mutex here
> +   */
> +  list_for_each_entry_safe_from(cgroup, cgrp,
> +    &curr_cgroup->children, sibling) {
> +   struct mem_cgroup *mem_child;
> +
> +   mem_child = mem_cgroup_from_cont(cgroup);
```

```
> +   ret = try_to_free_mem_cgroup_pages(mem_child,
> +         gfp_mask);
> +   mem->last_scanned_child = mem_child;
> +   if (ret == 0)
> +    break;
> +  }
> +
> +  if (!nr_retries--) {
> +   mem_cgroup_out_of_memory(mem, gfp_mask);
> +   ret = -ENOMEM;
> +   break;
> +  }
> + }
> +
> +out:
> + return ret;
> +}
> +
> +/*
>   * Charge the memory controller for page usage.
>   * Return
>   * 0 if the charge was successful
> @@ -519,7 +612,6 @@ static int mem_cgroup_charge_common(stru
>   struct mem_cgroup *mem;
>   struct page_cgroup *pc;
>   unsigned long flags;
> - unsigned long nr_retries = MEM_CGROUP_RECLAIM_RETRIES;
>   struct mem_cgroup_per_zone *mz;
>
>   if (mem_cgroup_subsys.disabled)
> @@ -570,28 +662,8 @@ retry:
>   css_get(&mem->css);
>   rcu_read_unlock();
>
> - while (res_counter_charge(&mem->res, PAGE_SIZE)) {
> -  if (!(gfp_mask & __GFP_WAIT))
> -   goto out;
> -
> -  if (try_to_free_mem_cgroup_pages(mem, gfp_mask))
> -   continue;
> -
> -  /*
> -   * try_to_free_mem_cgroup_pages() might not give us a full
> -   * picture of reclaim. Some pages are reclaimed and might be
> -   * moved to swap cache or just unmapped from the cgroup.
> -   * Check the limit again to see if the reclaim reduced the
> -   * current usage of the cgroup before giving up
> -   */
```

```
> -  if (res_counter_check_under_limit(&mem->res))
> -   continue;
> -
> -  if (!nr_retries--) {
> -   mem_cgroup_out_of_memory(mem, gfp_mask);
> -   goto out;
> -  }
> - }
> + if (mem_cgroup_charge_and_reclaim(mem, gfp_mask))
> +  goto out;
>
>   pc->ref_cnt = 1;
>   pc->mem_cgroup = mem;
> @@ -986,19 +1058,23 @@ static void free_mem_cgroup_per_zone_inf
>  static struct cgroup_subsys_state *
>  mem_cgroup_create(struct cgroup_subsys *ss, struct cgroup *cont)
>  {
> - struct mem_cgroup *mem;
> + struct mem_cgroup *mem, *parent;
>   int node;
>
>   if (unlikely((cont->parent) == NULL)) {
>    mem = &init_mem_cgroup;
>    page_cgroup_cache = KMEM_CACHE(page_cgroup, SLAB_PANIC);
> - } else
> +  parent = NULL;
> + } else {
>    mem = kzalloc(sizeof(struct mem_cgroup), GFP_KERNEL);
> +  parent = mem_cgroup_from_cont(cont->parent);
> + }
>
>   if (mem == NULL)
>    return ERR_PTR(-ENOMEM);
>
> - res_counter_init(&mem->res);
> + res_counter_init(&mem->res, parent ? &parent->res : NULL);
> + mem->last_scanned_child = NULL;
```

I though about it recently. Can we have a cgroup file, which will control whether to attach a res_counter to the parent? This will address the YEMEMOTO's question about the performance.

```
>   memset(&mem->info, 0, sizeof(mem->info));
>
> _
>
```

Subject: Re: [RFC][-mm] Memory controller hierarchy support (v1)
Posted by Paul Menage on Sat, 19 Apr 2008 15:49:02 GMT
View Forum Message <> Reply to Message

On Fri, Apr 18, 2008 at 10:35 PM, Balbir Singh
<balbir@linux.vnet.ibm.com> wrote:
>
> 1. We need to hold cgroup_mutex while walking through the children
> in reclaim. We need to figure out the best way to do so. Should
> cgroups provide a helper function/macro for it?

There's already a function, cgroup_lock(). But it would be nice to avoid such a heavy locking here, particularly since memory allocations can occur with cgroup_mutex held, which could lead to a nasty deadlock if the allocation triggered reclaim.

One of the things that I've been considering was to put the parent/child/sibling hierarchy explicitly in cgroup_subsys_state. This would give subsystems their own copy to refer to, and could use their own internal locking to synchronize with callbacks from cgroups that might change the hierarchy. Cpusets could make use of this too, since it has to traverse hierarchies sometimes.

> 2. Do not allow children to have a limit greater than their parents.
> 3. Allow the user to select if hierarchial support is required

My thoughts on this would be:

1) Never attach a first-level child's counter to its parent. As Yamamoto points out, otherwise we end up with extra global operations whenever any cgroup allocates or frees memory. Limiting the total system memory used by all user processes doesn't seem to be something that people are going to generally want to do, and if they really do want to they can just create a non-root child and move the whole system into that.

The one big advantage that you currently get from having all first-level children be attached to the root is that the reclaim logic automatically scans other groups when it reaches the top-level - but I think that can be provided as a special-case in the reclaim traversal, avoiding the overhead of hitting the root cgroup that we have in this patch.

2) Always attach other children's counters to their parents - if the
user didn't want a hierarchy, they could create a flat grouping rather
than nested groupings.

Paul

_____
Containers mailing list
Containers@lists.linux-foundation.org
https://lists.linux-foundation.org/mailman/listinfo/containers

---

Subject: Re: [RFC][-mm] Memory controller hierarchy support (v1)
Posted by Balbir Singh on Sun, 20 Apr 2008 07:43:57 GMT
View Forum Message <> Reply to Message

Pavel Emelyanov wrote:
>
> Are you going to split this patch? As is it looks rather huge :)
>

Sure

>> TODO's/Open Questions
>>
>> 1. We need to hold cgroup_mutex while walking through the children
>>   in reclaim. We need to figure out the best way to do so. Should
>>   cgroups provide a helper function/macro for it?
>> 2. Do not allow children to have a limit greater than their parents.
>> 3. Allow the user to select if hierarchial support is required
>> 4. Fine tune reclaim from children logic
>>
>
> I though about it recently. Can we have a cgroup file, which will
> control whether to attach a res_counter to the parent? This will
> address the YEMEMOTO's question about the performance.
>

It's one of the TODOS

--
Warm Regards,
Balbir Singh
Linux Technology Center
IBM, ISTL

_____
Containers mailing list
Containers@lists.linux-foundation.org
https://lists.linux-foundation.org/mailman/listinfo/containers

Subject: Re: [RFC][-mm] Memory controller hierarchy support (v1)
Posted by Balbir Singh on Sun, 20 Apr 2008 08:16:37 GMT
View Forum Message <> Reply to Message

Paul Menage wrote:
> On Fri, Apr 18, 2008 at 10:35 PM, Balbir Singh
> <balbir@linux.vnet.ibm.com> wrote:
>> 1. We need to hold cgroup_mutex while walking through the children
>> in reclaim. We need to figure out the best way to do so. Should
>> cgroups provide a helper function/macro for it?
>
> There's already a function, cgroup_lock(). But it would be nice to
> avoid such a heavy locking here, particularly since memory allocations
> can occur with cgroup_mutex held, which could lead to a nasty deadlock
> if the allocation triggered reclaim.
>

Hmm.. probably..

> One of the things that I've been considering was to put the
> parent/child/sibling hierarchy explicitly in cgroup_subsys_state. This
> would give subsystems their own copy to refer to, and could use their
> own internal locking to synchronize with callbacks from cgroups that
> might change the hierarchy. Cpusets could make use of this too, since
> it has to traverse hierarchies sometimes.
>

Very cool! I look forward to that infrastructure. I'll also look at the cpuset
code and see how to traverse the hierarchy.

>> 2. Do not allow children to have a limit greater than their parents.
>> 3. Allow the user to select if hierarchial support is required
>
> My thoughts on this would be:
>
> 1) Never attach a first-level child's counter to its parent. As
> Yamamoto points out, otherwise we end up with extra global operations
> whenever any cgroup allocates or frees memory. Limiting the total
> system memory used by all user processes doesn't seem to be something
> that people are going to generally want to do, and if they really do
> want to they can just create a non-root child and move the whole
> system into that.
>
> The one big advantage that you currently get from having all
> first-level children be attached to the root is that the reclaim logic
> automatically scans other groups when it reaches the top-level - but I
> think that can be provided as a special-case in the reclaim traversal,
> avoiding the overhead of hitting the root cgroup that we have in this
> patch.

>

I've been doing some thinking along these lines, I'll think more about this.

> 2) Always attach other children's counters to their parents - if the
> user didn't want a hierarchy, they could create a flat grouping rather
> than nested groupings.
>

Yes, that's a TODO

> Paul

--
Warm Regards,
Balbir Singh
Linux Technology Center
IBM, ISTL

_____
Containers mailing list
Containers@lists.linux-foundation.org
https://lists.linux-foundation.org/mailman/listinfo/containers

---

Subject: Re: [RFC][-mm] Memory controller hierarchy support (v1)
Posted by KAMEZAWA Hiroyuki on Mon, 21 Apr 2008 00:41:43 GMT
View Forum Message <> Reply to Message

On Sat, 19 Apr 2008 14:04:00 +0530
Balbir Singh <balbir@linux.vnet.ibm.com> wrote:

> YAMAMOTO Takashi wrote:
> >> - spin_lock_irqsave(&counter->lock, flags);
> >> - ret = res_counter_charge_locked(counter, val);
> >> - spin_unlock_irqrestore(&counter->lock, flags);
> >> + *limit_exceeded_at = NULL;
> >> + local_irq_save(flags);
> >> + for (c = counter; c != NULL; c = c->parent) {
> >> + spin_lock(&c->lock);
> >> + ret = res_counter_charge_locked(c, val);
> >> + spin_unlock(&c->lock);
> >> + if (ret < 0) {
> >> +   *limit_exceeded_at = c;
> >> +   goto unroll;
> >> + }
> >> + }
> >> + local_irq_restore(flags);

> >> + return 0;
> >> +
> >> +unroll:
> >> + for (unroll_c = counter; unroll_c != c; unroll_c = unroll_c->parent) {
> >> + spin_lock(&unroll_c->lock);
> >> + res_counter_uncharge_locked(unroll_c, val);
> >> + spin_unlock(&unroll_c->lock);
> >> + }
> >> + local_irq_restore(flags);
> >>   return ret;
> >> }
> >
> > i wonder how much performance impacts this involves.
> >
> > it increases the number of atomic ops per charge/uncharge and
> > makes the common case (success) of every charge/uncharge in a system
> > touch a global (ie. root cgroup's) cachelines.
> >
>
> Yes, it does. I'll run some tests to see what the overhead looks like. The
> multi-hierarchy feature is very useful though and one of the TODOs is to make
> the feature user selectable (possibly at run-time)
>
I think multilevel cgroup is useful but this routines handling of hierarchy
seems never good. An easy idea to aginst this is making a child borrow some
amount of charge from its parent for reducing checks.
If you go this way, please show possibility to reducing overhead in your plan.

BTW, do you have ideas of attributes for children<->parent other than 'limit' ?
For example, 'priority' between childlen.

Thanks,
-Kame

_____
Containers mailing list
Containers@lists.linux-foundation.org
https://lists.linux-foundation.org/mailman/listinfo/containers

---

Subject: Re: [RFC][-mm] Memory controller hierarchy support (v1)
Posted by Paul Jackson on Mon, 21 Apr 2008 06:33:25 GMT
View Forum Message <> Reply to Message

Paul M wrote:
> Cpusets could make use of this too, since
> it has to traverse hierarchies sometimes.

Yeah - I suppose cpusets could use it, though
it's not critical. A fair bit of work already
went into cpusets so that it would not need to
traverse this hierarchy on any critical code path,
or while holding inconvenient locks.

So cpusets shouldn't be the driving motivation
for this, but it will likely be happy to go along
for the ride.

--

I won't rest till it's the best ...
Programmer, Linux Scalability
Paul Jackson <pj@sgi.com> 1.940.382.4214

_____

Containers mailing list
Containers@lists.linux-foundation.org
https://lists.linux-foundation.org/mailman/listinfo/containers