Subject: [PATCH v4 00/14] kmem controller for memcg.
Posted by Glauber Costa on Mon, 08 Oct 2012 10:06:06 GMT
View Forum Message <> Reply to Message

Hi,

This is the first part of the kernel memory controller for memcg. It has been discussed many times, and I consider this stable enough to be on tree. A follow up to this series are the patches to also track slab memory. They are not included here because I believe we could benefit from merging them separately for better testing coverage. If there are any issues preventing this to be merged, let me know. I'll be happy to address them.

```
*v4:	- kmem_accounted can no longer become unlimited
	- kmem_accounted can no longer become limited, if group has children.
	- documentation moved to this patchset
	- more style changes
	- css_get in charge path to ensure task won't move during charge
*v3:
 - Changed function names to match memcg's
 - avoid doing get/put in charge/uncharge path
 - revert back to keeping the account enabled after it is first activated
```

The slab patches are going through a rework of the cache selection mechanism. But after some rounds of discussion, the bulk of it is still stable in my self evaluation and could be merged not too long after this. For the reference, the last discussion about them happened at http://lwn.net/Articles/516705/. I expect to send patches for that within the week.

Numbers can be found at https://lkml.org/lkml/2012/9/13/239

A (throwaway) git tree with them is placed at:

  git://git.kernel.org/pub/scm/linux/kernel/git/glommer/memcg. git kmemcg-stack

A general explanation of what this is all about follows:

The kernel memory limitation mechanism for memcg concerns itself with disallowing potentially non-reclaimable allocations to happen in exaggerate quantities by a particular set of processes (cgroup). Those allocations could create pressure that affects the behavior of a different and unrelated set of processes.

Its basic working mechanism is to annotate some allocations with the _GFP_KMEMCG flag. When this flag is set, the current process allocating will have its memcg identified and charged against. When reaching a specific limit, further allocations will be denied.

One example of such problematic pressure that can be prevented by this work is a fork bomb conducted in a shell. We prevent it by noting that processes use a limited amount of stack pages. Seen this way, a fork bomb is just a special case of resource abuse. If the offender is unable to grab more pages for the stack, no new processes can be created.

There are also other things the general mechanism protects against. For example, using too much of pinned dentry and inode cache, by touching files an leaving them in memory forever.

In fact, a simple:

```
while true; do mkdir x; cd x; done
```

can halt your system easily because the file system limits are hard to reach (big disks), but the kernel memory is not. Those are examples, but the list certainly don't stop here.

An important use case for all that, is concerned with people offering hosting services through containers. In a physical box we can put a limit to some resources, like total number of processes or threads. But in an environment where each independent user gets its own piece of the machine, we don't want a potentially malicious user to destroy good users' services.

This might be true for systemd as well, that now groups services inside cgroups. They generally want to put forward a set of guarantees that limits the running service in a variety of ways, so that if they become badly behaved, they won't interfere with the rest of the system.

There is, of course, a cost for that. To attempt to mitigate that, static branches are used to make sure that even if the feature is compiled in with potentially a lot of memory cgroups deployed this code will only be enabled after the first user of this service configures any limit. Limits lower than the user limit effectively means there is a separate kernel memory limit that may be reached independently than the user limit. Values equal or greater than the user limit implies only that kernel memory is tracked. This provides a unified vision of "maximum memory", be it kernel or user memory. Because this is all default-off, existing deployments will see no change in behavior.

```
Glauber Costa (12):
  memcg: change defines to an enum
  kmem accounting basic infrastructure
  Add a __GFP_KMEMCG flag
  memcg: kmem controller infrastructure
  mm: Allocate kernel pages to the right memcg
  res_counter: return amount of charges after res_counter_uncharge
  memcg: kmem accounting lifecycle management
  memcg: use static branches when code not in use
  memcg: allow a memcg with kmem charges to be destructed.
  execute the whole memcg freeing in free_worker
  protect architectures where THREAD_SIZE >= PAGE_SIZE against fork
    bombs
  Add documentation about the kmem controller

Suleiman Souhlal (2):
  memcg: Make it possible to use the stock for more than one page.
  memcg: Reclaim when more than one page needed.

 Documentation/cgroups/memory.txt          |  55 ++-
 Documentation/cgroups/resource_counter.txt |   7 +-
 include/linux/gfp.h                        |   6 +-
 include/linux/memcontrol.h                 |  97 +++++
 include/linux/res_counter.h                |  12 +-
 include/linux/thread_info.h                |   2 +
 include/trace/events/gfpflags.h            |   1 +
 kernel/fork.c                              |   4 +-
 kernel/res_counter.c                       |  20 +-
 mm/memcontrol.c                            | 555 +++++++++++++++++++++++++----
 mm/page_alloc.c                            |  35 ++
 11 files changed, 713 insertions(+), 81 deletions(-)

-- 
1.7.11.4
```

Subject: [PATCH v4 01/14] memcg: Make it possible to use the stock for more than one page.
Posted by Glauber Costa on Mon, 08 Oct 2012 10:06:07 GMT

View Forum Message <> Reply to Message

From: Suleiman Souhlal <ssouhlal@FreeBSD.org>

We currently have a percpu stock cache scheme that charges one page at a
time from memcg->res, the user counter. When the kernel memory
controller comes into play, we'll need to charge more than that.

This is because kernel memory allocations will also draw from the user
counter, and can be bigger than a single page, as it is the case with
the stack (usually 2 pages) or some higher order slabs.

[glommer@parallels.com: added a changelog]

Signed-off-by: Suleiman Souhlal <suleiman@google.com>
Signed-off-by: Glauber Costa <glommer@parallels.com>

Acked-by: David Rientjes <rientjes@google.com>
Acked-by: Kamezawa Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com>
Acked-by: Michal Hocko <mhocko@suse.cz>
Acked-by: Johannes Weiner <hannes@cmpxchg.org>

```
---
 mm/memcontrol.c | 28 ++++++++++++++++++----------
 1 file changed, 18 insertions(+), 10 deletions(-)

diff --git a/mm/memcontrol.c b/mm/memcontrol.c
index 7acf43b..47cb019 100644
--- a/mm/memcontrol.c
+++ b/mm/memcontrol.c
@@ -2028,20 +2028,28 @@ struct memcg_stock_pcp {
 static DEFINE_PER_CPU(struct memcg_stock_pcp, memcg_stock);
 static DEFINE_MUTEX(percpu_charge_mutex);

-/*
- * Try to consume stocked charge on this cpu. If success, one page is consumed
- * from local stock and true is returned. If the stock is 0 or charges from a
- * cgroup which is not current target, returns false. This stock will be
- * refilled.
+/**
+ * consume_stock: Try to consume stocked charge on this cpu.
+ * @memcg: memcg to consume from.
+ * @nr_pages: how many pages to charge.
+ *
+ * The charges will only happen if @memcg matches the current cpu's memcg
+ * stock, and at least @nr_pages are available in that stock.  Failure to
+ * service an allocation will refill the stock.
+ *
+ * returns true if succesfull, false otherwise.
  */
-static bool consume_stock(struct mem_cgroup *memcg)
+static bool consume_stock(struct mem_cgroup *memcg, int nr_pages)
 {
  struct memcg_stock_pcp *stock;
  bool ret = true;

+ if (nr_pages > CHARGE_BATCH)
+  return false;
+
  stock = &get_cpu_var(memcg_stock);
- if (memcg == stock->cached && stock->nr_pages)
-  stock->nr_pages--;
+ if (memcg == stock->cached && stock->nr_pages >= nr_pages)
+  stock->nr_pages -= nr_pages;
  else /* need to call res_counter_charge */
   ret = false;
  put_cpu_var(memcg_stock);
@@ -2340,7 +2348,7 @@ again:
   VM_BUG_ON(css_is_removed(&memcg->css));
   if (mem_cgroup_is_root(memcg))
    goto done;
-  if (nr_pages == 1 && consume_stock(memcg))
+  if (consume_stock(memcg, nr_pages))
    goto done;
   css_get(&memcg->css);
  } else {
@@ -2365,7 +2373,7 @@ again:
    rcu_read_unlock();
    goto done;
   }
-  if (nr_pages == 1 && consume_stock(memcg)) {
+  if (consume_stock(memcg, nr_pages)) {
    /*
     * It seems dagerous to access memcg without css_get().
     * But considering how consume_stok works, it's not
--
1.7.11.4
```

Subject: [PATCH v4 02/14] memcg: Reclaim when more than one page needed.
Posted by Glauber Costa on Mon, 08 Oct 2012 10:06:08 GMT
View Forum Message <> Reply to Message

From: Suleiman Souhlal <ssouhlal@FreeBSD.org>

mem_cgroup_do_charge() was written before kmem accounting, and expects
three cases: being called for 1 page, being called for a stock of 32
pages, or being called for a hugepage. If we call for 2 or 3 pages (and
both the stack and several slabs used in process creation are such, at
least with the debug options I had), it assumed it's being called for
stock and just retried without reclaiming.

Fix that by passing down a minsize argument in addition to the csize.

And what to do about that (csize == PAGE_SIZE && ret) retry? If it's
needed at all (and presumably is since it's there, perhaps to handle
races), then it should be extended to more than PAGE_SIZE, yet how far?
And should there be a retry count limit, of what? For now retry up to
COSTLY_ORDER (as page_alloc.c does) and make sure not to do it if
__GFP_NORETRY.

[v4: fixed nr pages calculation pointed out by Christoph Lameter]

Signed-off-by: Suleiman Souhlal <suleiman@google.com>

```
Signed-off-by: Glauber Costa <glommer@parallels.com>
Reviewed-by: Kamezawa Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com>
Acked-by: Michal Hocko <mhocko@suse.cz>
Acked-by: Johannes Weiner <hannes@cmpxchg.org>
---
 mm/memcontrol.c | 16 +++++++++-------
 1 file changed, 9 insertions(+), 7 deletions(-)

diff --git a/mm/memcontrol.c b/mm/memcontrol.c
index 47cb019..7a9652a 100644
--- a/mm/memcontrol.c
+++ b/mm/memcontrol.c
@@ -2226,7 +2226,8 @@ enum {
 };

 static int mem_cgroup_do_charge(struct mem_cgroup *memcg, gfp_t gfp_mask,
-    unsigned int nr_pages, bool oom_check)
+    unsigned int nr_pages, unsigned int min_pages,
+    bool oom_check)
 {
  unsigned long csize = nr_pages * PAGE_SIZE;
  struct mem_cgroup *mem_over_limit;
@@ -2249,18 +2250,18 @@ static int mem_cgroup_do_charge(struct mem_cgroup *memcg,
gfp_t gfp_mask,
  } else
   mem_over_limit = mem_cgroup_from_res_counter(fail_res, res);
  /*
-  * nr_pages can be either a huge page (HPAGE_PMD_NR), a batch
-  * of regular pages (CHARGE_BATCH), or a single regular page (1).
-  *
   * Never reclaim on behalf of optional batching, retry with a
   * single page instead.
   */
- if (nr_pages == CHARGE_BATCH)
+ if (nr_pages > min_pages)
   return CHARGE_RETRY;

  if (!(gfp_mask & __GFP_WAIT))
   return CHARGE_WOULDBLOCK;

+ if (gfp_mask & __GFP_NORETRY)
+  return CHARGE_NOMEM;
+
  ret = mem_cgroup_reclaim(mem_over_limit, gfp_mask, flags);
  if (mem_cgroup_margin(mem_over_limit) >= nr_pages)
   return CHARGE_RETRY;
@@ -2273,7 +2274,7 @@ static int mem_cgroup_do_charge(struct mem_cgroup *memcg, gfp_t
gfp_mask,
   * unlikely to succeed so close to the limit, and we fall back
   * to regular pages anyway in case of failure.
   */
- if (nr_pages == 1 && ret)
+ if (nr_pages <= (1 << PAGE_ALLOC_COSTLY_ORDER) && ret)
   return CHARGE_RETRY;

  /*
@@ -2408,7 +2409,8 @@ again:
    nr_oom_retries = MEM_CGROUP_RECLAIM_RETRIES;
   }

-  ret = mem_cgroup_do_charge(memcg, gfp_mask, batch, oom_check);
+  ret = mem_cgroup_do_charge(memcg, gfp_mask, batch, nr_pages,
+      oom_check);
   switch (ret) {
   case CHARGE_OK:
    break;
--
1.7.11.4
```

Subject: [PATCH v4 03/14] memcg: change defines to an enum
Posted by Glauber Costa on Mon, 08 Oct 2012 10:06:09 GMT

View Forum Message <> Reply to Message

This is just a cleanup patch for clarity of expression. In earlier
submissions, people asked it to be in a separate patch, so here it is.

[v2: use named enum as type throughout the file as well]

Signed-off-by: Glauber Costa <glommer@parallels.com>
Acked-by: Kamezawa Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com>
Acked-by: Michal Hocko <mhocko@suse.cz>
Acked-by: Johannes Weiner <hannes@cmpxchg.org>

```
---
 mm/memcontrol.c | 26 ++++++++++++++++----------
 1 file changed, 16 insertions(+), 10 deletions(-)

diff --git a/mm/memcontrol.c b/mm/memcontrol.c
index 7a9652a..71d259e 100644
--- a/mm/memcontrol.c
+++ b/mm/memcontrol.c
@@ -386,9 +386,12 @@ enum charge_type {
 };

 /* for encoding cft->private value on file */
-#define _MEM   (0)
```

```
-#define _MEMSWAP  (1)
-#define _OOM_TYPE  (2)
+enum res_type {
+ _MEM,
+ _MEMSWAP,
+ _OOM_TYPE,
+};
+
 #define MEMFILE_PRIVATE(x, val) ((x) << 16 | (val))
 #define MEMFILE_TYPE(val) ((val) >> 16 & 0xffff)
 #define MEMFILE_ATTR(val) ((val) & 0xffff)
@@ -3915,7 +3918,8 @@ static ssize_t mem_cgroup_read(struct cgroup *cont, struct cftype *cft,
  struct mem_cgroup *memcg = mem_cgroup_from_cont(cont);
  char str[64];
  u64 val;
- int type, name, len;
+ int name, len;
+ enum res_type type;

  type = MEMFILE_TYPE(cft->private);
  name = MEMFILE_ATTR(cft->private);
@@ -3951,7 +3955,8 @@ static int mem_cgroup_write(struct cgroup *cont, struct cftype *cft,
        const char *buffer)
 {
  struct mem_cgroup *memcg = mem_cgroup_from_cont(cont);
- int type, name;
+ enum res_type type;
+ int name;
  unsigned long long val;
  int ret;

@@ -4027,7 +4032,8 @@ out:
 static int mem_cgroup_reset(struct cgroup *cont, unsigned int event)
 {
  struct mem_cgroup *memcg = mem_cgroup_from_cont(cont);
- int type, name;
+ int name;
+ enum res_type type;

  type = MEMFILE_TYPE(event);
  name = MEMFILE_ATTR(event);
@@ -4363,7 +4369,7 @@ static int mem_cgroup_usage_register_event(struct cgroup *cgrp,
  struct mem_cgroup *memcg = mem_cgroup_from_cont(cgrp);
  struct mem_cgroup_thresholds *thresholds;
  struct mem_cgroup_threshold_ary *new;
- int type = MEMFILE_TYPE(cft->private);
+ enum res_type type = MEMFILE_TYPE(cft->private);
  u64 threshold, usage;
  int i, size, ret;

@@ -4446,7 +4452,7 @@ static void mem_cgroup_usage_unregister_event(struct cgroup *cgrp,
  struct mem_cgroup *memcg = mem_cgroup_from_cont(cgrp);
  struct mem_cgroup_thresholds *thresholds;
  struct mem_cgroup_threshold_ary *new;
- int type = MEMFILE_TYPE(cft->private);
+ enum res_type type = MEMFILE_TYPE(cft->private);
  u64 usage;
  int i, j, size;

@@ -4524,7 +4530,7 @@ static int mem_cgroup_oom_register_event(struct cgroup *cgrp,
 {
  struct mem_cgroup *memcg = mem_cgroup_from_cont(cgrp);
  struct mem_cgroup_eventfd_list *event;
- int type = MEMFILE_TYPE(cft->private);
+ enum res_type type = MEMFILE_TYPE(cft->private);

  BUG_ON(type != _OOM_TYPE);
  event = kmalloc(sizeof(*event), GFP_KERNEL);
@@ -4549,7 +4555,7 @@ static void mem_cgroup_oom_unregister_event(struct cgroup *cgrp,
 {
  struct mem_cgroup *memcg = mem_cgroup_from_cont(cgrp);
  struct mem_cgroup_eventfd_list *ev, *tmp;
- int type = MEMFILE_TYPE(cft->private);
+ enum res_type type = MEMFILE_TYPE(cft->private);

  BUG_ON(type != _OOM_TYPE);

--
1.7.11.4
```

Subject: [PATCH v4 04/14] kmem accounting basic infrastructure
Posted by Glauber Costa on Mon, 08 Oct 2012 10:06:10 GMT

View Forum Message <> Reply to Message

This patch adds the basic infrastructure for the accounting of the slab
caches. To control that, the following files are created:

 * memory.kmem.usage_in_bytes
 * memory.kmem.limit_in_bytes
 * memory.kmem.failcnt
 * memory.kmem.max_usage_in_bytes

They have the same meaning of their user memory counterparts. They
reflect the state of the "kmem" res_counter.

Per cgroup slab memory accounting is not enabled until a limit is set
for the group. Once the limit is set the accounting cannot be disabled
for that group. This means that after the patch is applied, no
behavioral changes exists for whoever is still using memcg to control
their memory usage, until memory.kmem.limit_in_bytes is set for the
first time.

We always account to both user and kernel resource_counters. This
effectively means that an independent kernel limit is in place when the
limit is set to a lower value than the user memory. A equal or higher
value means that the user limit will always hit first, meaning that kmem
is effectively unlimited.

People who want to track kernel memory but not limit it, can set this
limit to a very high number (like RESOURCE_MAX - 1page - that no one
will ever hit, or equal to the user memory)

[v4: make kmem files part of the main array;
      do not allow limit to be set for non-empty cgroups]

Signed-off-by: Glauber Costa <glommer@parallels.com>
CC: Michal Hocko <mhocko@suse.cz>
CC: Johannes Weiner <hannes@cmpxchg.org>
Acked-by: Kamezawa Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com>

```
---
 mm/memcontrol.c | 123 +++++++++++++++++++++++++++++++++++++++++++++++++++++++++++-
 1 file changed, 122 insertions(+), 1 deletion(-)

diff --git a/mm/memcontrol.c b/mm/memcontrol.c
index 71d259e..ba855cc 100644
--- a/mm/memcontrol.c
+++ b/mm/memcontrol.c
@@ -266,6 +266,10 @@ struct mem_cgroup {
  };

  /*
+  * the counter to account for kernel memory usage.
+  */
+ struct res_counter kmem;
+ /*
   * Per cgroup active and inactive list, similar to the
   * per zone LRU lists.
   */
@@ -280,6 +284,7 @@ struct mem_cgroup {
   * Should the accounting and control be hierarchical, per subtree?
   */
  bool use_hierarchy;
+ unsigned long kmem_accounted; /* See KMEM_ACCOUNTED_*, below */
```

```
  bool  oom_lock;
  atomic_t under_oom;
@@ -332,6 +337,26 @@ struct mem_cgroup {
 #endif
 };

+/* internal only representation about the status of kmem accounting. */
+enum {
+ KMEM_ACCOUNTED_ACTIVE = 0, /* accounted by this cgroup itself */
+};
+
+/* first bit */
+#define KMEM_ACCOUNTED_MASK 0x1
+
+#ifdef CONFIG_MEMCG_KMEM
+static void memcg_kmem_set_active(struct mem_cgroup *memcg)
+{
+ set_bit(KMEM_ACCOUNTED_ACTIVE, &memcg->kmem_accounted);
+}
+
+static bool memcg_kmem_is_accounted(struct mem_cgroup *memcg)
+{
+ return test_bit(KMEM_ACCOUNTED_ACTIVE, &memcg->kmem_accounted);
+}
+#endif
+
 /* Stuffs for move charges at task migration. */
 /*
  * Types of charges to be moved. "move_charge_at_immitgrate" is treated as a
@@ -390,6 +415,7 @@ enum res_type {
  _MEM,
  _MEMSWAP,
  _OOM_TYPE,
+ _KMEM,
 };

 #define MEMFILE_PRIVATE(x, val) ((x) << 16 | (val))
@@ -1433,6 +1459,10 @@ done:
   res_counter_read_u64(&memcg->memsw, RES_USAGE) >> 10,
   res_counter_read_u64(&memcg->memsw, RES_LIMIT) >> 10,
   res_counter_read_u64(&memcg->memsw, RES_FAILCNT));
+ printk(KERN_INFO "kmem: usage %llukB, limit %llukB, failcnt %llu\n",
+  res_counter_read_u64(&memcg->kmem, RES_USAGE) >> 10,
+  res_counter_read_u64(&memcg->kmem, RES_LIMIT) >> 10,
+  res_counter_read_u64(&memcg->kmem, RES_FAILCNT));
 }
 /*
@@ -3940,6 +3970,9 @@ static ssize_t mem_cgroup_read(struct cgroup *cont, struct cftype *cft,
   else
    val = res_counter_read_u64(&memcg->memsw, name);
   break;
+ case _KMEM:
+  val = res_counter_read_u64(&memcg->kmem, name);
+  break;
  default:
   BUG();
  }
@@ -3947,6 +3980,58 @@ static ssize_t mem_cgroup_read(struct cgroup *cont, struct cftype
*cft,
  len = scnprintf(str, sizeof(str), "%llu\n", (unsigned long long)val);
  return simple_read_from_buffer(buf, nbytes, ppos, str, len);
 }
+
+static int memcg_update_kmem_limit(struct cgroup *cont, u64 val)
+{
+ int ret = -EINVAL;
+#ifdef CONFIG_MEMCG_KMEM
+ struct mem_cgroup *memcg = mem_cgroup_from_cont(cont);
+ /*
+  * For simplicity, we won't allow this to be disabled.  It also can't
+  * be changed if the cgroup has children already, or if tasks had
+  * already joined.
+  *
+  * If tasks join before we set the limit, a person looking at
+  * kmem.usage_in_bytes will have no way to determine when it took
+  * place, which makes the value quite meaningless.
+  *
+  * After it first became limited, changes in the value of the limit are
+  * of course permitted.
+  *
+  * Taking the cgroup_lock is really offensive, but it is so far the only
+  * way to guarantee that no children will appear. There are plenty of
+  * other offenders, and they should all go away. Fine grained locking
+  * is probably the way to go here. When we are fully hierarchical, we
+  * can also get rid of the use_hierarchy check.
+  */
+ cgroup_lock();
+ mutex_lock(&set_limit_mutex);
+ if (!memcg->kmem_accounted && val != RESOURCE_MAX) {
+  if (cgroup_task_count(cont) || (memcg->use_hierarchy &&
+      !list_empty(&cont->children))) {
+   ret = -EBUSY;
+   goto out;
+  }
+  ret = res_counter_set_limit(&memcg->kmem, val);
+  if (ret)
+   goto out;
+
+  memcg_kmem_set_active(memcg);
+ } else
+  ret = res_counter_set_limit(&memcg->kmem, val);
+out:
+ mutex_unlock(&set_limit_mutex);
+ cgroup_unlock();
+#endif
+ return ret;
+}
+
+static void memcg_propagate_kmem(struct mem_cgroup *memcg,
+     struct mem_cgroup *parent)
+{
+ memcg->kmem_accounted = parent->kmem_accounted;
+}
+
 /*
  * The user of this function is...
  * RES_LIMIT.
@@ -3978,8 +4063,12 @@ static int mem_cgroup_write(struct cgroup *cont, struct cftype *cft,
    break;
   if (type == _MEM)
    ret = mem_cgroup_resize_limit(memcg, val);
-  else
+  else if (type == _MEMSWAP)
    ret = mem_cgroup_resize_memsw_limit(memcg, val);
+  else if (type == _KMEM)
+   ret = memcg_update_kmem_limit(cont, val);
+  else
+   return -EINVAL;
   break;
  case RES_SOFT_LIMIT:
   ret = res_counter_memparse_write_strategy(buffer, &val);
@@ -4045,12 +4134,16 @@ static int mem_cgroup_reset(struct cgroup *cont, unsigned int
event)
  case RES_MAX_USAGE:
   if (type == _MEM)
    res_counter_reset_max(&memcg->res);
+  else if (type == _KMEM)
+   res_counter_reset_max(&memcg->kmem);
   else
    res_counter_reset_max(&memcg->memsw);
   break;
  case RES_FAILCNT:
   if (type == _MEM)
    res_counter_reset_failcnt(&memcg->res);
+  else if (type == _KMEM)
+   res_counter_reset_failcnt(&memcg->kmem);
   else
    res_counter_reset_failcnt(&memcg->memsw);
   break;
@@ -4728,6 +4821,31 @@ static struct cftype mem_cgroup_files[] = {
   .read = mem_cgroup_read,
  },
 #endif
+#ifdef CONFIG_MEMCG_KMEM
+ {
+  .name = "kmem.limit_in_bytes",
+  .private = MEMFILE_PRIVATE(_KMEM, RES_LIMIT),
+  .write_string = mem_cgroup_write,
+  .read = mem_cgroup_read,
+ },
+ {
+  .name = "kmem.usage_in_bytes",
+  .private = MEMFILE_PRIVATE(_KMEM, RES_USAGE),
+  .read = mem_cgroup_read,
+ },
+ {
+  .name = "kmem.failcnt",
+  .private = MEMFILE_PRIVATE(_KMEM, RES_FAILCNT),
+  .trigger = mem_cgroup_reset,
+  .read = mem_cgroup_read,
+ },
+ {
+  .name = "kmem.max_usage_in_bytes",
+  .private = MEMFILE_PRIVATE(_KMEM, RES_MAX_USAGE),
+  .trigger = mem_cgroup_reset,
+  .read = mem_cgroup_read,
+ },
+#endif
  { }, /* terminate */
 };

@@ -4973,6 +5091,7 @@ mem_cgroup_create(struct cgroup *cont)
  if (parent && parent->use_hierarchy) {
   res_counter_init(&memcg->res, &parent->res);
   res_counter_init(&memcg->memsw, &parent->memsw);
+  res_counter_init(&memcg->kmem, &parent->kmem);
   /*
    * We increment refcnt of the parent to ensure that we can
    * safely access it on res_counter_charge/uncharge.
@@ -4980,9 +5099,11 @@ mem_cgroup_create(struct cgroup *cont)
```

```
    * mem_cgroup(see mem_cgroup_put).
    */
   mem_cgroup_get(parent);
+  memcg_propagate_kmem(memcg, parent);
  } else {
   res_counter_init(&memcg->res, NULL);
   res_counter_init(&memcg->memsw, NULL);
+  res_counter_init(&memcg->kmem, NULL);
   /*
    * Deeper hierachy with use_hierarchy == false doesn't make
    * much sense so let cgroup subsystem know about this
--
1.7.11.4
```

Subject: [PATCH v4 05/14] Add a __GFP_KMEMCG flag
Posted by Glauber Costa on Mon, 08 Oct 2012 10:06:11 GMT

View Forum Message <> Reply to Message

This flag is used to indicate to the callees that this allocation is a
kernel allocation in process context, and should be accounted to
current's memcg. It takes numerical place of the of the recently removed
__GFP_NO_KSWAPD.

[v4: make flag unconditional, also declare it in trace code]

Signed-off-by: Glauber Costa <glommer@parallels.com>
CC: Christoph Lameter <cl@linux.com>
CC: Pekka Enberg <penberg@cs.helsinki.fi>
CC: Michal Hocko <mhocko@suse.cz>
CC: Suleiman Souhlal <suleiman@google.com>
Acked-by: Johannes Weiner <hannes@cmpxchg.org>
Acked-by: Rik van Riel <riel@redhat.com>
Acked-by: Mel Gorman <mel@csn.ul.ie>
Acked-by: Kamezawa Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com>

```
---
 include/linux/gfp.h             | 3 ++-
 include/trace/events/gfpflags.h | 1 +
 2 files changed, 3 insertions(+), 1 deletion(-)

diff --git a/include/linux/gfp.h b/include/linux/gfp.h
index 02c1c97..9289d46 100644
--- a/include/linux/gfp.h
+++ b/include/linux/gfp.h
@@ -31,6 +31,7 @@ struct vm_area_struct;
 #define ___GFP_THISNODE  0x40000u
 #define ___GFP_RECLAIMABLE 0x80000u
 #define ___GFP_NOTRACK  0x200000u
```

```
+#define ___GFP_KMEMCG  0x400000u
 #define ___GFP_OTHER_NODE 0x800000u
 #define ___GFP_WRITE  0x1000000u

@@ -87,7 +88,7 @@ struct vm_area_struct;

 #define __GFP_OTHER_NODE ((__force gfp_t)___GFP_OTHER_NODE) /* On behalf of other
node */
 #define __GFP_WRITE ((__force gfp_t)___GFP_WRITE) /* Allocator intends to dirty page */
-
+#define __GFP_KMEMCG ((__force gfp_t)___GFP_KMEMCG) /* Allocation comes from a
memcg-accounted resource */
 /*
  * This may seem redundant, but it's a way of annotating false positives vs.
  * allocations that simply cannot be supported (e.g. page tables).
diff --git a/include/trace/events/gfpflags.h b/include/trace/events/gfpflags.h
index 9391706..730df12 100644
--- a/include/trace/events/gfpflags.h
+++ b/include/trace/events/gfpflags.h
@@ -36,6 +36,7 @@
  {(unsigned long)__GFP_RECLAIMABLE, "GFP_RECLAIMABLE"}, \
  {(unsigned long)__GFP_MOVABLE,  "GFP_MOVABLE"},  \
  {(unsigned long)__GFP_NOTRACK,  "GFP_NOTRACK"},  \
+ {(unsigned long)__GFP_KMEMCG,  "GFP_KMEMCG"},  \
  {(unsigned long)__GFP_OTHER_NODE, "GFP_OTHER_NODE"} \
  ) : "GFP_NOWAIT"

--
1.7.11.4
```

Subject: [PATCH v4 06/14] memcg: kmem controller infrastructure
Posted by Glauber Costa on Mon, 08 Oct 2012 10:06:12 GMT
View Forum Message <> Reply to Message

This patch introduces infrastructure for tracking kernel memory pages to
a given memcg. This will happen whenever the caller includes the flag
__GFP_KMEMCG flag, and the task belong to a memcg other than the root.

In memcontrol.h those functions are wrapped in inline acessors. The
idea is to later on, patch those with static branches, so we don't incur
any overhead when no mem cgroups with limited kmem are being used.

Users of this functionality shall interact with the memcg core code
through the following functions:

```
memcg_kmem_newpage_charge: will return true if the group can handle the
                           allocation. At this point, struct page is not
```

```
                         yet allocated.

memcg_kmem_commit_charge: will either revert the charge, if struct page
                         allocation failed, or embed memcg information
                         into page_cgroup.

memcg_kmem_uncharge_page: called at free time, will revert the charge.

[ v2: improved comments and standardized function names ]
[ v3: handle no longer opaque, functions not exported,
  even more comments ]
[ v4: reworked Used bit handling and surroundings for more clarity ]
[ v5: simplified code for kmemcg compiled out and core functions in
  memcontrol.c, moved kmem code to the middle to avoid forward decls ]

Signed-off-by: Glauber Costa <glommer@parallels.com>
CC: Christoph Lameter <cl@linux.com>
CC: Pekka Enberg <penberg@cs.helsinki.fi>
CC: Michal Hocko <mhocko@suse.cz>
CC: Kamezawa Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com>
CC: Johannes Weiner <hannes@cmpxchg.org>
---
 include/linux/memcontrol.h |  95 +++++++++++++++++++++++++
 mm/memcontrol.c            | 173 +++++++++++++++++++++++++++++++++++++++++++++--
 2 files changed, 263 insertions(+), 5 deletions(-)

diff --git a/include/linux/memcontrol.h b/include/linux/memcontrol.h
index 8d9489f..783cd78 100644
--- a/include/linux/memcontrol.h
+++ b/include/linux/memcontrol.h
@@ -21,6 +21,7 @@
 #define _LINUX_MEMCONTROL_H
 #include <linux/cgroup.h>
 #include <linux/vm_event_item.h>
+#include <linux/hardirq.h>

 struct mem_cgroup;
 struct page_cgroup;
@@ -399,6 +400,85 @@ struct sock;
 #ifdef CONFIG_MEMCG_KMEM
 void sock_update_memcg(struct sock *sk);
 void sock_release_memcg(struct sock *sk);
+
+static inline bool memcg_kmem_enabled(void)
+{
+ return true;
+}
+
+extern bool __memcg_kmem_newpage_charge(gfp_t gfp, struct mem_cgroup **memcg,
+     int order);
+extern void __memcg_kmem_commit_charge(struct page *page,
+           struct mem_cgroup *memcg, int order);
+extern void __memcg_kmem_uncharge_page(struct page *page, int order);
+
+/**
+ * memcg_kmem_newpage_charge: verify if a new kmem allocation is allowed.
+ * @gfp: the gfp allocation flags.
+ * @memcg: a pointer to the memcg this was charged against.
+ * @order: allocation order.
+ *
+ * returns true if the memcg where the current task belongs can hold this
+ * allocation.
+ *
+ * We return true automatically if this allocation is not to be accounted to
+ * any memcg.
+ */
+static __always_inline bool
+memcg_kmem_newpage_charge(gfp_t gfp, struct mem_cgroup **memcg, int order)
+{
+ if (!memcg_kmem_enabled())
+  return true;
+
+ /*
+  * __GFP_NOFAIL allocations will move on even if charging is not
+  * possible. Therefore we don't even try, and have this allocation
+  * unaccounted. We could in theory charge it with
+  * res_counter_charge_nofail, but we hope those allocations are rare,
+  * and won't be worth the trouble.
+  */
+ if (!(gfp & __GFP_KMEMCG) || (gfp & __GFP_NOFAIL))
+  return true;
+ if (in_interrupt() || (!current->mm) || (current->flags & PF_KTHREAD))
+  return true;
+ /* mem_cgroup_try_charge will test this too, but better exit quickly */
+ if (test_thread_flag(TIF_MEMDIE))
+  return true;
+ return __memcg_kmem_newpage_charge(gfp, memcg, order);
+}
+
+/**
+ * memcg_kmem_uncharge_page: uncharge pages from memcg
+ * @page: pointer to struct page being freed
+ * @order: allocation order.
+ *
+ * there is no need to specify memcg here, since it is embedded in page_cgroup
+ */
```

```
+static __always_inline void
+memcg_kmem_uncharge_page(struct page *page, int order)
+{
+ if (memcg_kmem_enabled())
+  __memcg_kmem_uncharge_page(page, order);
+}
+
+/**
+ * memcg_kmem_commit_charge: embeds correct memcg in a page
+ * @page: pointer to struct page recently allocated
+ * @memcg: the memcg structure we charged against
+ * @order: allocation order.
+ *
+ * Needs to be called after memcg_kmem_newpage_charge, regardless of success or
+ * failure of the allocation. if @page is NULL, this function will revert the
+ * charges. Otherwise, it will commit the memcg given by @memcg to the
+ * corresponding page_cgroup.
+ */
+static __always_inline void
+memcg_kmem_commit_charge(struct page *page, struct mem_cgroup *memcg, int order)
+{
+ if (memcg_kmem_enabled() && memcg)
+  __memcg_kmem_commit_charge(page, memcg, order);
+}
+
 #else
 static inline void sock_update_memcg(struct sock *sk)
 {
@@ -406,6 +486,21 @@ static inline void sock_update_memcg(struct sock *sk)
 static inline void sock_release_memcg(struct sock *sk)
 {
 }
+
+static inline bool
+memcg_kmem_newpage_charge(gfp_t gfp, struct mem_cgroup **memcg, int order)
+{
+ return true;
+}
+
+static inline void  memcg_kmem_uncharge_page(struct page *page, int order)
+{
+}
+
+static inline void
+memcg_kmem_commit_charge(struct page *page, struct mem_cgroup *memcg, int order)
+{
+}
 #endif /* CONFIG_MEMCG_KMEM */
 #endif /* _LINUX_MEMCONTROL_H */

diff --git a/mm/memcontrol.c b/mm/memcontrol.c
index ba855cc..dda54f0 100644
--- a/mm/memcontrol.c
+++ b/mm/memcontrol.c
@@ -10,6 +10,10 @@
  * Copyright (C) 2009 Nokia Corporation
  * Author: Kirill A. Shutemov
  *
+ * Kernel Memory Controller
+ * Copyright (C) 2012 Parallels Inc. and Google Inc.
+ * Authors: Glauber Costa and Suleiman Souhlal
+ *
  * This program is free software; you can redistribute it and/or modify
  * it under the terms of the GNU General Public License as published by
  * the Free Software Foundation; either version 2 of the License, or
@@ -350,11 +354,6 @@ static void memcg_kmem_set_active(struct mem_cgroup *memcg)
 {
  set_bit(KMEM_ACCOUNTED_ACTIVE, &memcg->kmem_accounted);
 }
-
-static bool memcg_kmem_is_accounted(struct mem_cgroup *memcg)
-{
- return test_bit(KMEM_ACCOUNTED_ACTIVE, &memcg->kmem_accounted);
-}
 #endif

 /* Stuffs for move charges at task migration. */
@@ -2636,6 +2635,170 @@ static void __mem_cgroup_commit_charge(struct mem_cgroup
*memcg,
  memcg_check_events(memcg, page);
 }

+#ifdef CONFIG_MEMCG_KMEM
+static inline bool memcg_can_account_kmem(struct mem_cgroup *memcg)
+{
+ return !mem_cgroup_disabled() && !mem_cgroup_is_root(memcg) &&
+  (memcg->kmem_accounted & KMEM_ACCOUNTED_MASK);
+}
+
+static int memcg_charge_kmem(struct mem_cgroup *memcg, gfp_t gfp, u64 size)
+{
+ struct res_counter *fail_res;
+ struct mem_cgroup *_memcg;
+ int ret = 0;
+ bool may_oom;
+
+ /*
+  * Conditions under which we can wait for the oom_killer.
+  * __GFP_NORETRY should be masked by __mem_cgroup_try_charge,
+  * but there is no harm in being explicit here
+  */
+ may_oom = (gfp & __GFP_WAIT) && !(gfp & __GFP_NORETRY);
+
+ _memcg = memcg;
+ ret = __mem_cgroup_try_charge(NULL, gfp, size >> PAGE_SHIFT,
+          &_memcg, may_oom);
+
+ if (!ret) {
+  ret = res_counter_charge(&memcg->kmem, size, &fail_res);
+  if (ret) {
+   res_counter_uncharge(&memcg->res, size);
+   if (do_swap_account)
+    res_counter_uncharge(&memcg->memsw, size);
+  }
+ } else if (ret == -EINTR)  {
+  /*
+   * __mem_cgroup_try_charge() chosed to bypass to root due to
+   * OOM kill or fatal signal.  Since our only options are to
+   * either fail the allocation or charge it to this cgroup, do
+   * it as a temporary condition. But we can't fail. From a
+   * kmem/slab perspective, the cache has already been selected,
+   * by mem_cgroup_get_kmem_cache(), so it is too late to change
+   * our minds
+   */
+  res_counter_charge_nofail(&memcg->res, size, &fail_res);
+  if (do_swap_account)
+   res_counter_charge_nofail(&memcg->memsw, size,
+        &fail_res);
+  res_counter_charge_nofail(&memcg->kmem, size, &fail_res);
+  ret = 0;
+ }
+
+ return ret;
+}
+
+static void memcg_uncharge_kmem(struct mem_cgroup *memcg, u64 size)
+{
+ res_counter_uncharge(&memcg->kmem, size);
+ res_counter_uncharge(&memcg->res, size);
+ if (do_swap_account)
+  res_counter_uncharge(&memcg->memsw, size);
+}
+
+/*
```

```
+ * We need to verify if the allocation against current->mm->owner's memcg is
+ * possible for the given order. But the page is not allocated yet, so we'll
+ * need a further commit step to do the final arrangements.
+ *
+ * It is possible for the task to switch cgroups in this mean time, so at
+ * commit time, we can't rely on task conversion any longer.  We'll then use
+ * the handle argument to return to the caller which cgroup we should commit
+ * against. We could also return the memcg directly and avoid the pointer
+ * passing, but a boolean return value gives better semantics considering
+ * the compiled-out case as well.
+ *
+ * Returning true means the allocation is possible.
+ */
+bool
+__memcg_kmem_newpage_charge(gfp_t gfp, struct mem_cgroup **_memcg, int order)
+{
+ struct mem_cgroup *memcg;
+ int ret;
+
+ *_memcg = NULL;
+ memcg = try_get_mem_cgroup_from_mm(current->mm);
+
+ /*
+  * very rare case described in mem_cgroup_from_task. Unfortunately there
+  * isn't much we can do without complicating this too much, and it would
+  * be gfp-dependent anyway. Just let it go
+  */
+ if (unlikely(!memcg))
+  return true;
+
+ if (!memcg_can_account_kmem(memcg)) {
+  css_put(&memcg->css);
+  return true;
+ }
+
+ mem_cgroup_get(memcg);
+
+ ret = memcg_charge_kmem(memcg, gfp, PAGE_SIZE << order);
+ if (!ret)
+  *_memcg = memcg;
+ else
+  mem_cgroup_put(memcg);
+
+ css_put(&memcg->css);
+ return (ret == 0);
+}
+
+void __memcg_kmem_commit_charge(struct page *page, struct mem_cgroup *memcg,
+        int order)
+{
+ struct page_cgroup *pc;
+
+ VM_BUG_ON(mem_cgroup_is_root(memcg));
+
+ /* The page allocation failed. Revert */
+ if (!page) {
+  memcg_uncharge_kmem(memcg, PAGE_SIZE << order);
+  mem_cgroup_put(memcg);
+  return;
+ }
+
+ pc = lookup_page_cgroup(page);
+ lock_page_cgroup(pc);
+ pc->mem_cgroup = memcg;
+ SetPageCgroupUsed(pc);
+ unlock_page_cgroup(pc);
+}
+
+void __memcg_kmem_uncharge_page(struct page *page, int order)
+{
+ struct mem_cgroup *memcg = NULL;
+ struct page_cgroup *pc;
+
+
+ pc = lookup_page_cgroup(page);
+ /*
+  * Fast unlocked return. Theoretically might have changed, have to
+  * check again after locking.
+  */
+ if (!PageCgroupUsed(pc))
+  return;
+
+ lock_page_cgroup(pc);
+ if (PageCgroupUsed(pc)) {
+  memcg = pc->mem_cgroup;
+  ClearPageCgroupUsed(pc);
+ }
+ unlock_page_cgroup(pc);
+
+ /*
+  * We trust that only if there is a memcg associated with the page, it
+  * is a valid allocation
+  */
+ if (!memcg)
+  return;
+
+ VM_BUG_ON(mem_cgroup_is_root(memcg));
+ memcg_uncharge_kmem(memcg, PAGE_SIZE << order);
+ mem_cgroup_put(memcg);
+}
+#endif /* CONFIG_MEMCG_KMEM */
+
 #ifdef CONFIG_TRANSPARENT_HUGEPAGE

 #define PCGF_NOCOPY_AT_SPLIT (1 << PCG_LOCK | 1 << PCG_MIGRATION)
--
1.7.11.4
```

Subject: [PATCH v4 07/14] mm: Allocate kernel pages to the right memcg
Posted by Glauber Costa on Mon, 08 Oct 2012 10:06:13 GMT

View Forum Message <> Reply to Message

When a process tries to allocate a page with the __GFP_KMEMCG flag, the
page allocator will call the corresponding memcg functions to validate
the allocation. Tasks in the root memcg can always proceed.

To avoid adding markers to the page - and a kmem flag that would
necessarily follow, as much as doing page_cgroup lookups for no reason,
whoever is marking its allocations with __GFP_KMEMCG flag is responsible
for telling the page allocator that this is such an allocation at
free_pages() time. This is done by the invocation of
__free_accounted_pages() and free_accounted_pages().

[v2: inverted test order to avoid a memcg_get leak,
  free_accounted_pages simplification]
[v4: test for TIF_MEMDIE at newpage_charge]

Signed-off-by: Glauber Costa <glommer@parallels.com>
Acked-by: Michal Hocko <mhocko@suse.cz>
Acked-by: Mel Gorman <mgorman@suse.de>
Acked-by: Kamezawa Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com>
CC: Christoph Lameter <cl@linux.com>
CC: Pekka Enberg <penberg@cs.helsinki.fi>
CC: Johannes Weiner <hannes@cmpxchg.org>
CC: Suleiman Souhlal <suleiman@google.com>

```
---
 include/linux/gfp.h |  3 +++
 mm/page_alloc.c     | 35 +++++++++++++++++++++++++++++++++++
 2 files changed, 38 insertions(+)

diff --git a/include/linux/gfp.h b/include/linux/gfp.h
index 9289d46..8f6fe34 100644
--- a/include/linux/gfp.h
```

```
+++ b/include/linux/gfp.h
@@ -362,6 +362,9 @@ extern void free_pages(unsigned long addr, unsigned int order);
 extern void free_hot_cold_page(struct page *page, int cold);
 extern void free_hot_cold_page_list(struct list_head *list, int cold);

+extern void __free_accounted_pages(struct page *page, unsigned int order);
+extern void free_accounted_pages(unsigned long addr, unsigned int order);
+
 #define __free_page(page) __free_pages((page), 0)
 #define free_page(addr) free_pages((addr), 0)

diff --git a/mm/page_alloc.c b/mm/page_alloc.c
index feddc7f..dcf33ad 100644
--- a/mm/page_alloc.c
+++ b/mm/page_alloc.c
@@ -2595,6 +2595,7 @@ __alloc_pages_nodemask(gfp_t gfp_mask, unsigned int order,
  int migratetype = allocflags_to_migratetype(gfp_mask);
  unsigned int cpuset_mems_cookie;
  int alloc_flags = ALLOC_WMARK_LOW|ALLOC_CPUSET;
+ struct mem_cgroup *memcg = NULL;

  gfp_mask &= gfp_allowed_mask;

@@ -2613,6 +2614,13 @@ __alloc_pages_nodemask(gfp_t gfp_mask, unsigned int order,
  if (unlikely(!zonelist->_zonerefs->zone))
   return NULL;

+ /*
+  * Will only have any effect when __GFP_KMEMCG is set.  This is
+  * verified in the (always inline) callee
+  */
+ if (!memcg_kmem_newpage_charge(gfp_mask, &memcg, order))
+  return NULL;
+
 retry_cpuset:
  cpuset_mems_cookie = get_mems_allowed();

@@ -2648,6 +2656,8 @@ out:
  if (unlikely(!put_mems_allowed(cpuset_mems_cookie) && !page))
   goto retry_cpuset;

+ memcg_kmem_commit_charge(page, memcg, order);
+
  return page;
 }
 EXPORT_SYMBOL(__alloc_pages_nodemask);
@@ -2700,6 +2710,31 @@ void free_pages(unsigned long addr, unsigned int order)
```

```
 EXPORT_SYMBOL(free_pages);

+/*
+ * __free_accounted_pages and free_accounted_pages will free pages allocated
+ * with __GFP_KMEMCG.
+ *
+ * Those pages are accounted to a particular memcg, embedded in the
+ * corresponding page_cgroup. To avoid adding a hit in the allocator to search
+ * for that information only to find out that it is NULL for users who have no
+ * interest in that whatsoever, we provide these functions.
+ *
+ * The caller knows better which flags it relies on.
+ */
+void __free_accounted_pages(struct page *page, unsigned int order)
+{
+ memcg_kmem_uncharge_page(page, order);
+ __free_pages(page, order);
+}
+
+void free_accounted_pages(unsigned long addr, unsigned int order)
+{
+ if (addr != 0) {
+  VM_BUG_ON(!virt_addr_valid((void *)addr));
+  __free_accounted_pages(virt_to_page((void *)addr), order);
+ }
+}
+
 static void *make_alloc_exact(unsigned long addr, unsigned order, size_t size)
 {
  if (addr) {
--
1.7.11.4
```

Subject: [PATCH v4 08/14] res_counter: return amount of charges after res_counter_uncharge
Posted by Glauber Costa on Mon, 08 Oct 2012 10:06:14 GMT
View Forum Message <> Reply to Message

It is useful to know how many charges are still left after a call to
res_counter_uncharge. While it is possible to issue a res_counter_read
after uncharge, this can be racy.

If we need, for instance, to take some action when the counters drop
down to 0, only one of the callers should see it. This is the same
semantics as the atomic variables in the kernel.

Since the current return value is void, we don't need to worry about

anything breaking due to this change: nobody relied on that, and only
users appearing from now on will be checking this value.

Signed-off-by: Glauber Costa <glommer@parallels.com>
CC: Michal Hocko <mhocko@suse.cz>
CC: Johannes Weiner <hannes@cmpxchg.org>
CC: Suleiman Souhlal <suleiman@google.com>
CC: Kamezawa Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com>

```
---
 Documentation/cgroups/resource_counter.txt |  7 ++++---
 include/linux/res_counter.h                | 12 +++++++-----
 kernel/res_counter.c                       | 20 +++++++++++++-------
 3 files changed, 24 insertions(+), 15 deletions(-)

diff --git a/Documentation/cgroups/resource_counter.txt
b/Documentation/cgroups/resource_counter.txt
index 0c4a344..c4d99ed 100644
--- a/Documentation/cgroups/resource_counter.txt
+++ b/Documentation/cgroups/resource_counter.txt
@@ -83,16 +83,17 @@ to work with it.
  res_counter->lock internally (it must be called with res_counter->lock
  held). The force parameter indicates whether we can bypass the limit.

- e. void res_counter_uncharge[_locked]
+ e. u64 res_counter_uncharge[_locked]
    (struct res_counter *rc, unsigned long val)

  When a resource is released (freed) it should be de-accounted
  from the resource counter it was accounted to.  This is called
- "uncharging".
+ "uncharging". The return value of this function indicate the amount
+ of charges still present in the counter.

  The _locked routines imply that the res_counter->lock is taken.

- f. void res_counter_uncharge_until
+ f. u64 res_counter_uncharge_until
   (struct res_counter *rc, struct res_counter *top,
    unsinged long val)

diff --git a/include/linux/res_counter.h b/include/linux/res_counter.h
index 7d7fbe2..4b173b6 100644
--- a/include/linux/res_counter.h
+++ b/include/linux/res_counter.h
@@ -130,14 +130,16 @@ int res_counter_charge_nofail(struct res_counter *counter,
  *
  * these calls check for usage underflow and show a warning on the console
  * _locked call expects the counter->lock to be taken
```

```
+ *
+ * returns the total charges still present in @counter.
  */

-void res_counter_uncharge_locked(struct res_counter *counter, unsigned long val);
-void res_counter_uncharge(struct res_counter *counter, unsigned long val);
+u64 res_counter_uncharge_locked(struct res_counter *counter, unsigned long val);
+u64 res_counter_uncharge(struct res_counter *counter, unsigned long val);

-void res_counter_uncharge_until(struct res_counter *counter,
-    struct res_counter *top,
-    unsigned long val);
+u64 res_counter_uncharge_until(struct res_counter *counter,
+          struct res_counter *top,
+          unsigned long val);
 /**
  * res_counter_margin - calculate chargeable space of a counter
  * @cnt: the counter
diff --git a/kernel/res_counter.c b/kernel/res_counter.c
index ad581aa..7b3d6dc 100644
--- a/kernel/res_counter.c
+++ b/kernel/res_counter.c
@@ -86,33 +86,39 @@ int res_counter_charge_nofail(struct res_counter *counter, unsigned long
val,
  return __res_counter_charge(counter, val, limit_fail_at, true);
 }

-void res_counter_uncharge_locked(struct res_counter *counter, unsigned long val)
+u64 res_counter_uncharge_locked(struct res_counter *counter, unsigned long val)
 {
  if (WARN_ON(counter->usage < val))
   val = counter->usage;

  counter->usage -= val;
+ return counter->usage;
 }

-void res_counter_uncharge_until(struct res_counter *counter,
-    struct res_counter *top,
-    unsigned long val)
+u64 res_counter_uncharge_until(struct res_counter *counter,
+          struct res_counter *top,
+          unsigned long val)
 {
  unsigned long flags;
  struct res_counter *c;
+ u64 ret = 0;
```

```
  local_irq_save(flags);
  for (c = counter; c != top; c = c->parent) {
+  u64 r;
   spin_lock(&c->lock);
-  res_counter_uncharge_locked(c, val);
+  r = res_counter_uncharge_locked(c, val);
+  if (c == counter)
+   ret = r;
   spin_unlock(&c->lock);
  }
  local_irq_restore(flags);
+ return ret;
 }

-void res_counter_uncharge(struct res_counter *counter, unsigned long val)
+u64 res_counter_uncharge(struct res_counter *counter, unsigned long val)
 {
- res_counter_uncharge_until(counter, NULL, val);
+ return res_counter_uncharge_until(counter, NULL, val);
 }

 static inline unsigned long long *
--
1.7.11.4
```

Subject: [PATCH v4 09/14] memcg: kmem accounting lifecycle management
Posted by Glauber Costa on Mon, 08 Oct 2012 10:06:15 GMT

View Forum Message <> Reply to Message

Because kmem charges can outlive the cgroup, we need to make sure that
we won't free the memcg structure while charges are still in flight.
For reviewing simplicity, the charge functions will issue
mem_cgroup_get() at every charge, and mem_cgroup_put() at every
uncharge.

This can get expensive, however, and we can do better. mem_cgroup_get()
only really needs to be issued once: when the first limit is set. In the
same spirit, we only need to issue mem_cgroup_put() when the last charge
is gone.

We'll need an extra bit in kmem_accounted for that: KMEM_ACCOUNTED_DEAD.
it will be set when the cgroup dies, if there are charges in the group.
If there aren't, we can proceed right away.

Our uncharge function will have to test that bit every time the charges
drop to 0. Because that is not the likely output of
res_counter_uncharge, this should not impose a big hit on us: it is

certainly much better than a reference count decrease at every
operation.

[v3: merged all lifecycle related patches in one]

Signed-off-by: Glauber Costa <glommer@parallels.com>
CC: Kamezawa Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com>
CC: Christoph Lameter <cl@linux.com>
CC: Pekka Enberg <penberg@cs.helsinki.fi>
CC: Michal Hocko <mhocko@suse.cz>
CC: Johannes Weiner <hannes@cmpxchg.org>
CC: Suleiman Souhlal <suleiman@google.com>

```
---
 mm/memcontrol.c | 56 +++++++++++++++++++++++++++++++++++++++++++++++++-------
 1 file changed, 49 insertions(+), 7 deletions(-)

diff --git a/mm/memcontrol.c b/mm/memcontrol.c
index dda54f0..634c7b5 100644
--- a/mm/memcontrol.c
+++ b/mm/memcontrol.c
@@ -344,6 +344,7 @@ struct mem_cgroup {
 /* internal only representation about the status of kmem accounting. */
 enum {
  KMEM_ACCOUNTED_ACTIVE = 0, /* accounted by this cgroup itself */
+ KMEM_ACCOUNTED_DEAD, /* dead memcg, pending kmem charges */
 };

 /* first bit */
@@ -354,6 +355,22 @@ static void memcg_kmem_set_active(struct mem_cgroup *memcg)
 {
  set_bit(KMEM_ACCOUNTED_ACTIVE, &memcg->kmem_accounted);
 }
+
+static bool memcg_kmem_is_active(struct mem_cgroup *memcg)
+{
+ return test_bit(KMEM_ACCOUNTED_ACTIVE, &memcg->kmem_accounted);
+}
+
+static void memcg_kmem_mark_dead(struct mem_cgroup *memcg)
+{
+ if (test_bit(KMEM_ACCOUNTED_ACTIVE, &memcg->kmem_accounted))
+  set_bit(KMEM_ACCOUNTED_DEAD, &memcg->kmem_accounted);
+}
+
+static bool memcg_kmem_dead(struct mem_cgroup *memcg)
+{
+ return test_and_clear_bit(KMEM_ACCOUNTED_DEAD, &memcg->kmem_accounted);
+}
```

```
 #endif

 /* Stuffs for move charges at task migration. */
@@ -2690,10 +2707,16 @@ static int memcg_charge_kmem(struct mem_cgroup *memcg, gfp_t
gfp, u64 size)

 static void memcg_uncharge_kmem(struct mem_cgroup *memcg, u64 size)
 {
- res_counter_uncharge(&memcg->kmem, size);
  res_counter_uncharge(&memcg->res, size);
  if (do_swap_account)
   res_counter_uncharge(&memcg->memsw, size);
+
+ /* Not down to 0 */
+ if (res_counter_uncharge(&memcg->kmem, size))
+  return;
+
+ if (memcg_kmem_dead(memcg))
+  mem_cgroup_put(memcg);
 }

 /*
@@ -2732,13 +2755,9 @@ __memcg_kmem_newpage_charge(gfp_t gfp, struct mem_cgroup
**_memcg, int order)
   return true;
  }

- mem_cgroup_get(memcg);
-
  ret = memcg_charge_kmem(memcg, gfp, PAGE_SIZE << order);
  if (!ret)
   *_memcg = memcg;
- else
-  mem_cgroup_put(memcg);

  css_put(&memcg->css);
  return (ret == 0);
@@ -2754,7 +2773,6 @@ void __memcg_kmem_commit_charge(struct page *page, struct
mem_cgroup *memcg,
  /* The page allocation failed. Revert */
  if (!page) {
   memcg_uncharge_kmem(memcg, PAGE_SIZE << order);
-  mem_cgroup_put(memcg);
   return;
  }

@@ -2795,7 +2813,6 @@ void __memcg_kmem_uncharge_page(struct page *page, int order)
```

```
  VM_BUG_ON(mem_cgroup_is_root(memcg));
  memcg_uncharge_kmem(memcg, PAGE_SIZE << order);
- mem_cgroup_put(memcg);
 }
 #endif /* CONFIG_MEMCG_KMEM */

@@ -4180,6 +4197,13 @@ static int memcg_update_kmem_limit(struct cgroup *cont, u64 val)
    goto out;

   memcg_kmem_set_active(memcg);
+  /*
+   * kmem charges can outlive the cgroup. In the case of slab
+   * pages, for instance, a page contain objects from various
+   * processes, so it is unfeasible to migrate them away. We
+   * need to reference count the memcg because of that.
+   */
+  mem_cgroup_get(memcg);
  } else
   ret = res_counter_set_limit(&memcg->kmem, val);
 out:
@@ -4193,6 +4217,10 @@ static void memcg_propagate_kmem(struct mem_cgroup *memcg,
      struct mem_cgroup *parent)
 {
  memcg->kmem_accounted = parent->kmem_accounted;
+#ifdef CONFIG_MEMCG_KMEM
+ if (memcg_kmem_is_active(memcg))
+  mem_cgroup_get(memcg);
+#endif
 }

 /*
@@ -4876,6 +4904,20 @@ static int memcg_init_kmem(struct mem_cgroup *memcg, struct
cgroup_subsys *ss)
 static void kmem_cgroup_destroy(struct mem_cgroup *memcg)
 {
  mem_cgroup_sockets_destroy(memcg);
+
+ memcg_kmem_mark_dead(memcg);
+
+ if (res_counter_read_u64(&memcg->kmem, RES_USAGE) != 0)
+  return;
+
+ /*
+  * Charges already down to 0, undo mem_cgroup_get() done in the charge
+  * path here, being careful not to race with memcg_uncharge_kmem: it is
+  * possible that the charges went down to 0 between mark_dead and the
+  * res_counter read, so in that case, we don't need the put
+  */
+ if (memcg_kmem_dead(memcg))
+  mem_cgroup_put(memcg);
 }
 #else
 static int memcg_init_kmem(struct mem_cgroup *memcg, struct cgroup_subsys *ss)
--
1.7.11.4
```

Subject: [PATCH v4 10/14] memcg: use static branches when code not in use
Posted by Glauber Costa on Mon, 08 Oct 2012 10:06:16 GMT

View Forum Message <> Reply to Message

We can use static branches to patch the code in or out when not used.

Because the _ACTIVE bit on kmem_accounted is only set after the
increment is done, we guarantee that the root memcg will always be
selected for kmem charges until all call sites are patched (see
memcg_kmem_enabled). This guarantees that no mischarges are applied.

static branch decrement happens when the last reference count from the
kmem accounting in memcg dies. This will only happen when the charges
drop down to 0.

When that happen, we need to disable the static branch only on those
memcgs that enabled it. To achieve this, we would be forced to
complicate the code by keeping track of which memcgs were the ones
that actually enabled limits, and which ones got it from its parents.

It is a lot simpler just to do static_key_slow_inc() on every child
that is accounted.

[v4: adapted this patch to the changes in kmem_accounted]

Signed-off-by: Glauber Costa <glommer@parallels.com>
CC: Kamezawa Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com>
CC: Christoph Lameter <cl@linux.com>
CC: Pekka Enberg <penberg@cs.helsinki.fi>
CC: Michal Hocko <mhocko@suse.cz>
CC: Johannes Weiner <hannes@cmpxchg.org>
CC: Suleiman Souhlal <suleiman@google.com>

```
---
 include/linux/memcontrol.h |  4 ++-
 mm/memcontrol.c            | 82 +++++++++++++++++++++++++++++++++++++++++++---
 2 files changed, 80 insertions(+), 6 deletions(-)

diff --git a/include/linux/memcontrol.h b/include/linux/memcontrol.h
index 783cd78..40d658d 100644
```

```
--- a/include/linux/memcontrol.h
+++ b/include/linux/memcontrol.h
@@ -22,6 +22,7 @@
 #include <linux/cgroup.h>
 #include <linux/vm_event_item.h>
 #include <linux/hardirq.h>
+#include <linux/jump_label.h>

 struct mem_cgroup;
 struct page_cgroup;
@@ -401,9 +402,10 @@ struct sock;
 void sock_update_memcg(struct sock *sk);
 void sock_release_memcg(struct sock *sk);

+extern struct static_key memcg_kmem_enabled_key;
 static inline bool memcg_kmem_enabled(void)
 {
- return true;
+ return static_key_false(&memcg_kmem_enabled_key);
 }

 extern bool __memcg_kmem_newpage_charge(gfp_t gfp, struct mem_cgroup **memcg,
diff --git a/mm/memcontrol.c b/mm/memcontrol.c
index 634c7b5..724a08b 100644
--- a/mm/memcontrol.c
+++ b/mm/memcontrol.c
@@ -344,11 +344,15 @@ struct mem_cgroup {
 /* internal only representation about the status of kmem accounting. */
 enum {
  KMEM_ACCOUNTED_ACTIVE = 0, /* accounted by this cgroup itself */
+ KMEM_ACCOUNTED_ACTIVATED, /* static key enabled. */
  KMEM_ACCOUNTED_DEAD, /* dead memcg, pending kmem charges */
 };

-/* first bit */
-#define KMEM_ACCOUNTED_MASK 0x1
+/*
+ * first two bits. We account when limit is on, but only after
+ * call sites are patched
+ */
+#define KMEM_ACCOUNTED_MASK 0x3

 #ifdef CONFIG_MEMCG_KMEM
 static void memcg_kmem_set_active(struct mem_cgroup *memcg)
@@ -361,6 +365,11 @@ static bool memcg_kmem_is_active(struct mem_cgroup *memcg)
  return test_bit(KMEM_ACCOUNTED_ACTIVE, &memcg->kmem_accounted);
 }
```

```
+static void memcg_kmem_set_activated(struct mem_cgroup *memcg)
+{
+ set_bit(KMEM_ACCOUNTED_ACTIVATED, &memcg->kmem_accounted);
+}
+
 static void memcg_kmem_mark_dead(struct mem_cgroup *memcg)
 {
  if (test_bit(KMEM_ACCOUNTED_ACTIVE, &memcg->kmem_accounted))
@@ -530,6 +539,26 @@ static void disarm_sock_keys(struct mem_cgroup *memcg)
 }
 #endif

+#ifdef CONFIG_MEMCG_KMEM
+struct static_key memcg_kmem_enabled_key;
+
+static void disarm_kmem_keys(struct mem_cgroup *memcg)
+{
+ if (memcg_kmem_is_active(memcg))
+  static_key_slow_dec(&memcg_kmem_enabled_key);
+}
+#else
+static void disarm_kmem_keys(struct mem_cgroup *memcg)
+{
+}
+#endif /* CONFIG_MEMCG_KMEM */
+
+static void disarm_static_keys(struct mem_cgroup *memcg)
+{
+ disarm_sock_keys(memcg);
+ disarm_kmem_keys(memcg);
+}
+
 static void drain_all_stock_async(struct mem_cgroup *memcg);

 static struct mem_cgroup_per_zone *
@@ -4165,6 +4194,8 @@ static int memcg_update_kmem_limit(struct cgroup *cont, u64 val)
 {
  int ret = -EINVAL;
 #ifdef CONFIG_MEMCG_KMEM
+ bool must_inc_static_branch = false;
+
  struct mem_cgroup *memcg = mem_cgroup_from_cont(cont);
  /*
   * For simplicity, we won't allow this to be disabled.  It also can't
@@ -4196,7 +4227,15 @@ static int memcg_update_kmem_limit(struct cgroup *cont, u64 val)
   if (ret)
    goto out;
```

```
-  memcg_kmem_set_active(memcg);
+  /*
+   * After this point, kmem_accounted (that we test atomically in
+   * the beginning of this conditional), is no longer 0. This
+   * guarantees only one process will set the following boolean
+   * to true. We don't need test_and_set because we're protected
+   * by the set_limit_mutex anyway.
+   */
+  memcg_kmem_set_activated(memcg);
+  must_inc_static_branch = true;
   /*
    * kmem charges can outlive the cgroup. In the case of slab
    * pages, for instance, a page contain objects from various
@@ -4209,6 +4248,27 @@ static int memcg_update_kmem_limit(struct cgroup *cont, u64 val)
 out:
  mutex_unlock(&set_limit_mutex);
  cgroup_unlock();
+
+ /*
+  * We are by now familiar with the fact that we can't inc the static
+  * branch inside cgroup_lock. See disarm functions for details. A
+  * worker here is overkill, but also wrong: After the limit is set, we
+  * must start accounting right away. Since this operation can't fail,
+  * we can safely defer it to here - no rollback will be needed.
+  *
+  * The boolean used to control this is also safe, because
+  * KMEM_ACCOUNTED_ACTIVATED guarantees that only one process will be
+  * able to set it to true;
+  */
+ if (must_inc_static_branch) {
+  static_key_slow_inc(&memcg_kmem_enabled_key);
+  /*
+   * setting the active bit after the inc will guarantee no one
+   * starts accounting before all call sites are patched
+   */
+  memcg_kmem_set_active(memcg);
+ }
+
 #endif
  return ret;
 }
@@ -4218,8 +4278,20 @@ static void memcg_propagate_kmem(struct mem_cgroup *memcg,
 {
  memcg->kmem_accounted = parent->kmem_accounted;
 #ifdef CONFIG_MEMCG_KMEM
- if (memcg_kmem_is_active(memcg))
+ /*
+  * When that happen, we need to disable the static branch only on those
+  * memcgs that enabled it. To achieve this, we would be forced to
+  * complicate the code by keeping track of which memcgs were the ones
+  * that actually enabled limits, and which ones got it from its
+  * parents.
+  *
+  * It is a lot simpler just to do static_key_slow_inc() on every child
+  * that is accounted.
+  */
+ if (memcg_kmem_is_active(memcg)) {
   mem_cgroup_get(memcg);
+  static_key_slow_inc(&memcg_kmem_enabled_key);
+ }
 #endif
 }

@@ -5139,7 +5211,7 @@ static void free_work(struct work_struct *work)
   * to move this code around, and make sure it is outside
   * the cgroup_lock.
   */
- disarm_sock_keys(memcg);
+ disarm_static_keys(memcg);
  if (size < PAGE_SIZE)
   kfree(memcg);
  else
--
1.7.11.4
```

Subject: [PATCH v4 11/14] memcg: allow a memcg with kmem charges to be destructed.
Posted by Glauber Costa on Mon, 08 Oct 2012 10:06:17 GMT
View Forum Message <> Reply to Message

Because the ultimate goal of the kmem tracking in memcg is to track slab
pages as well, we can't guarantee that we'll always be able to point a
page to a particular process, and migrate the charges along with it -
since in the common case, a page will contain data belonging to multiple
processes.

Because of that, when we destroy a memcg, we only make sure the
destruction will succeed by discounting the kmem charges from the user
charges when we try to empty the cgroup.

Signed-off-by: Glauber Costa <glommer@parallels.com>
Acked-by: Kamezawa Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com>
Reviewed-by: Michal Hocko <mhocko@suse.cz>
CC: Christoph Lameter <cl@linux.com>
CC: Pekka Enberg <penberg@cs.helsinki.fi>

```
CC: Johannes Weiner <hannes@cmpxchg.org>
CC: Suleiman Souhlal <suleiman@google.com>
---
 mm/memcontrol.c | 17 ++++++++++++++++-
 1 file changed, 16 insertions(+), 1 deletion(-)

diff --git a/mm/memcontrol.c b/mm/memcontrol.c
index 724a08b..2f92f89 100644
--- a/mm/memcontrol.c
+++ b/mm/memcontrol.c
@@ -546,6 +546,11 @@ static void disarm_kmem_keys(struct mem_cgroup *memcg)
 {
  if (memcg_kmem_is_active(memcg))
   static_key_slow_dec(&memcg_kmem_enabled_key);
+ /*
+  * This check can't live in kmem destruction function,
+  * since the charges will outlive the cgroup
+  */
+ WARN_ON(res_counter_read_u64(&memcg->kmem, RES_USAGE) != 0);
 }
 #else
 static void disarm_kmem_keys(struct mem_cgroup *memcg)
@@ -3994,6 +3999,7 @@ static int mem_cgroup_force_empty(struct mem_cgroup *memcg, bool free_all)
  int node, zid, shrink;
  int nr_retries = MEM_CGROUP_RECLAIM_RETRIES;
  struct cgroup *cgrp = memcg->css.cgroup;
+ u64 usage;

  css_get(&memcg->css);

@@ -4027,8 +4033,17 @@ move_account:
   mem_cgroup_end_move(memcg);
   memcg_oom_recover(memcg);
   cond_resched();
+  /*
+   * Kernel memory may not necessarily be trackable to a specific
+   * process. So they are not migrated, and therefore we can't
+   * expect their value to drop to 0 here.
+   *
+   * having res filled up with kmem only is enough
+   */
+  usage = res_counter_read_u64(&memcg->res, RES_USAGE) -
+   res_counter_read_u64(&memcg->kmem, RES_USAGE);
  /* "ret" should also be checked to ensure all lists are empty. */
- } while (res_counter_read_u64(&memcg->res, RES_USAGE) > 0 || ret);
+ } while (usage > 0 || ret);
 out:
  css_put(&memcg->css);
  return ret;
--
1.7.11.4
```

Subject: [PATCH v4 12/14] execute the whole memcg freeing in free_worker
Posted by Glauber Costa on Mon, 08 Oct 2012 10:06:18 GMT
View Forum Message <> Reply to Message

A lot of the initialization we do in mem_cgroup_create() is done with
softirqs enabled. This include grabbing a css id, which holds
&ss->id_lock->rlock, and the per-zone trees, which holds
rtpz->lock->rlock. All of those signal to the lockdep mechanism that
those locks can be used in SOFTIRQ-ON-W context. This means that the
freeing of memcg structure must happen in a compatible context,
otherwise we'll get a deadlock, like the one bellow, caught by lockdep:

```
  [<ffffffff81103095>] free_accounted_pages+0x47/0x4c
  [<ffffffff81047f90>] free_task+0x31/0x5c
  [<ffffffff8104807d>] __put_task_struct+0xc2/0xdb
  [<ffffffff8104dfc7>] put_task_struct+0x1e/0x22
  [<ffffffff8104e144>] delayed_put_task_struct+0x7a/0x98
  [<ffffffff810cf0e5>] __rcu_process_callbacks+0x269/0x3df
  [<ffffffff810cf28c>] rcu_process_callbacks+0x31/0x5b
  [<ffffffff8105266d>] __do_softirq+0x122/0x277
```

This usage pattern could not be triggered before kmem came into play.
With the introduction of kmem stack handling, it is possible that we
call the last mem_cgroup_put() from the task destructor, which is run in
an rcu callback. Such callbacks are run with softirqs disabled, leading
to the offensive usage pattern.

In general, we have little, if any, means to guarantee in which context
the last memcg_put will happen. The best we can do is test it and try to
make sure no invalid context releases are happening. But as we add more
code to memcg, the possible interactions grow in number and expose more
ways to get context conflicts. One thing to keep in mind, is that part
of the freeing process is already deferred to a worker, such as vfree(),
that can only be called from process context.

For the moment, the only two functions we really need moved away are:

  * free_css_id(), and
  * mem_cgroup_remove_from_trees().

But because the later accesses per-zone info,
free_mem_cgroup_per_zone_info() needs to be moved as well. With that, we

are left with the per_cpu stats only. Better move it all.

Signed-off-by: Glauber Costa <glommer@parallels.com>
Tested-by: Greg Thelen <gthelen@google.com>
CC: KAMEZAWA Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com>
CC: Michal Hocko <mhocko@suse.cz>
CC: Johannes Weiner <hannes@cmpxchg.org>
CC: Tejun Heo <tj@kernel.org>

```
---
 mm/memcontrol.c | 66 +++++++++++++++++++++++++++++++++---------------------------
 1 file changed, 34 insertions(+), 32 deletions(-)

diff --git a/mm/memcontrol.c b/mm/memcontrol.c
index 2f92f89..c5215f1 100644
--- a/mm/memcontrol.c
+++ b/mm/memcontrol.c
@@ -5205,16 +5205,29 @@ out_free:
 }

 /*
- * 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.
+ * At destroying mem_cgroup, references from swap_cgroup can remain.
+ * (scanning all at force_empty is too costly...)
+ *
+ * Instead of clearing all references at force_empty, we remember
+ * the number of reference from swap_cgroup and free mem_cgroup when
+ * it goes down to 0.
+ *
+ * Removal of cgroup itself succeeds regardless of refs from swap.
  */
-static void free_work(struct work_struct *work)
+
+static void __mem_cgroup_free(struct mem_cgroup *memcg)
 {
- struct mem_cgroup *memcg;
+ int node;
  int size = sizeof(struct mem_cgroup);

- memcg = container_of(work, struct mem_cgroup, work_freeing);
+ mem_cgroup_remove_from_trees(memcg);
+ free_css_id(&mem_cgroup_subsys, &memcg->css);
+
+ for_each_node(node)
+  free_mem_cgroup_per_zone_info(memcg, node);
+
+ free_percpu(memcg->stat);
```

```
+
  /*
   * We need to make sure that (at least for now), the jump label
   * destruction code runs outside of the cgroup lock. This is because
@@ -5233,38 +5246,27 @@ static void free_work(struct work_struct *work)
   vfree(memcg);
 }

-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, free_work);
- schedule_work(&memcg->work_freeing);
-}

 /*
- * At destroying mem_cgroup, references from swap_cgroup can remain.
- * (scanning all at force_empty is too costly...)
- *
- * Instead of clearing all references at force_empty, we remember
- * the number of reference from swap_cgroup and free mem_cgroup when
- * it goes down to 0.
- *
- * Removal of cgroup itself succeeds regardless of refs from swap.
+ * 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 __mem_cgroup_free(struct mem_cgroup *memcg)
+static void free_work(struct work_struct *work)
 {
- int node;
+ struct mem_cgroup *memcg;

- mem_cgroup_remove_from_trees(memcg);
- free_css_id(&mem_cgroup_subsys, &memcg->css);
+ memcg = container_of(work, struct mem_cgroup, work_freeing);
+ __mem_cgroup_free(memcg);
+}

- for_each_node(node)
-  free_mem_cgroup_per_zone_info(memcg, node);
+static void free_rcu(struct rcu_head *rcu_head)
+{
+ struct mem_cgroup *memcg;
```

```
- free_percpu(memcg->stat);
- call_rcu(&memcg->rcu_freeing, free_rcu);
+ memcg = container_of(rcu_head, struct mem_cgroup, rcu_freeing);
+ INIT_WORK(&memcg->work_freeing, free_work);
+ schedule_work(&memcg->work_freeing);
 }

 static void mem_cgroup_get(struct mem_cgroup *memcg)
@@ -5276,7 +5278,7 @@ 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);
-  __mem_cgroup_free(memcg);
+  call_rcu(&memcg->rcu_freeing, free_rcu);
   if (parent)
    mem_cgroup_put(parent);
  }
--
1.7.11.4
```

Subject: [PATCH v4 13/14] protect architectures where THREAD_SIZE &gt;= PAGE_SIZE against fork bombs
Posted by Glauber Costa on Mon, 08 Oct 2012 10:06:19 GMT

View Forum Message <> Reply to Message

Because those architectures will draw their stacks directly from the
page allocator, rather than the slab cache, we can directly pass
__GFP_KMEMCG flag, and issue the corresponding free_pages.

This code path is taken when the architecture doesn't define
CONFIG_ARCH_THREAD_INFO_ALLOCATOR (only ia64 seems to), and has
THREAD_SIZE >= PAGE_SIZE. Luckily, most - if not all - of the remaining
architectures fall in this category.

This will guarantee that every stack page is accounted to the memcg the
process currently lives on, and will have the allocations to fail if
they go over limit.

For the time being, I am defining a new variant of THREADINFO_GFP, not
to mess with the other path. Once the slab is also tracked by memcg, we
can get rid of that flag.

Tested to successfully protect against :(){ :|:& };:

Signed-off-by: Glauber Costa <glommer@parallels.com>
Acked-by: Frederic Weisbecker <fweisbec@redhat.com>

```
Acked-by: Kamezawa Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com>
Reviewed-by: Michal Hocko <mhocko@suse.cz>
CC: Christoph Lameter <cl@linux.com>
CC: Pekka Enberg <penberg@cs.helsinki.fi>
CC: Johannes Weiner <hannes@cmpxchg.org>
CC: Suleiman Souhlal <suleiman@google.com>
---
 include/linux/thread_info.h | 2 ++
 kernel/fork.c               | 4 ++--
 2 files changed, 4 insertions(+), 2 deletions(-)

diff --git a/include/linux/thread_info.h b/include/linux/thread_info.h
index ccc1899..e7e0473 100644
--- a/include/linux/thread_info.h
+++ b/include/linux/thread_info.h
@@ -61,6 +61,8 @@ extern long do_no_restart_syscall(struct restart_block *parm);
 # define THREADINFO_GFP  (GFP_KERNEL | __GFP_NOTRACK)
 #endif

+#define THREADINFO_GFP_ACCOUNTED (THREADINFO_GFP | __GFP_KMEMCG)
+
 /*
  * flag set/clear/test wrappers
  * - pass TIF_xxxx constants to these functions
diff --git a/kernel/fork.c b/kernel/fork.c
index 03b86f1..b3f6298 100644
--- a/kernel/fork.c
+++ b/kernel/fork.c
@@ -146,7 +146,7 @@ void __weak arch_release_thread_info(struct thread_info *ti)
 static struct thread_info *alloc_thread_info_node(struct task_struct *tsk,
         int node)
 {
- struct page *page = alloc_pages_node(node, THREADINFO_GFP,
+ struct page *page = alloc_pages_node(node, THREADINFO_GFP_ACCOUNTED,
           THREAD_SIZE_ORDER);

  return page ? page_address(page) : NULL;
@@ -154,7 +154,7 @@ static struct thread_info *alloc_thread_info_node(struct task_struct *tsk,

 static inline void free_thread_info(struct thread_info *ti)
 {
- free_pages((unsigned long)ti, THREAD_SIZE_ORDER);
+ free_accounted_pages((unsigned long)ti, THREAD_SIZE_ORDER);
 }
 # else
 static struct kmem_cache *thread_info_cache;
--
1.7.11.4
```

Subject: [PATCH v4 14/14] Add documentation about the kmem controller
Posted by Glauber Costa on Mon, 08 Oct 2012 10:06:20 GMT
View Forum Message <> Reply to Message

```
Signed-off-by: Glauber Costa <glommer@parallels.com>
---
 Documentation/cgroups/memory.txt | 55 +++++++++++++++++++++++++++++++++++++++++++++++-
 1 file changed, 54 insertions(+), 1 deletion(-)

diff --git a/Documentation/cgroups/memory.txt b/Documentation/cgroups/memory.txt
index c07f7b4..9b08548 100644
--- a/Documentation/cgroups/memory.txt
+++ b/Documentation/cgroups/memory.txt
@@ -71,6 +71,11 @@ Brief summary of control files.
  memory.oom_control   # set/show oom controls.
  memory.numa_stat   # show the number of memory usage per numa node

+ memory.kmem.limit_in_bytes      # set/show hard limit for kernel memory
+ memory.kmem.usage_in_bytes      # show current kernel memory allocation
+ memory.kmem.failcnt             # show the number of kernel memory usage hits limits
+ memory.kmem.max_usage_in_bytes  # show max kernel memory usage recorded
+
  memory.kmem.tcp.limit_in_bytes  # set/show hard limit for tcp buf memory
  memory.kmem.tcp.usage_in_bytes  # show current tcp buf memory allocation
  memory.kmem.tcp.failcnt            # show the number of tcp buf memory usage hits limits
@@ -268,20 +273,62 @@ the amount of kernel memory used by the system. Kernel memory is fundamentally
 different than user memory, since it can't be swapped out, which makes it
 possible to DoS the system by consuming too much of this precious resource.

+Kernel memory won't be accounted at all until it is limited. This allows for
+existing setups to continue working without disruption. Note that it is
+possible to account it without an effective limit by setting the limits
+to a very high number (like RESOURCE_MAX -1page). The limit cannot be set
+if the cgroup have children, or if there are already tasks in the cgroup.
+
+After a controller is first limited, it will be kept being accounted until it
+is removed. The memory limitation itself, can of course be removed by writing
+-1 to memory.kmem.limit_in_bytes
+
 Kernel memory limits are not imposed for the root cgroup. Usage for the root
-cgroup may or may not be accounted.
+cgroup may or may not be accounted. The memory used is accumulated into
+memory.kmem.usage_in_bytes, or in a separate counter when it makes sense.
+The main "kmem" counter is fed into the main counter, so kmem charges will
+also be visible from the user counter.

 Currently no soft limit is implemented for kernel memory. It is future work
 to trigger slab reclaim when those limits are reached.
 2.7.1 Current Kernel Memory resources accounted

+* stack pages: every process consumes some stack pages. By accounting into
+kernel memory, we prevent new processes from being created when the kernel
+memory usage is too high.
+
 * sockets memory pressure: some sockets protocols have memory pressure
 thresholds. The Memory Controller allows them to be controlled individually
 per cgroup, instead of globally.

 * tcp memory pressure: sockets memory pressure for the tcp protocol.

+2.7.3 Common use cases
+
+Because the "kmem" counter is fed to the main user counter, kernel memory can
+never be limited completely independently of user memory. Say "U" is the user
+limit, and "K" the kernel limit. There are three possible ways limits can be
+set:
+
+    U != 0, K = 0:
+    This is the standard memcg limitation mechanism already present before kmem
+    accounting. Kernel memory is completely ignored.
+
+    U,K != 0, K < U:
+    Kernel memory is effectively set as a percentage of the user memory. This
+    setup is useful in deployments where the total amount of memory per-cgroup
+    is overcommited. Overcommiting kernel memory limits is definitely not
+    recommended, since the box can still run out of non-reclaimable memory.
+    In this case, the admin could set up K so that the sum of all groups is
+    never greater than the total memory, and freely set U at the cost of his
+    QoS.
+
+    U,K != 0, K >= U:
+    Since kmem charges will also be fed to the user counter, this setup gives
+    the admin a unified view of memory. Reclaim will be triggered for the cgroup
+    for both kinds of memory.
+
 3. User Interface

 0. Configuration
@@ -290,6 +337,7 @@ a. Enable CONFIG_CGROUPS
 b. Enable CONFIG_RESOURCE_COUNTERS
 c. Enable CONFIG_MEMCG
 d. Enable CONFIG_MEMCG_SWAP (to use swap extension)
+d. Enable CONFIG_MEMCG_KMEM (to use kmem extension)

 1. Prepare the cgroups (see cgroups.txt, Why are cgroups needed?)
 # mount -t tmpfs none /sys/fs/cgroup
@@ -406,6 +454,11 @@ About use_hierarchy, see Section 6.
   Because rmdir() moves all pages to parent, some out-of-use page caches can be
   moved to the parent. If you want to avoid that, force_empty will be useful.

+  Also, note that when memory.kmem.limit_in_bytes is set the charges due to
+  kernel pages will still be seen. This is not considered a failure and the
+  write will still return success. In this case, it is expected that
+  memory.kmem.usage_in_bytes == memory.usage_in_bytes.
+
   About use_hierarchy, see Section 6.

 5.2 stat file
--
1.7.11.4
```

Subject: Re: [PATCH v4 05/14] Add a __GFP_KMEMCG flag
Posted by Michal Hocko on Tue, 09 Oct 2012 15:04:03 GMT
View Forum Message <> Reply to Message

On Mon 08-10-12 14:06:11, Glauber Costa wrote:
> This flag is used to indicate to the callees that this allocation is a
> kernel allocation in process context, and should be accounted to
> current's memcg. It takes numerical place of the of the recently removed
> __GFP_NO_KSWAPD.
>
> [v4: make flag unconditional, also declare it in trace code]
>
> Signed-off-by: Glauber Costa <glommer@parallels.com>
> CC: Christoph Lameter <cl@linux.com>
> CC: Pekka Enberg <penberg@cs.helsinki.fi>
> CC: Michal Hocko <mhocko@suse.cz>
> CC: Suleiman Souhlal <suleiman@google.com>
> Acked-by: Johannes Weiner <hannes@cmpxchg.org>
> Acked-by: Rik van Riel <riel@redhat.com>
> Acked-by: Mel Gorman <mel@csn.ul.ie>
> Acked-by: Kamezawa Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com>

Acked-by: Michal Hocko <mhocko@suse.cz>

```
> ---
>  include/linux/gfp.h             | 3 ++-
>  include/trace/events/gfpflags.h | 1 +
>  2 files changed, 3 insertions(+), 1 deletion(-)
>
> diff --git a/include/linux/gfp.h b/include/linux/gfp.h
> index 02c1c97..9289d46 100644
> --- a/include/linux/gfp.h
> +++ b/include/linux/gfp.h
> @@ -31,6 +31,7 @@ struct vm_area_struct;
>  #define ___GFP_THISNODE  0x40000u
>  #define ___GFP_RECLAIMABLE 0x80000u
>  #define ___GFP_NOTRACK  0x200000u
> +#define ___GFP_KMEMCG  0x400000u
>  #define ___GFP_OTHER_NODE 0x800000u
>  #define ___GFP_WRITE  0x1000000u
>
> @@ -87,7 +88,7 @@ struct vm_area_struct;
>
>  #define __GFP_OTHER_NODE ((__force gfp_t)___GFP_OTHER_NODE) /* On behalf of other
node */
>  #define __GFP_WRITE ((__force gfp_t)___GFP_WRITE) /* Allocator intends to dirty page */
> -
> +#define __GFP_KMEMCG ((__force gfp_t)___GFP_KMEMCG) /* Allocation comes from a
memcg-accounted resource */
>  /*
>   * This may seem redundant, but it's a way of annotating false positives vs.
>   * allocations that simply cannot be supported (e.g. page tables).
> diff --git a/include/trace/events/gfpflags.h b/include/trace/events/gfpflags.h
> index 9391706..730df12 100644
> --- a/include/trace/events/gfpflags.h
> +++ b/include/trace/events/gfpflags.h
> @@ -36,6 +36,7 @@
>   {(unsigned long)__GFP_RECLAIMABLE, "GFP_RECLAIMABLE"}, \
>   {(unsigned long)__GFP_MOVABLE,  "GFP_MOVABLE"},  \
>   {(unsigned long)__GFP_NOTRACK,  "GFP_NOTRACK"},  \
> + {(unsigned long)__GFP_KMEMCG,  "GFP_KMEMCG"},  \
>   {(unsigned long)__GFP_OTHER_NODE, "GFP_OTHER_NODE"} \
>   ) : "GFP_NOWAIT"
>
> --
> 1.7.11.4
>
> --
> To unsubscribe from this list: send the line "unsubscribe cgroups" in
> the body of a message to majordomo@vger.kernel.org
> More majordomo info at  http://vger.kernel.org/majordomo-info.html
```

--
Michal Hocko
SUSE Labs

Subject: Re: [PATCH v4 08/14] res_counter: return amount of charges after

res_counter_uncharge
Posted by Michal Hocko on Tue, 09 Oct 2012 15:08:45 GMT
View Forum Message <> Reply to Message

On Mon 08-10-12 14:06:14, Glauber Costa wrote:
[...]
```
> diff --git a/kernel/res_counter.c b/kernel/res_counter.c
> index ad581aa..7b3d6dc 100644
> --- a/kernel/res_counter.c
> +++ b/kernel/res_counter.c
> @@ -86,33 +86,39 @@ int res_counter_charge_nofail(struct res_counter *counter, unsigned long val,
>   return __res_counter_charge(counter, val, limit_fail_at, true);
>  }
>
> -void res_counter_uncharge_locked(struct res_counter *counter, unsigned long val)
> +u64 res_counter_uncharge_locked(struct res_counter *counter, unsigned long val)
>  {
>   if (WARN_ON(counter->usage < val))
>    val = counter->usage;
>
>   counter->usage -= val;
> + return counter->usage;
>  }
>
> -void res_counter_uncharge_until(struct res_counter *counter,
> -    struct res_counter *top,
> -    unsigned long val)
> +u64 res_counter_uncharge_until(struct res_counter *counter,
> +          struct res_counter *top,
> +          unsigned long val)
>  {
>   unsigned long flags;
>   struct res_counter *c;
> + u64 ret = 0;
>
>   local_irq_save(flags);
>   for (c = counter; c != top; c = c->parent) {
> +  u64 r;
>    spin_lock(&c->lock);
> -  res_counter_uncharge_locked(c, val);
> +  r = res_counter_uncharge_locked(c, val);
> +  if (c == counter)
> +   ret = r;
>    spin_unlock(&c->lock);
>   }
>   local_irq_restore(flags);
> + return ret;
```

As I have already mentioned in my previous feedback this is cetainly not
atomic as you the lock protects only one group in the hierarchy. How is
the return value from this function supposed to be used?
--
Michal Hocko
SUSE Labs

Subject: Re: [PATCH v4 08/14] res_counter: return amount of charges after res_counter_uncharge
Posted by Glauber Costa on Tue, 09 Oct 2012 15:14:57 GMT
View Forum Message <> Reply to Message

On 10/09/2012 07:08 PM, Michal Hocko wrote:
> As I have already mentioned in my previous feedback this is cetainly not
> atomic as you the lock protects only one group in the hierarchy. How is
> the return value from this function supposed to be used?

So, I tried to make that clearer in the updated changelog.

Only the value of the base memcg (the one passed to the function) is
returned, and it is atomic, in the sense that it has the same semantics
as the atomic variables: If 2 threads uncharge 4k each from a 8 k
counter, a subsequent read can return 0 for both. The return value here
will guarantee that only one sees the drop to 0.

This is used in the patch "kmem_accounting lifecycle management" to be
sure that only one process will call mem_cgroup_put() in the memcg
structure.

Subject: Re: [PATCH v4 08/14] res_counter: return amount of charges after res_counter_uncharge
Posted by Michal Hocko on Tue, 09 Oct 2012 15:35:06 GMT
View Forum Message <> Reply to Message

On Tue 09-10-12 19:14:57, Glauber Costa wrote:
> On 10/09/2012 07:08 PM, Michal Hocko wrote:
> > As I have already mentioned in my previous feedback this is cetainly not
> > atomic as you the lock protects only one group in the hierarchy. How is
> > the return value from this function supposed to be used?
>
> So, I tried to make that clearer in the updated changelog.
>
> Only the value of the base memcg (the one passed to the function) is
> returned, and it is atomic, in the sense that it has the same semantics
> as the atomic variables: If 2 threads uncharge 4k each from a 8 k

> counter, a subsequent read can return 0 for both. The return value here
> will guarantee that only one sees the drop to 0.
>
> This is used in the patch "kmem_accounting lifecycle management" to be
> sure that only one process will call mem_cgroup_put() in the memcg
> structure.

Yes, you are using res_counter_uncharge and its semantic makes sense.
I was refering to res_counter_uncharge_until (you removed that context
from my reply) because that one can race resulting that nobody sees 0
even though that parents get down to 0 as a result:

```
  A
  |
  B
 / \
     C(x)  D(y)
```

D and C uncharge everything.

```
CPU0    CPU1
ret += uncharge(D) [0]  ret += uncharge(C) [0]
ret += uncharge(B) [x-from C]
    ret += uncharge(B) [0]
    ret += uncharge(A) [y-from D]
ret += uncharge(A) [0]

ret == x   ret == y
```

--
Michal Hocko
SUSE Labs

Subject: Re: [PATCH v4 08/14] res_counter: return amount of charges after res_counter_uncharge
Posted by Glauber Costa on Wed, 10 Oct 2012 09:03:39 GMT
View Forum Message <> Reply to Message

On 10/09/2012 07:35 PM, Michal Hocko wrote:
> On Tue 09-10-12 19:14:57, Glauber Costa wrote:
>> On 10/09/2012 07:08 PM, Michal Hocko wrote:
>>> As I have already mentioned in my previous feedback this is cetainly not
>>> atomic as you the lock protects only one group in the hierarchy. How is
>>> the return value from this function supposed to be used?
>>
>> So, I tried to make that clearer in the updated changelog.
>>
>> Only the value of the base memcg (the one passed to the function) is
>> returned, and it is atomic, in the sense that it has the same semantics

```
>> as the atomic variables: If 2 threads uncharge 4k each from a 8 k
>> counter, a subsequent read can return 0 for both. The return value here
>> will guarantee that only one sees the drop to 0.
>>
>> This is used in the patch "kmem_accounting lifecycle management" to be
>> sure that only one process will call mem_cgroup_put() in the memcg
>> structure.
>
> Yes, you are using res_counter_uncharge and its semantic makes sense.
> I was refering to res_counter_uncharge_until (you removed that context
> from my reply) because that one can race resulting that nobody sees 0
> even though that parents get down to 0 as a result:
>   A
>   |
>   B
>  / \
>      C(x)  D(y)
>
> D and C uncharge everything.
>
> CPU0    CPU1
> ret += uncharge(D) [0]  ret += uncharge(C) [0]
> ret += uncharge(B) [x-from C]
>     ret += uncharge(B) [0]
>     ret += uncharge(A) [y-from D]
> ret += uncharge(A) [0]
>
> ret == x   ret == y
>
```

Sorry Michal, I didn't realize you were talking about
res_counter_uncharge_until.

I don't really need res_counter_uncharge_until to return anything, so I
can just remove that if you prefer, keeping just the main
res_counter_uncharge.

However, I still can't make sense of your concern.

The return value will return the value of the counter passed as a
parameter to the function:

```
		r = res_counter_uncharge_locked(c, val);
		if (c == counter)
			ret = r;
```

So when you call res_counter_uncharge_until(D, whatever, x), you will

see zero here as a result, and when you call
res_counter_uncharge_until(D, whatever, y) you will see 0 here as well.

A doesn't get involved with that.

Subject: Re: [PATCH v4 08/14] res_counter: return amount of charges after res_counter_uncharge
Posted by Michal Hocko on Wed, 10 Oct 2012 11:24:47 GMT
View Forum Message <> Reply to Message

On Wed 10-10-12 13:03:39, Glauber Costa wrote:
> On 10/09/2012 07:35 PM, Michal Hocko wrote:
> > On Tue 09-10-12 19:14:57, Glauber Costa wrote:
> >> On 10/09/2012 07:08 PM, Michal Hocko wrote:
> >>> As I have already mentioned in my previous feedback this is cetainly not
> >>> atomic as you the lock protects only one group in the hierarchy. How is
> >>> the return value from this function supposed to be used?
> >>
> >> So, I tried to make that clearer in the updated changelog.
> >>
> >> Only the value of the base memcg (the one passed to the function) is
> >> returned, and it is atomic, in the sense that it has the same semantics
> >> as the atomic variables: If 2 threads uncharge 4k each from a 8 k
> >> counter, a subsequent read can return 0 for both. The return value here
> >> will guarantee that only one sees the drop to 0.
> >>
> >> This is used in the patch "kmem_accounting lifecycle management" to be
> >> sure that only one process will call mem_cgroup_put() in the memcg
> >> structure.
> >
> > Yes, you are using res_counter_uncharge and its semantic makes sense.
> > I was refering to res_counter_uncharge_until (you removed that context
> > from my reply) because that one can race resulting that nobody sees 0
> > even though that parents get down to 0 as a result:
> >   A
> > |
> >   B
> > / \
> >   C(x) D(y)
> >
> > D and C uncharge everything.
> >
> > CPU0   CPU1
> > ret += uncharge(D) [0] ret += uncharge(C) [0]
> > ret += uncharge(B) [x-from C]
> >   ret += uncharge(B) [0]
> >   ret += uncharge(A) [y-from D]

> > ret += uncharge(A) [0]
> >
> > ret == x   ret == y
> >
>
> Sorry Michal, I didn't realize you were talking about
> res_counter_uncharge_until.

I could have been more specific.

> I don't really need res_counter_uncharge_until to return anything, so I
> can just remove that if you prefer, keeping just the main
> res_counter_uncharge.
>
> However, I still can't make sense of your concern.
>
> The return value will return the value of the counter passed as a
> parameter to the function:
>
```
>               r = res_counter_uncharge_locked(c, val);
>               if (c == counter)
>                       ret = r;
```

Dohh. I have no idea where I took ret += r from. Sorry about the noise.

> So when you call res_counter_uncharge_until(D, whatever, x), you will
> see zero here as a result, and when you call
> res_counter_uncharge_until(D, whatever, y) you will see 0 here as well.
>
> A doesn't get involved with that.

You are right.

--
Michal Hocko
SUSE Labs

Subject: Re: [PATCH v4 08/14] res_counter: return amount of charges after res_counter_uncharge
Posted by Michal Hocko on Wed, 10 Oct 2012 11:25:49 GMT
View Forum Message <> Reply to Message

On Mon 08-10-12 14:06:14, Glauber Costa wrote:
> It is useful to know how many charges are still left after a call to
> res_counter_uncharge. While it is possible to issue a res_counter_read
> after uncharge, this can be racy.
>

> If we need, for instance, to take some action when the counters drop
> down to 0, only one of the callers should see it. This is the same
> semantics as the atomic variables in the kernel.
>
> Since the current return value is void, we don't need to worry about
> anything breaking due to this change: nobody relied on that, and only
> users appearing from now on will be checking this value.
>
> Signed-off-by: Glauber Costa <glommer@parallels.com>
> CC: Michal Hocko <mhocko@suse.cz>
> CC: Johannes Weiner <hannes@cmpxchg.org>
> CC: Suleiman Souhlal <suleiman@google.com>
> CC: Kamezawa Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com>

Reviewed-by: Michal Hocko <mhocko@suse.cz>

```
> ---
>  Documentation/cgroups/resource_counter.txt |  7 ++++---
>  include/linux/res_counter.h                | 12 +++++++-----
>  kernel/res_counter.c                       | 20 +++++++++++++-------
>  3 files changed, 24 insertions(+), 15 deletions(-)
>
> diff --git a/Documentation/cgroups/resource_counter.txt
b/Documentation/cgroups/resource_counter.txt
> index 0c4a344..c4d99ed 100644
> --- a/Documentation/cgroups/resource_counter.txt
> +++ b/Documentation/cgroups/resource_counter.txt
> @@ -83,16 +83,17 @@ to work with it.
>   res_counter->lock internally (it must be called with res_counter->lock
>   held). The force parameter indicates whether we can bypass the limit.
>
> - e. void res_counter_uncharge[_locked]
> + e. u64 res_counter_uncharge[_locked]
>     (struct res_counter *rc, unsigned long val)
>
>   When a resource is released (freed) it should be de-accounted
>   from the resource counter it was accounted to.  This is called
> - "uncharging".
> + "uncharging". The return value of this function indicate the amount
> + of charges still present in the counter.
>
>   The _locked routines imply that the res_counter->lock is taken.
>
> - f. void res_counter_uncharge_until
> + f. u64 res_counter_uncharge_until
>    (struct res_counter *rc, struct res_counter *top,
>     unsinged long val)
>
```

```
> diff --git a/include/linux/res_counter.h b/include/linux/res_counter.h
> index 7d7fbe2..4b173b6 100644
> --- a/include/linux/res_counter.h
> +++ b/include/linux/res_counter.h
> @@ -130,14 +130,16 @@ int res_counter_charge_nofail(struct res_counter *counter,
>   *
>   * these calls check for usage underflow and show a warning on the console
>   * _locked call expects the counter->lock to be taken
> + *
> + * returns the total charges still present in @counter.
>   */
>
> -void res_counter_uncharge_locked(struct res_counter *counter, unsigned long val);
> -void res_counter_uncharge(struct res_counter *counter, unsigned long val);
> +u64 res_counter_uncharge_locked(struct res_counter *counter, unsigned long val);
> +u64 res_counter_uncharge(struct res_counter *counter, unsigned long val);
>
> -void res_counter_uncharge_until(struct res_counter *counter,
> -    struct res_counter *top,
> -    unsigned long val);
> +u64 res_counter_uncharge_until(struct res_counter *counter,
> +          struct res_counter *top,
> +          unsigned long val);
>  /**
>   * res_counter_margin - calculate chargeable space of a counter
>   * @cnt: the counter
> diff --git a/kernel/res_counter.c b/kernel/res_counter.c
> index ad581aa..7b3d6dc 100644
> --- a/kernel/res_counter.c
> +++ b/kernel/res_counter.c
> @@ -86,33 +86,39 @@ int res_counter_charge_nofail(struct res_counter *counter, unsigned
long val,
>   return __res_counter_charge(counter, val, limit_fail_at, true);
>  }
>
> -void res_counter_uncharge_locked(struct res_counter *counter, unsigned long val)
> +u64 res_counter_uncharge_locked(struct res_counter *counter, unsigned long val)
>  {
>   if (WARN_ON(counter->usage < val))
>    val = counter->usage;
>
>   counter->usage -= val;
> + return counter->usage;
>  }
>
> -void res_counter_uncharge_until(struct res_counter *counter,
> -    struct res_counter *top,
> -    unsigned long val)
> +u64 res_counter_uncharge_until(struct res_counter *counter,
> +          struct res_counter *top,
> +          unsigned long val)
>  {
>   unsigned long flags;
>   struct res_counter *c;
> + u64 ret = 0;
>
>   local_irq_save(flags);
>   for (c = counter; c != top; c = c->parent) {
> +  u64 r;
>    spin_lock(&c->lock);
> -  res_counter_uncharge_locked(c, val);
> +  r = res_counter_uncharge_locked(c, val);
> +  if (c == counter)
> +   ret = r;
>    spin_unlock(&c->lock);
>   }
>   local_irq_restore(flags);
> + return ret;
>  }
>
> -void res_counter_uncharge(struct res_counter *counter, unsigned long val)
> +u64 res_counter_uncharge(struct res_counter *counter, unsigned long val)
>  {
> - res_counter_uncharge_until(counter, NULL, val);
> + return res_counter_uncharge_until(counter, NULL, val);
>  }
>
>  static inline unsigned long long *
> --
> 1.7.11.4
>
> --
> To unsubscribe from this list: send the line "unsubscribe cgroups" in
> the body of a message to majordomo@vger.kernel.org
> More majordomo info at  http://vger.kernel.org/majordomo-info.html
```

--
Michal Hocko
SUSE Labs

Subject: Re: [PATCH v4 04/14] kmem accounting basic infrastructure
Posted by Michal Hocko on Thu, 11 Oct 2012 10:11:19 GMT
View Forum Message <> Reply to Message

On Mon 08-10-12 14:06:10, Glauber Costa wrote:

```
> This patch adds the basic infrastructure for the accounting of the slab
> caches. To control that, the following files are created:
>
>  * memory.kmem.usage_in_bytes
>  * memory.kmem.limit_in_bytes
>  * memory.kmem.failcnt
>  * memory.kmem.max_usage_in_bytes
>
> They have the same meaning of their user memory counterparts. They
> reflect the state of the "kmem" res_counter.
>
> Per cgroup slab memory accounting is not enabled until a limit is set

s/slab/kmem/ right?

> for the group. Once the limit is set the accounting cannot be disabled
> for that group.  This means that after the patch is applied, no
> behavioral changes exists for whoever is still using memcg to control
> their memory usage, until memory.kmem.limit_in_bytes is set for the
> first time.
>
> We always account to both user and kernel resource_counters. This
> effectively means that an independent kernel limit is in place when the
> limit is set to a lower value than the user memory. A equal or higher
> value means that the user limit will always hit first, meaning that kmem
> is effectively unlimited.
>
> People who want to track kernel memory but not limit it, can set this
> limit to a very high number (like RESOURCE_MAX - 1page - that no one
> will ever hit, or equal to the user memory)
>
> [ v4: make kmem files part of the main array;
>       do not allow limit to be set for non-empty cgroups ]
>
> Signed-off-by: Glauber Costa <glommer@parallels.com>
> CC: Michal Hocko <mhocko@suse.cz>
> CC: Johannes Weiner <hannes@cmpxchg.org>
> Acked-by: Kamezawa Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com>
> ---
>  mm/memcontrol.c | 123
+++++++++++++++++++++++++++++++++++++++++++++++++++++++++-
>  1 file changed, 122 insertions(+), 1 deletion(-)
>
> diff --git a/mm/memcontrol.c b/mm/memcontrol.c
> index 71d259e..ba855cc 100644
> --- a/mm/memcontrol.c
> +++ b/mm/memcontrol.c
[...]
```

```
> @@ -332,6 +337,26 @@ struct mem_cgroup {
>  #endif
>  };
>
> +/* internal only representation about the status of kmem accounting. */
> +enum {
> + KMEM_ACCOUNTED_ACTIVE = 0, /* accounted by this cgroup itself */
> +};
> +
> +/* first bit */
> +#define KMEM_ACCOUNTED_MASK 0x1
> +
> +#ifdef CONFIG_MEMCG_KMEM
> +static void memcg_kmem_set_active(struct mem_cgroup *memcg)
> +{
> + set_bit(KMEM_ACCOUNTED_ACTIVE, &memcg->kmem_accounted);
> +}
> +
> +static bool memcg_kmem_is_accounted(struct mem_cgroup *memcg)
> +{
> + return test_bit(KMEM_ACCOUNTED_ACTIVE, &memcg->kmem_accounted);
> +}
> +#endif
```

set_active vs. is_accounted. Is there any reason for inconsistency here?

```
> +
>  /* Stuffs for move charges at task migration. */
>  /*
>   * Types of charges to be moved. "move_charge_at_immitgrate" is treated as a
[...]
> @@ -3947,6 +3980,58 @@ static ssize_t mem_cgroup_read(struct cgroup *cont, struct cftype
*cft,
>   len = scnprintf(str, sizeof(str), "%llu\n", (unsigned long long)val);
>   return simple_read_from_buffer(buf, nbytes, ppos, str, len);
>  }
> +
> +static int memcg_update_kmem_limit(struct cgroup *cont, u64 val)
> +{
> + int ret = -EINVAL;
> +#ifdef CONFIG_MEMCG_KMEM
> + struct mem_cgroup *memcg = mem_cgroup_from_cont(cont);
> + /*
> +  * For simplicity, we won't allow this to be disabled.  It also can't
> +  * be changed if the cgroup has children already, or if tasks had
> +  * already joined.
> +  *
> +  * If tasks join before we set the limit, a person looking at
> +  * kmem.usage_in_bytes will have no way to determine when it took
> +  * place, which makes the value quite meaningless.
> +  *
> +  * After it first became limited, changes in the value of the limit are
> +  * of course permitted.
> +  *
> +  * Taking the cgroup_lock is really offensive, but it is so far the only
> +  * way to guarantee that no children will appear. There are plenty of
> +  * other offenders, and they should all go away. Fine grained locking
> +  * is probably the way to go here. When we are fully hierarchical, we
> +  * can also get rid of the use_hierarchy check.
> +  */
> + cgroup_lock();
> + mutex_lock(&set_limit_mutex);
> + if (!memcg->kmem_accounted && val != RESOURCE_MAX) {
```

Just a nit but wouldn't memcg_kmem_is_accounted(memcg) be better than
directly checking kmem_accounted?
Besides that I am not sure I fully understand RESOURCE_MAX test. Say I
want to have kmem accounting for monitoring so I do
echo -1 > memory.kmem.limit_in_bytes

so you set the value but do not activate it. Isn't this just a reminder
from the time when the accounting could be deactivated?

```
> +  if (cgroup_task_count(cont) || (memcg->use_hierarchy &&
> +      !list_empty(&cont->children))) {
> +   ret = -EBUSY;
> +   goto out;
> +  }
> +  ret = res_counter_set_limit(&memcg->kmem, val);
```

VM_BUG_IN(ret) ?
There shouldn't be any usage when you enable it or something bad is
going on.

```
> +  if (ret)
> +   goto out;
> +
> +  memcg_kmem_set_active(memcg);
> + } else
> +  ret = res_counter_set_limit(&memcg->kmem, val);
> +out:
> + mutex_unlock(&set_limit_mutex);
> + cgroup_unlock();
> +#endif
> + return ret;
> +}
```

> +
> +static void memcg_propagate_kmem(struct mem_cgroup *memcg,
> +   struct mem_cgroup *parent)
> +{
> + memcg->kmem_accounted = parent->kmem_accounted;
> +}
> +
> /*
>   * The user of this function is...
>   * RES_LIMIT.
[...]
--
Michal Hocko
SUSE Labs

Subject: Re: [PATCH v4 06/14] memcg: kmem controller infrastructure
Posted by Michal Hocko on Thu, 11 Oct 2012 12:42:12 GMT

View Forum Message <> Reply to Message

On Mon 08-10-12 14:06:12, Glauber Costa wrote:
> This patch introduces infrastructure for tracking kernel memory pages to
> a given memcg. This will happen whenever the caller includes the flag
> __GFP_KMEMCG flag, and the task belong to a memcg other than the root.
>
> In memcontrol.h those functions are wrapped in inline acessors. The
> idea is to later on, patch those with static branches, so we don't incur
> any overhead when no mem cgroups with limited kmem are being used.
>
> Users of this functionality shall interact with the memcg core code
> through the following functions:
>
> memcg_kmem_newpage_charge: will return true if the group can handle the
>   allocation. At this point, struct page is not
>   yet allocated.
>
> memcg_kmem_commit_charge: will either revert the charge, if struct page
>   allocation failed, or embed memcg information
>   into page_cgroup.
>
> memcg_kmem_uncharge_page: called at free time, will revert the charge.
>
> [v2: improved comments and standardized function names]
> [v3: handle no longer opaque, functions not exported,
>   even more comments]
> [v4: reworked Used bit handling and surroundings for more clarity]
> [v5: simplified code for kmemcg compiled out and core functions in
>   memcontrol.c, moved kmem code to the middle to avoid forward decls]

```
>
> Signed-off-by: Glauber Costa <glommer@parallels.com>
> CC: Christoph Lameter <cl@linux.com>
> CC: Pekka Enberg <penberg@cs.helsinki.fi>
> CC: Michal Hocko <mhocko@suse.cz>
> CC: Kamezawa Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com>
> CC: Johannes Weiner <hannes@cmpxchg.org>
> ---
>  include/linux/memcontrol.h |  95 +++++++++++++++++++++++++
>  mm/memcontrol.c            | 173 ++++++++++++++++++++++++++++++++++++++++++++++++++--
>  2 files changed, 263 insertions(+), 5 deletions(-)
>
> diff --git a/include/linux/memcontrol.h b/include/linux/memcontrol.h
> index 8d9489f..783cd78 100644
> --- a/include/linux/memcontrol.h
> +++ b/include/linux/memcontrol.h
> @@ -21,6 +21,7 @@
>  #define _LINUX_MEMCONTROL_H
>  #include <linux/cgroup.h>
>  #include <linux/vm_event_item.h>
> +#include <linux/hardirq.h>
>
>  struct mem_cgroup;
>  struct page_cgroup;
> @@ -399,6 +400,85 @@ struct sock;
>  #ifdef CONFIG_MEMCG_KMEM
>  void sock_update_memcg(struct sock *sk);
>  void sock_release_memcg(struct sock *sk);
> +
> +static inline bool memcg_kmem_enabled(void)
> +{
> + return true;
> +}
> +V
> +extern bool __memcg_kmem_newpage_charge(gfp_t gfp, struct mem_cgroup **memcg,
> +     int order);
> +extern void __memcg_kmem_commit_charge(struct page *page,
> +           struct mem_cgroup *memcg, int order);
> +extern void __memcg_kmem_uncharge_page(struct page *page, int order);
```

Just a nit. Hmm we are far from being consisten in using vs. not using
externs in header files for function declarations but I do not see any
reason why to use them here. Names are just longer without any
additional value.

```
[...]
> +static int memcg_charge_kmem(struct mem_cgroup *memcg, gfp_t gfp, u64 size)
> +{
```

```
> + struct res_counter *fail_res;
> + struct mem_cgroup *_memcg;
> + int ret = 0;
> + bool may_oom;
> +
> + /*
> +  * Conditions under which we can wait for the oom_killer.
> +  * __GFP_NORETRY should be masked by __mem_cgroup_try_charge,
> +  * but there is no harm in being explicit here
> +  */
> + may_oom = (gfp & __GFP_WAIT) && !(gfp & __GFP_NORETRY);
```

Well we _have to_ check __GFP_NORETRY here because if we don't then we
can end up in OOM. mem_cgroup_do_charge returns CHARGE_NOMEM for
__GFP_NORETRY (without doing any reclaim) and of oom==true we decrement
oom retries counter and eventually hit OOM killer. So the comment is
misleading.

```
> +
> + _memcg = memcg;
> + ret = __mem_cgroup_try_charge(NULL, gfp, size >> PAGE_SHIFT,
> +          &_memcg, may_oom);
> +
> + if (!ret) {
> +  ret = res_counter_charge(&memcg->kmem, size, &fail_res);
```

Now that I'm thinking about the charging ordering we should charge the
kmem first because we would like to hit kmem limit before we hit u+k
limit, don't we.
Say that you have kmem limit 10M and the total limit 50M. Current `u'
would be 40M and this charge would cause kmem to hit the `k' limit. I
think we should fail to charge kmem before we go to u+k and potentially
reclaim/oom.
Or has this been alredy discussed and I just do not remember?

```
> +  if (ret) {
> +   res_counter_uncharge(&memcg->res, size);
> +   if (do_swap_account)
> +    res_counter_uncharge(&memcg->memsw, size);
> +  }
[...]
> +bool
> +__memcg_kmem_newpage_charge(gfp_t gfp, struct mem_cgroup **_memcg, int order)
> +{
> + struct mem_cgroup *memcg;
> + int ret;
> +
> + *_memcg = NULL;
> + memcg = try_get_mem_cgroup_from_mm(current->mm);
```

> +
> + /*
> + * very rare case described in mem_cgroup_from_task. Unfortunately there
> + * isn't much we can do without complicating this too much, and it would
> + * be gfp-dependent anyway. Just let it go
> + */
> + if (unlikely(!memcg))
> + return true;
> +
> + if (!memcg_can_account_kmem(memcg)) {
> + css_put(&memcg->css);
> + return true;
> + }
> +
 /*
  * Keep reference on memcg while the page is charged to prevent
  * group from vanishing because allocation can outlive their
  * tasks. The reference is dropped in __memcg_kmem_uncharge_page
  */

please
> + mem_cgroup_get(memcg);
> +
> + ret = memcg_charge_kmem(memcg, gfp, PAGE_SIZE << order);
> + if (!ret)
> + *_memcg = memcg;
> + else
> + mem_cgroup_put(memcg);
> +
> + css_put(&memcg->css);
> + return (ret == 0);
> +}
> +
[...]
--
Michal Hocko
SUSE Labs

Subject: Re: [PATCH v4 04/14] kmem accounting basic infrastructure
Posted by Michal Hocko on Thu, 11 Oct 2012 12:53:10 GMT
View Forum Message <> Reply to Message

On Thu 11-10-12 12:11:19, Michal Hocko wrote:
> On Mon 08-10-12 14:06:10, Glauber Costa wrote:
[...]
> > +static void memcg_kmem_set_active(struct mem_cgroup *memcg)
> > +{

> > + set_bit(KMEM_ACCOUNTED_ACTIVE, &memcg->kmem_accounted);
> > +}
> > +
> > +static bool memcg_kmem_is_accounted(struct mem_cgroup *memcg)
> > +{
> > + return test_bit(KMEM_ACCOUNTED_ACTIVE, &memcg->kmem_accounted);
> > +}
> > +#endif
>
> set_active vs. is_accounted. Is there any reason for inconsistency here?

Ahh, fixed later and 09/14 makes it memcg_kmem_is_active so this is just
a code churn. I think making it memcg_kmem_is_active here would be
better.

--
Michal Hocko
SUSE Labs

Subject: Re: [PATCH v4 06/14] memcg: kmem controller infrastructure
Posted by Michal Hocko on Thu, 11 Oct 2012 12:56:03 GMT
View Forum Message <> Reply to Message

On Thu 11-10-12 14:42:12, Michal Hocko wrote:
[...]
> /*
>   * Keep reference on memcg while the page is charged to prevent
>   * group from vanishing because allocation can outlive their
>   * tasks. The reference is dropped in __memcg_kmem_uncharge_page
>   */
>
> please
> > + mem_cgroup_get(memcg);

Ahh, this will go away. The it doesn't make much sense to add the
comment here.
--
Michal Hocko
SUSE Labs

Subject: Re: [PATCH v4 09/14] memcg: kmem accounting lifecycle management
Posted by Michal Hocko on Thu, 11 Oct 2012 13:11:43 GMT
View Forum Message <> Reply to Message

On Mon 08-10-12 14:06:15, Glauber Costa wrote:

> Because kmem charges can outlive the cgroup, we need to make sure that
> we won't free the memcg structure while charges are still in flight.
> For reviewing simplicity, the charge functions will issue
> mem_cgroup_get() at every charge, and mem_cgroup_put() at every
> uncharge.
>
> This can get expensive, however, and we can do better. mem_cgroup_get()
> only really needs to be issued once: when the first limit is set. In the
> same spirit, we only need to issue mem_cgroup_put() when the last charge
> is gone.
>
> We'll need an extra bit in kmem_accounted for that: KMEM_ACCOUNTED_DEAD.
> it will be set when the cgroup dies, if there are charges in the group.
> If there aren't, we can proceed right away.
>
> Our uncharge function will have to test that bit every time the charges
> drop to 0. Because that is not the likely output of
> res_counter_uncharge, this should not impose a big hit on us: it is
> certainly much better than a reference count decrease at every
> operation.
>
> [v3: merged all lifecycle related patches in one]
>
> Signed-off-by: Glauber Costa <glommer@parallels.com>
> CC: Kamezawa Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com>
> CC: Christoph Lameter <cl@linux.com>
> CC: Pekka Enberg <penberg@cs.helsinki.fi>
> CC: Michal Hocko <mhocko@suse.cz>
> CC: Johannes Weiner <hannes@cmpxchg.org>
> CC: Suleiman Souhlal <suleiman@google.com>

OK, I like the optimization. I have just one comment to the
memcg_kmem_dead naming but other than that

Acked-by: Michal Hocko <mhocko@suse.cz>

[...]
> +static bool memcg_kmem_dead(struct mem_cgroup *memcg)

The name is tricky because it doesn't tell you that it clears the flag
which made me scratch my head when reading comment in kmem_cgroup_destroy

```
> +{
> + return test_and_clear_bit(KMEM_ACCOUNTED_DEAD, &memcg->kmem_accounted);
> +}
>  #endif
>
>  /* Stuffs for move charges at task migration. */
```

[...]
> @@ -4876,6 +4904,20 @@ static int memcg_init_kmem(struct mem_cgroup *memcg, struct cgroup_subsys *ss)
> static void kmem_cgroup_destroy(struct mem_cgroup *memcg)
> {
>   mem_cgroup_sockets_destroy(memcg);
> +
> + memcg_kmem_mark_dead(memcg);
> +
> + if (res_counter_read_u64(&memcg->kmem, RES_USAGE) != 0)
> + return;
> +
> + /*
> + * Charges already down to 0, undo mem_cgroup_get() done in the charge
> + * path here, being careful not to race with memcg_uncharge_kmem: it is
> + * possible that the charges went down to 0 between mark_dead and the
> + * res_counter read, so in that case, we don't need the put
> + */
> + if (memcg_kmem_dead(memcg))
> + mem_cgroup_put(memcg);

--
Michal Hocko
SUSE Labs

Subject: Re: [PATCH v4 04/14] kmem accounting basic infrastructure
Posted by Michal Hocko on Thu, 11 Oct 2012 13:38:56 GMT
View Forum Message <> Reply to Message

On Thu 11-10-12 12:11:19, Michal Hocko wrote:
> On Mon 08-10-12 14:06:10, Glauber Costa wrote:

> > + cgroup_lock();
> > + mutex_lock(&set_limit_mutex);
> > + if (!memcg->kmem_accounted && val != RESOURCE_MAX) {
>
> Just a nit but wouldn't memcg_kmem_is_accounted(memcg) be better than
> directly checking kmem_accounted?

OK, I see that jump lable patch changes this and we set activated inside
the conditional so kmem_accounted check catches both flags. That could
be changed to memcg_kmem_is_activated in that patch but what ever.
--
Michal Hocko
SUSE Labs

Subject: Re: [PATCH v4 10/14] memcg: use static branches when code not in use
Posted by Michal Hocko on Thu, 11 Oct 2012 13:40:28 GMT
View Forum Message <> Reply to Message

On Mon 08-10-12 14:06:16, Glauber Costa wrote:
> We can use static branches to patch the code in or out when not used.
>
> Because the _ACTIVE bit on kmem_accounted is only set after the
> increment is done, we guarantee that the root memcg will always be
> selected for kmem charges until all call sites are patched (see
> memcg_kmem_enabled). This guarantees that no mischarges are applied.
>
> static branch decrement happens when the last reference count from the
> kmem accounting in memcg dies. This will only happen when the charges
> drop down to 0.
>
> When that happen, we need to disable the static branch only on those
> memcgs that enabled it. To achieve this, we would be forced to
> complicate the code by keeping track of which memcgs were the ones
> that actually enabled limits, and which ones got it from its parents.
>
> It is a lot simpler just to do static_key_slow_inc() on every child
> that is accounted.
>
> [v4: adapted this patch to the changes in kmem_accounted]
>
> Signed-off-by: Glauber Costa <glommer@parallels.com>
> CC: Kamezawa Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com>
> CC: Christoph Lameter <cl@linux.com>
> CC: Pekka Enberg <penberg@cs.helsinki.fi>
> CC: Michal Hocko <mhocko@suse.cz>
> CC: Johannes Weiner <hannes@cmpxchg.org>
> CC: Suleiman Souhlal <suleiman@google.com>

Looks reasonable to me
Acked-by: Michal Hocko <mhocko@suse.cz>

Just a little nit.

[...]

```
> diff --git a/mm/memcontrol.c b/mm/memcontrol.c
> index 634c7b5..724a08b 100644
> --- a/mm/memcontrol.c
> +++ b/mm/memcontrol.c
> @@ -344,11 +344,15 @@ struct mem_cgroup {
>  /* internal only representation about the status of kmem accounting. */
>  enum {
>   KMEM_ACCOUNTED_ACTIVE = 0, /* accounted by this cgroup itself */
```

> + KMEM_ACCOUNTED_ACTIVATED, /* static key enabled. */
>   KMEM_ACCOUNTED_DEAD, /* dead memcg, pending kmem charges */
> };
>
> -/* first bit */
> -#define KMEM_ACCOUNTED_MASK 0x1
> +/*
> + * first two bits. We account when limit is on, but only after
> + * call sites are patched
> + */
> +#define KMEM_ACCOUNTED_MASK 0x3

The names are long but why not use KMEM_ACCOUNTED_ACTIVE*
#define KMEM_ACCOUNTED_MASK 1<<KMEM_ACCOUNTED_ACTIVE |
1<<KMEM_ACCOUNTED_ACTIVATED

[...]
--
Michal Hocko
SUSE Labs

Subject: Re: [PATCH v4 12/14] execute the whole memcg freeing in free_worker
Posted by Michal Hocko on Thu, 11 Oct 2012 14:21:27 GMT

View Forum Message <> Reply to Message

On Mon 08-10-12 14:06:18, Glauber Costa wrote:
> A lot of the initialization we do in mem_cgroup_create() is done with
> softirqs enabled. This include grabbing a css id, which holds
> &ss->id_lock->rlock, and the per-zone trees, which holds
> rtpz->lock->rlock. All of those signal to the lockdep mechanism that
> those locks can be used in SOFTIRQ-ON-W context. This means that the
> freeing of memcg structure must happen in a compatible context,
> otherwise we'll get a deadlock, like the one bellow, caught by lockdep:
>
```
>   [<ffffffff81103095>] free_accounted_pages+0x47/0x4c
>   [<ffffffff81047f90>] free_task+0x31/0x5c
>   [<ffffffff8104807d>] __put_task_struct+0xc2/0xdb
>   [<ffffffff8104dfc7>] put_task_struct+0x1e/0x22
>   [<ffffffff8104e144>] delayed_put_task_struct+0x7a/0x98
>   [<ffffffff810cf0e5>] __rcu_process_callbacks+0x269/0x3df
>   [<ffffffff810cf28c>] rcu_process_callbacks+0x31/0x5b
>   [<ffffffff8105266d>] __do_softirq+0x122/0x277
```
>
> This usage pattern could not be triggered before kmem came into play.
> With the introduction of kmem stack handling, it is possible that we
> call the last mem_cgroup_put() from the task destructor, which is run in
> an rcu callback. Such callbacks are run with softirqs disabled, leading

> to the offensive usage pattern.
>
> In general, we have little, if any, means to guarantee in which context
> the last memcg_put will happen. The best we can do is test it and try to
> make sure no invalid context releases are happening. But as we add more
> code to memcg, the possible interactions grow in number and expose more
> ways to get context conflicts. One thing to keep in mind, is that part
> of the freeing process is already deferred to a worker, such as vfree(),
> that can only be called from process context.
>
> For the moment, the only two functions we really need moved away are:
>
>   * free_css_id(), and
>   * mem_cgroup_remove_from_trees().
>
> But because the later accesses per-zone info,
> free_mem_cgroup_per_zone_info() needs to be moved as well. With that, we
> are left with the per_cpu stats only. Better move it all.
>
> Signed-off-by: Glauber Costa <glommer@parallels.com>
> Tested-by: Greg Thelen <gthelen@google.com>
> CC: KAMEZAWA Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com>
> CC: Michal Hocko <mhocko@suse.cz>
> CC: Johannes Weiner <hannes@cmpxchg.org>
> CC: Tejun Heo <tj@kernel.org>

OK, it seems it is much easier this way.
Acked-by: Michal Hocko <mhocko@suse.cz>

```
> ---
>  mm/memcontrol.c | 66 +++++++++++++++++++++++++++++++++-------------------------------
>  1 file changed, 34 insertions(+), 32 deletions(-)
>
> diff --git a/mm/memcontrol.c b/mm/memcontrol.c
> index 2f92f89..c5215f1 100644
> --- a/mm/memcontrol.c
> +++ b/mm/memcontrol.c
> @@ -5205,16 +5205,29 @@ out_free:
>  }
>
>  /*
> - * 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.
> + * At destroying mem_cgroup, references from swap_cgroup can remain.
> + * (scanning all at force_empty is too costly...)
> + *
> + * Instead of clearing all references at force_empty, we remember
> + * the number of reference from swap_cgroup and free mem_cgroup when
> + * it goes down to 0.
> + *
> + * Removal of cgroup itself succeeds regardless of refs from swap.
>   */
> -static void free_work(struct work_struct *work)
> +
> +static void __mem_cgroup_free(struct mem_cgroup *memcg)
>  {
> - struct mem_cgroup *memcg;
> + int node;
>   int size = sizeof(struct mem_cgroup);
>
> - memcg = container_of(work, struct mem_cgroup, work_freeing);
> + mem_cgroup_remove_from_trees(memcg);
> + free_css_id(&mem_cgroup_subsys, &memcg->css);
> +
> + for_each_node(node)
> +  free_mem_cgroup_per_zone_info(memcg, node);
> +
> + free_percpu(memcg->stat);
> +
>   /*
>    * We need to make sure that (at least for now), the jump label
>    * destruction code runs outside of the cgroup lock. This is because
> @@ -5233,38 +5246,27 @@ static void free_work(struct work_struct *work)
>    vfree(memcg);
>  }
>
> -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, free_work);
> - schedule_work(&memcg->work_freeing);
> -}
>
>  /*
> - * At destroying mem_cgroup, references from swap_cgroup can remain.
> - * (scanning all at force_empty is too costly...)
> - *
> - * Instead of clearing all references at force_empty, we remember
> - * the number of reference from swap_cgroup and free mem_cgroup when
> - * it goes down to 0.
> - *
> - * Removal of cgroup itself succeeds regardless of refs from swap.
> + * 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 __mem_cgroup_free(struct mem_cgroup *memcg)
> +static void free_work(struct work_struct *work)
>  {
> - int node;
> + struct mem_cgroup *memcg;
>
> - mem_cgroup_remove_from_trees(memcg);
> - free_css_id(&mem_cgroup_subsys, &memcg->css);
> + memcg = container_of(work, struct mem_cgroup, work_freeing);
> + __mem_cgroup_free(memcg);
> +}
>
> - for_each_node(node)
> -  free_mem_cgroup_per_zone_info(memcg, node);
> +static void free_rcu(struct rcu_head *rcu_head)
> +{
> + struct mem_cgroup *memcg;
>
> - free_percpu(memcg->stat);
> - call_rcu(&memcg->rcu_freeing, free_rcu);
> + memcg = container_of(rcu_head, struct mem_cgroup, rcu_freeing);
> + INIT_WORK(&memcg->work_freeing, free_work);
> + schedule_work(&memcg->work_freeing);
>  }
>
>  static void mem_cgroup_get(struct mem_cgroup *memcg)
> @@ -5276,7 +5278,7 @@ 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);
> -  __mem_cgroup_free(memcg);
> +  call_rcu(&memcg->rcu_freeing, free_rcu);
>    if (parent)
>     mem_cgroup_put(parent);
>   }
> --
> 1.7.11.4
>
> --
> To unsubscribe from this list: send the line "unsubscribe cgroups" in
> the body of a message to majordomo@vger.kernel.org
> More majordomo info at  http://vger.kernel.org/majordomo-info.html
```

--
Michal Hocko
SUSE Labs

Subject: Re: [PATCH v4 14/14] Add documentation about the kmem controller
Posted by Michal Hocko on Thu, 11 Oct 2012 14:35:59 GMT
View Forum Message <> Reply to Message

On Mon 08-10-12 14:06:20, Glauber Costa wrote:
```
> Signed-off-by: Glauber Costa <glommer@parallels.com>
> ---
>  Documentation/cgroups/memory.txt | 55 +++++++++++++++++++++++++++++++++++++++-
>  1 file changed, 54 insertions(+), 1 deletion(-)
>
> diff --git a/Documentation/cgroups/memory.txt b/Documentation/cgroups/memory.txt
> index c07f7b4..9b08548 100644
> --- a/Documentation/cgroups/memory.txt
> +++ b/Documentation/cgroups/memory.txt
> @@ -71,6 +71,11 @@ Brief summary of control files.
>   memory.oom_control   # set/show oom controls.
>   memory.numa_stat   # show the number of memory usage per numa node
>
> + memory.kmem.limit_in_bytes      # set/show hard limit for kernel memory
> + memory.kmem.usage_in_bytes      # show current kernel memory allocation
> + memory.kmem.failcnt             # show the number of kernel memory usage hits limits
> + memory.kmem.max_usage_in_bytes  # show max kernel memory usage recorded
> +
>   memory.kmem.tcp.limit_in_bytes  # set/show hard limit for tcp buf memory
>   memory.kmem.tcp.usage_in_bytes  # show current tcp buf memory allocation
>   memory.kmem.tcp.failcnt            # show the number of tcp buf memory usage hits limits
> @@ -268,20 +273,62 @@ the amount of kernel memory used by the system. Kernel memory is fundamentally
>  different than user memory, since it can't be swapped out, which makes it
>  possible to DoS the system by consuming too much of this precious resource.
>
> +Kernel memory won't be accounted at all until it is limited. This allows for
```

until limit on a group is set.

```
> +existing setups to continue working without disruption. Note that it is
> +possible to account it without an effective limit by setting the limits
> +to a very high number (like RESOURCE_MAX -1page).
```

I have brought that up in an earlier patch already. Why not just do echo
-1 (which translates to RESOURCE_MAX internally) and be done with that.
RESOURCE_MAX-1 sounds quite inconvenient.

> The limit cannot be set
> +if the cgroup have children, or if there are already tasks in the cgroup.

I would start by stating that if children are accounted automatically if
their parent is accounted already and there is no need to set a limit to
enforce that. In fact the limit cannot be set if ....

> +
> +After a controller is first limited, it will be kept being accounted until it

group is limited not the controller.

> +is removed. The memory limitation itself, can of course be removed by writing
> +-1 to memory.kmem.limit_in_bytes

This might be confusing and one could think that also accounting would
be removed. I wouldn't mention it at all.
> +
> Kernel memory limits are not imposed for the root cgroup. Usage for the root
> -cgroup may or may not be accounted.
> +cgroup may or may not be accounted. The memory used is accumulated into
> +memory.kmem.usage_in_bytes, or in a separate counter when it makes sense.

Which separate counter? Is this about tcp kmem?

> +The main "kmem" counter is fed into the main counter, so kmem charges will
> +also be visible from the user counter.
>
> Currently no soft limit is implemented for kernel memory. It is future work
> to trigger slab reclaim when those limits are reached.
>
> 2.7.1 Current Kernel Memory resources accounted
>
> +* stack pages: every process consumes some stack pages. By accounting into
> +kernel memory, we prevent new processes from being created when the kernel
> +memory usage is too high.
> +
> * sockets memory pressure: some sockets protocols have memory pressure
> thresholds. The Memory Controller allows them to be controlled individually
> per cgroup, instead of globally.
>
> * tcp memory pressure: sockets memory pressure for the tcp protocol.
>
> +2.7.3 Common use cases
> +
> +Because the "kmem" counter is fed to the main user counter, kernel memory can
> +never be limited completely independently of user memory. Say "U" is the user
> +limit, and "K" the kernel limit. There are three possible ways limits can be

> +set:
> +
> +   U != 0, K = 0:

K is not 0 it is unaccounted (disabled)

> +   This is the standard memcg limitation mechanism already present before kmem
> +   accounting. Kernel memory is completely ignored.
> +
> +   U,K != 0, K < U:

I would keep K < U
> +   Kernel memory is effectively set as a percentage of the user memory. This

not a percentage it is subset of the user memory

> +   setup is useful in deployments where the total amount of memory per-cgroup
> +   is overcommited. Overcommiting kernel memory limits is definitely not
> +   recommended, since the box can still run out of non-reclaimable memory.
> +   In this case, the admin could set up K so that the sum of all groups is
> +   never greater than the total memory, and freely set U at the cost of his
> +   QoS.
> +
> +   U,K != 0, K >= U:
> +   Since kmem charges will also be fed to the user counter, this setup gives
> +   the admin a unified view of memory. Reclaim will be triggered for the cgroup
> +   for both kinds of memory.

This is also useful for tracking kernel memory allocation.

> +
> 3. User Interface
>
> 0. Configuration
> @@ -290,6 +337,7 @@ a. Enable CONFIG_CGROUPS
> b. Enable CONFIG_RESOURCE_COUNTERS
> c. Enable CONFIG_MEMCG
> d. Enable CONFIG_MEMCG_SWAP (to use swap extension)
> +d. Enable CONFIG_MEMCG_KMEM (to use kmem extension)
>
> 1. Prepare the cgroups (see cgroups.txt, Why are cgroups needed?)
> # mount -t tmpfs none /sys/fs/cgroup
> @@ -406,6 +454,11 @@ About use_hierarchy, see Section 6.
>   Because rmdir() moves all pages to parent, some out-of-use page caches can be
>   moved to the parent. If you want to avoid that, force_empty will be useful.
>
> + Also, note that when memory.kmem.limit_in_bytes is set the charges due to
> + kernel pages will still be seen. This is not considered a failure and the

> + write will still return success. In this case, it is expected that
> + memory.kmem.usage_in_bytes == memory.usage_in_bytes.
> +
>   About use_hierarchy, see Section 6.
>
> 5.2 stat file
> --
> 1.7.11.4
>
> --
> To unsubscribe from this list: send the line "unsubscribe cgroups" in
> the body of a message to majordomo@vger.kernel.org
> More majordomo info at http://vger.kernel.org/majordomo-info.html

--
Michal Hocko
SUSE Labs

Subject: Re: [PATCH v4 04/14] kmem accounting basic infrastructure
Posted by Glauber Costa on Fri, 12 Oct 2012 07:36:38 GMT
View Forum Message <> Reply to Message

On 10/11/2012 02:11 PM, Michal Hocko wrote:
> On Mon 08-10-12 14:06:10, Glauber Costa wrote:
>> This patch adds the basic infrastructure for the accounting of the slab
>> caches. To control that, the following files are created:
>>
>> * memory.kmem.usage_in_bytes
>> * memory.kmem.limit_in_bytes
>> * memory.kmem.failcnt
>> * memory.kmem.max_usage_in_bytes
>>
>> They have the same meaning of their user memory counterparts. They
>> reflect the state of the "kmem" res_counter.
>>
>> Per cgroup slab memory accounting is not enabled until a limit is set
>
> s/slab/kmem/ right?
>
right.

>> +static int memcg_update_kmem_limit(struct cgroup *cont, u64 val)
>> +{
>> + int ret = -EINVAL;
>> +#ifdef CONFIG_MEMCG_KMEM
>> + struct mem_cgroup *memcg = mem_cgroup_from_cont(cont);
>> + /*

```
>> +  * For simplicity, we won't allow this to be disabled.  It also can't
>> +  * be changed if the cgroup has children already, or if tasks had
>> +  * already joined.
>> +  *
>> +  * If tasks join before we set the limit, a person looking at
>> +  * kmem.usage_in_bytes will have no way to determine when it took
>> +  * place, which makes the value quite meaningless.
>> +  *
>> +  * After it first became limited, changes in the value of the limit are
>> +  * of course permitted.
>> +  *
>> +  * Taking the cgroup_lock is really offensive, but it is so far the only
>> +  * way to guarantee that no children will appear. There are plenty of
>> +  * other offenders, and they should all go away. Fine grained locking
>> +  * is probably the way to go here. When we are fully hierarchical, we
>> +  * can also get rid of the use_hierarchy check.
>> +  */
>> + cgroup_lock();
>> + mutex_lock(&set_limit_mutex);
>> + if (!memcg->kmem_accounted && val != RESOURCE_MAX) {
>
> Just a nit but wouldn't memcg_kmem_is_accounted(memcg) be better than
> directly checking kmem_accounted?
> Besides that I am not sure I fully understand RESOURCE_MAX test. Say I
> want to have kmem accounting for monitoring so I do
> echo -1 > memory.kmem.limit_in_bytes
>
> so you set the value but do not activate it. Isn't this just a reminder
> from the time when the accounting could be deactivated?
>
```

No, not at all.

I see you have talked about that in other e-mails, (I was on sick leave yesterday), so let me consolidate it all here:

What we discussed before, regarding to echo -1 > ... was around the disable code, something that we no longer allow. So now, if you will echo -1 to that file *after* it is limited, you get in track only mode.

But for you to start that, you absolutely have to write something different than -1.

Just one example: libcgroup, regardless of how lame we think it is in this regard, will write to all cgroup files by default when a file is updated. If you haven't written anything, it will still write the same value that the file had before.

This means that an already deployed libcg-managed installation will
suddenly enable kmem for every cgroup. Sure this can be fixed in
userspace, but:

1) There is no reason to break it, if we can
2) It is perfectly reasonable to expect that if you write to a file the
same value that was already there, nothing happens.

I'll update the docs to say that you can just write -1 *after* it is
limited, but i believe enabling it has to be a very clear transition,
for sanity's sake.

```
>> +  if (cgroup_task_count(cont) || (memcg->use_hierarchy &&
>> +      !list_empty(&cont->children))) {
>> +   ret = -EBUSY;
>> +   goto out;
>> +  }
>> +  ret = res_counter_set_limit(&memcg->kmem, val);
>
> VM_BUG_IN(ret) ?
> There shouldn't be any usage when you enable it or something bad is
> going on.
>
```
Good point, this is indeed an impossible scenario I was just being
overcautious about.

Subject: Re: [PATCH v4 06/14] memcg: kmem controller infrastructure
Posted by Glauber Costa on Fri, 12 Oct 2012 07:45:46 GMT

View Forum Message <> Reply to Message

On 10/11/2012 04:42 PM, Michal Hocko wrote:
```
> On Mon 08-10-12 14:06:12, Glauber Costa wrote:
>> This patch introduces infrastructure for tracking kernel memory pages to
>> a given memcg. This will happen whenever the caller includes the flag
>> __GFP_KMEMCG flag, and the task belong to a memcg other than the root.
>>
>> In memcontrol.h those functions are wrapped in inline acessors.  The
>> idea is to later on, patch those with static branches, so we don't incur
>> any overhead when no mem cgroups with limited kmem are being used.
>>
>> Users of this functionality shall interact with the memcg core code
>> through the following functions:
>>
>> memcg_kmem_newpage_charge: will return true if the group can handle the
>>                          allocation. At this point, struct page is not
>>                          yet allocated.
>>
```

```
>> memcg_kmem_commit_charge: will either revert the charge, if struct page
>>                          allocation failed, or embed memcg information
>>                          into page_cgroup.
>>
>> memcg_kmem_uncharge_page: called at free time, will revert the charge.
>>
>> [ v2: improved comments and standardized function names ]
>> [ v3: handle no longer opaque, functions not exported,
>>   even more comments ]
>> [ v4: reworked Used bit handling and surroundings for more clarity ]
>> [ v5: simplified code for kmemcg compiled out and core functions in
>>   memcontrol.c, moved kmem code to the middle to avoid forward decls ]
>>
>> Signed-off-by: Glauber Costa <glommer@parallels.com>
>> CC: Christoph Lameter <cl@linux.com>
>> CC: Pekka Enberg <penberg@cs.helsinki.fi>
>> CC: Michal Hocko <mhocko@suse.cz>
>> CC: Kamezawa Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com>
>> CC: Johannes Weiner <hannes@cmpxchg.org>
>> ---
>>  include/linux/memcontrol.h |  95 +++++++++++++++++++++++++
>>  mm/memcontrol.c            | 173 +++++++++++++++++++++++++++++++++++++++++++++--
>>  2 files changed, 263 insertions(+), 5 deletions(-)
>>
>
> Just a nit. Hmm we are far from being consisten in using vs. not using
> externs in header files for function declarations but I do not see any
> reason why to use them here. Names are just longer without any
> additional value.
>
```

Neither do I.
I don't like externs for functions, I am just using them because they
seem to be used quite extensively around...

```
> [...]
>> +static int memcg_charge_kmem(struct mem_cgroup *memcg, gfp_t gfp, u64 size)
>> +{
>> + struct res_counter *fail_res;
>> + struct mem_cgroup *_memcg;
>> + int ret = 0;
>> + bool may_oom;
>> +
>> + /*
>> +  * Conditions under which we can wait for the oom_killer.
>> +  * __GFP_NORETRY should be masked by __mem_cgroup_try_charge,
>> +  * but there is no harm in being explicit here
>> +  */
```

>> + may_oom = (gfp & __GFP_WAIT) && !(gfp & __GFP_NORETRY);
>
> Well we _have to_ check __GFP_NORETRY here because if we don't then we
> can end up in OOM. mem_cgroup_do_charge returns CHARGE_NOMEM for
> __GFP_NORETRY (without doing any reclaim) and of oom==true we decrement
> oom retries counter and eventually hit OOM killer. So the comment is
> misleading.

I will update. What i understood from your last message is that we don't
really need to, because try_charge will do it.

```
>> +
>> + _memcg = memcg;
>> + ret = __mem_cgroup_try_charge(NULL, gfp, size >> PAGE_SHIFT,
>> +          &_memcg, may_oom);
>> +
>> + if (!ret) {
>> +  ret = res_counter_charge(&memcg->kmem, size, &fail_res);
```
>
> Now that I'm thinking about the charging ordering we should charge the
> kmem first because we would like to hit kmem limit before we hit u+k
> limit, don't we.
> Say that you have kmem limit 10M and the total limit 50M. Current `u'
> would be 40M and this charge would cause kmem to hit the `k' limit. I
> think we should fail to charge kmem before we go to u+k and potentially
> reclaim/oom.
> Or has this been alredy discussed and I just do not remember?
>
This has never been discussed as far as I remember. We charged u first
since day0, and you are so far the first one to raise it...

One of the things in favor of charging 'u' first is that
mem_cgroup_try_charge is already equipped to make a lot of decisions,
like when to allow reclaim, when to bypass charges, and it would be good
if we can reuse all that.

You oom-based argument makes some sense, if all other scenarios are
unchanged by this, I can change it. I will give this some more
consideration.

```
>> +  if (ret) {
>> +   res_counter_uncharge(&memcg->res, size);
>> +   if (do_swap_account)
>> +    res_counter_uncharge(&memcg->memsw, size);
>> +  }
> [...]
>> +bool
>> +__memcg_kmem_newpage_charge(gfp_t gfp, struct mem_cgroup **_memcg, int order)
>> +{
>> + struct mem_cgroup *memcg;
>> + int ret;
>> +
>> + *_memcg = NULL;
>> + memcg = try_get_mem_cgroup_from_mm(current->mm);
>> +
>> + /*
>> +  * very rare case described in mem_cgroup_from_task. Unfortunately there
>> +  * isn't much we can do without complicating this too much, and it would
>> +  * be gfp-dependent anyway. Just let it go
>> +  */
>> + if (unlikely(!memcg))
>> +  return true;
>> +
>> + if (!memcg_can_account_kmem(memcg)) {
>> +  css_put(&memcg->css);
>> +  return true;
>> + }
>> +
>  /*
>   * Keep reference on memcg while the page is charged to prevent
>   * group from vanishing because allocation can outlive their
>   * tasks. The reference is dropped in __memcg_kmem_uncharge_page
>   */
>
> please
```

I can do that, but keep in mind this piece of code is going away soon =)

Subject: Re: [PATCH v4 09/14] memcg: kmem accounting lifecycle management
Posted by Glauber Costa on Fri, 12 Oct 2012 07:47:17 GMT

View Forum Message <> Reply to Message

On 10/11/2012 05:11 PM, Michal Hocko wrote:
> On Mon 08-10-12 14:06:15, Glauber Costa wrote:
>> Because kmem charges can outlive the cgroup, we need to make sure that
>> we won't free the memcg structure while charges are still in flight.
>> For reviewing simplicity, the charge functions will issue
>> mem_cgroup_get() at every charge, and mem_cgroup_put() at every
>> uncharge.
>>
>> This can get expensive, however, and we can do better. mem_cgroup_get()
>> only really needs to be issued once: when the first limit is set. In the
>> same spirit, we only need to issue mem_cgroup_put() when the last charge
>> is gone.
>>

>> We'll need an extra bit in kmem_accounted for that: KMEM_ACCOUNTED_DEAD.
>> it will be set when the cgroup dies, if there are charges in the group.
>> If there aren't, we can proceed right away.
>>
>> Our uncharge function will have to test that bit every time the charges
>> drop to 0. Because that is not the likely output of
>> res_counter_uncharge, this should not impose a big hit on us: it is
>> certainly much better than a reference count decrease at every
>> operation.
>>
>> [v3: merged all lifecycle related patches in one]
>>
>> Signed-off-by: Glauber Costa <glommer@parallels.com>
>> CC: Kamezawa Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com>
>> CC: Christoph Lameter <cl@linux.com>
>> CC: Pekka Enberg <penberg@cs.helsinki.fi>
>> CC: Michal Hocko <mhocko@suse.cz>
>> CC: Johannes Weiner <hannes@cmpxchg.org>
>> CC: Suleiman Souhlal <suleiman@google.com>
>
> OK, I like the optimization. I have just one comment to the
> memcg_kmem_dead naming but other than that
>
> Acked-by: Michal Hocko <mhocko@suse.cz>
>
> [...]
>> +static bool memcg_kmem_dead(struct mem_cgroup *memcg)
>
> The name is tricky because it doesn't tell you that it clears the flag
> which made me scratch my head when reading comment in kmem_cgroup_destroy
>
memcg_kmem_finally_kill_that_bastard() ?

Subject: Re: [PATCH v4 10/14] memcg: use static branches when code not in use
Posted by Glauber Costa on Fri, 12 Oct 2012 07:47:53 GMT

View Forum Message <> Reply to Message

On 10/11/2012 05:40 PM, Michal Hocko wrote:
> On Mon 08-10-12 14:06:16, Glauber Costa wrote:
>> We can use static branches to patch the code in or out when not used.
>>
>> Because the _ACTIVE bit on kmem_accounted is only set after the
>> increment is done, we guarantee that the root memcg will always be
>> selected for kmem charges until all call sites are patched (see
>> memcg_kmem_enabled). This guarantees that no mischarges are applied.
>>
>> static branch decrement happens when the last reference count from the

```
>> kmem accounting in memcg dies. This will only happen when the charges
>> drop down to 0.
>>
>> When that happen, we need to disable the static branch only on those
>> memcgs that enabled it. To achieve this, we would be forced to
>> complicate the code by keeping track of which memcgs were the ones
>> that actually enabled limits, and which ones got it from its parents.
>>
>> It is a lot simpler just to do static_key_slow_inc() on every child
>> that is accounted.
>>
>> [ v4: adapted this patch to the changes in kmem_accounted ]
>>
>> Signed-off-by: Glauber Costa <glommer@parallels.com>
>> CC: Kamezawa Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com>
>> CC: Christoph Lameter <cl@linux.com>
>> CC: Pekka Enberg <penberg@cs.helsinki.fi>
>> CC: Michal Hocko <mhocko@suse.cz>
>> CC: Johannes Weiner <hannes@cmpxchg.org>
>> CC: Suleiman Souhlal <suleiman@google.com>
>
> Looks reasonable to me
> Acked-by: Michal Hocko <mhocko@suse.cz>
>
> Just a little nit.
>
> [...]
>
>> diff --git a/mm/memcontrol.c b/mm/memcontrol.c
>> index 634c7b5..724a08b 100644
>> --- a/mm/memcontrol.c
>> +++ b/mm/memcontrol.c
>> @@ -344,11 +344,15 @@ struct mem_cgroup {
>>  /* internal only representation about the status of kmem accounting. */
>>  enum {
>>   KMEM_ACCOUNTED_ACTIVE = 0, /* accounted by this cgroup itself */
>> + KMEM_ACCOUNTED_ACTIVATED, /* static key enabled. */
>>   KMEM_ACCOUNTED_DEAD, /* dead memcg, pending kmem charges */
>>  };
>>
>> -/* first bit */
>> -#define KMEM_ACCOUNTED_MASK 0x1
>> +/*
>> + * first two bits. We account when limit is on, but only after
>> + * call sites are patched
>> + */
>> +#define KMEM_ACCOUNTED_MASK 0x3
>
```

> The names are long but why not use KMEM_ACCOUNTED_ACTIVE*
> #define KMEM_ACCOUNTED_MASK 1<<KMEM_ACCOUNTED_ACTIVE |
1<<KMEM_ACCOUNTED_ACTIVATED
>
Because the names are long! =)

Subject: Re: [PATCH v4 14/14] Add documentation about the kmem controller
Posted by Glauber Costa on Fri, 12 Oct 2012 07:53:23 GMT

View Forum Message <> Reply to Message

On 10/11/2012 06:35 PM, Michal Hocko wrote:
> On Mon 08-10-12 14:06:20, Glauber Costa wrote:
>> Signed-off-by: Glauber Costa <glommer@parallels.com>
>> ---
>> Documentation/cgroups/memory.txt | 55 +++-
>> 1 file changed, 54 insertions(+), 1 deletion(-)
>>
>> diff --git a/Documentation/cgroups/memory.txt b/Documentation/cgroups/memory.txt
>> index c07f7b4..9b08548 100644
>> --- a/Documentation/cgroups/memory.txt
>> +++ b/Documentation/cgroups/memory.txt
>> @@ -71,6 +71,11 @@ Brief summary of control files.
>>   memory.oom_control   # set/show oom controls.
>>   memory.numa_stat   # show the number of memory usage per numa node
>>
>> + memory.kmem.limit_in_bytes   # set/show hard limit for kernel memory
>> + memory.kmem.usage_in_bytes   # show current kernel memory allocation
>> + memory.kmem.failcnt   # show the number of kernel memory usage hits limits
>> + memory.kmem.max_usage_in_bytes # show max kernel memory usage recorded
>> +
>>   memory.kmem.tcp.limit_in_bytes # set/show hard limit for tcp buf memory
>>   memory.kmem.tcp.usage_in_bytes # show current tcp buf memory allocation
>>   memory.kmem.tcp.failcnt   # show the number of tcp buf memory usage hits limits
>> @@ -268,20 +273,62 @@ the amount of kernel memory used by the system. Kernel memory
is fundamentally
>> different than user memory, since it can't be swapped out, which makes it
>> possible to DoS the system by consuming too much of this precious resource.
>>
>> +Kernel memory won't be accounted at all until it is limited. This allows for
>
> until limit on a group is set.
>
ok.

>> +existing setups to continue working without disruption. Note that it is
>> +possible to account it without an effective limit by setting the limits
>> +to a very high number (like RESOURCE_MAX -1page).

>
> I have brought that up in an earlier patch already. Why not just do echo
> -1 (which translates to RESOURCE_MAX internally) and be done with that.
> RESOURCE_MAX-1 sounds quite inconvenient.
>

For the case that you are limited already, and then want to unlimit,
keeping the accounting, yes, it makes sense.

>> The limit cannot be set
>> +if the cgroup have children, or if there are already tasks in the cgroup.
>
> I would start by stating that if children are accounted automatically if
> their parent is accounted already and there is no need to set a limit to
> enforce that. In fact the limit cannot be set if ....
>

ok.

>> +
>> +After a controller is first limited, it will be kept being accounted until it
>
> group is limited not the controller.
>

true, thanks.

>> +
>> Kernel memory limits are not imposed for the root cgroup. Usage for the root
>> -cgroup may or may not be accounted.
>> +cgroup may or may not be accounted. The memory used is accumulated into
>> +memory.kmem.usage_in_bytes, or in a separate counter when it makes sense.
>
> Which separate counter? Is this about tcp kmem?
>

So far, yes, this is the only case that makes sense, and the fewer the
better. In any case it exists, and I wanted to be generic.

>> +The main "kmem" counter is fed into the main counter, so kmem charges will
>> +also be visible from the user counter.
>>
>> Currently no soft limit is implemented for kernel memory. It is future work
>> to trigger slab reclaim when those limits are reached.
>>
>> 2.7.1 Current Kernel Memory resources accounted
>>

```
>> +* stack pages: every process consumes some stack pages. By accounting into
>> +kernel memory, we prevent new processes from being created when the kernel
>> +memory usage is too high.
>> +
>>  * sockets memory pressure: some sockets protocols have memory pressure
>>  thresholds. The Memory Controller allows them to be controlled individually
>>  per cgroup, instead of globally.
>>
>>  * tcp memory pressure: sockets memory pressure for the tcp protocol.
>>
>> +2.7.3 Common use cases
>> +
>> +Because the "kmem" counter is fed to the main user counter, kernel memory can
>> +never be limited completely independently of user memory. Say "U" is the user
>> +limit, and "K" the kernel limit. There are three possible ways limits can be
>> +set:
>> +
>> +    U != 0, K = 0:
>
> K is not 0 it is unaccounted (disabled)
>
>> +    This is the standard memcg limitation mechanism already present before kmem
>> +    accounting. Kernel memory is completely ignored.
>> +
>> +    U,K != 0, K < U:
>
> I would keep K < U
>> +    Kernel memory is effectively set as a percentage of the user memory. This
>
> not a percentage it is subset of the user memory
>
Well, this is semantics. I can change, but for me it makes a lot of
sense to think of it in terms of a percentage, because it is easy to
administer. You don't actually write a percentage, which I tried to
clarify by using the term "effective set as a percentage".


>> +    setup is useful in deployments where the total amount of memory per-cgroup
>> +    is overcommited. Overcommiting kernel memory limits is definitely not
>> +    recommended, since the box can still run out of non-reclaimable memory.
>> +    In this case, the admin could set up K so that the sum of all groups is
>> +    never greater than the total memory, and freely set U at the cost of his
>> +    QoS.
>> +
>> +    U,K != 0, K >= U:
>> +    Since kmem charges will also be fed to the user counter, this setup gives
>> +    the admin a unified view of memory. Reclaim will be triggered for the cgroup
>> +    for both kinds of memory.
```

>
> This is also useful for tracking kernel memory allocation.
>
ok.

Subject: Re: [PATCH v4 04/14] kmem accounting basic infrastructure
Posted by Michal Hocko on Fri, 12 Oct 2012 08:27:28 GMT

View Forum Message <> Reply to Message

On Fri 12-10-12 11:36:38, Glauber Costa wrote:
> On 10/11/2012 02:11 PM, Michal Hocko wrote:
> > On Mon 08-10-12 14:06:10, Glauber Costa wrote:
[...]
> >> + if (!memcg->kmem_accounted && val != RESOURCE_MAX) {
> >
> > Just a nit but wouldn't memcg_kmem_is_accounted(memcg) be better than
> > directly checking kmem_accounted?
> > Besides that I am not sure I fully understand RESOURCE_MAX test. Say I
> > want to have kmem accounting for monitoring so I do
> > echo -1 > memory.kmem.limit_in_bytes
> >
> > so you set the value but do not activate it. Isn't this just a reminder
> > from the time when the accounting could be deactivated?
> >
>
> No, not at all.
>
> I see you have talked about that in other e-mails, (I was on sick leave
> yesterday), so let me consolidate it all here:
>
> What we discussed before, regarding to echo -1 > ... was around the
> disable code, something that we no longer allow. So now, if you will
> echo -1 to that file *after* it is limited, you get in track only mode.
>
> But for you to start that, you absolutely have to write something
> different than -1.
>
> Just one example: libcgroup, regardless of how lame we think it is in
> this regard, will write to all cgroup files by default when a file is
> updated. If you haven't written anything, it will still write the same
> value that the file had before.

Ohh, I wasn't aware of that and it sounds pretty lame.

> This means that an already deployed libcg-managed installation will
> suddenly enable kmem for every cgroup. Sure this can be fixed in
> userspace, but:

>
> 1) There is no reason to break it, if we can

You are right

> 2) It is perfectly reasonable to expect that if you write to a file the
> same value that was already there, nothing happens.

Fair enough

--
Michal Hocko
SUSE Labs

Subject: Re: [PATCH v4 06/14] memcg: kmem controller infrastructure
Posted by Michal Hocko on Fri, 12 Oct 2012 08:39:45 GMT

View Forum Message <> Reply to Message

On Fri 12-10-12 11:45:46, Glauber Costa wrote:
> On 10/11/2012 04:42 PM, Michal Hocko wrote:
> > On Mon 08-10-12 14:06:12, Glauber Costa wrote:
[...]
```
> >> + /*
> >> +  * Conditions under which we can wait for the oom_killer.
> >> +  * __GFP_NORETRY should be masked by __mem_cgroup_try_charge,
> >> +  * but there is no harm in being explicit here
> >> +  */
> >> + may_oom = (gfp & __GFP_WAIT) && !(gfp & __GFP_NORETRY);
```
> >
> > Well we _have to_ check __GFP_NORETRY here because if we don't then we
> > can end up in OOM. mem_cgroup_do_charge returns CHARGE_NOMEM for
> > __GFP_NORETRY (without doing any reclaim) and of oom==true we decrement
> > oom retries counter and eventually hit OOM killer. So the comment is
> > misleading.
>
> I will update. What i understood from your last message is that we don't
> really need to, because try_charge will do it.

IIRC I just said it couldn't happen before because migration doesn't go
through charge and thp disable oom by default.

```
> >> +
> >> + _memcg = memcg;
> >> + ret = __mem_cgroup_try_charge(NULL, gfp, size >> PAGE_SHIFT,
> >> +         &_memcg, may_oom);
> >> +
> >> + if (!ret) {
```

> >> + ret = res_counter_charge(&memcg->kmem, size, &fail_res);
> >
> > Now that I'm thinking about the charging ordering we should charge the
> > kmem first because we would like to hit kmem limit before we hit u+k
> > limit, don't we.
> > Say that you have kmem limit 10M and the total limit 50M. Current `u'
> > would be 40M and this charge would cause kmem to hit the `k' limit. I
> > think we should fail to charge kmem before we go to u+k and potentially
> > reclaim/oom.
> > Or has this been alredy discussed and I just do not remember?
> >
> This has never been discussed as far as I remember. We charged u first
> since day0, and you are so far the first one to raise it...
>
> One of the things in favor of charging 'u' first is that
> mem_cgroup_try_charge is already equipped to make a lot of decisions,
> like when to allow reclaim, when to bypass charges, and it would be good
> if we can reuse all that.

Hmm, I think that we should prevent from those decisions if kmem charge
would fail anyway (especially now when we do not have targeted slab
reclaim).

> You oom-based argument makes some sense, if all other scenarios are
> unchanged by this, I can change it. I will give this some more
> consideration.
>
[...]
> > /*
> >   * Keep reference on memcg while the page is charged to prevent
> >   * group from vanishing because allocation can outlive their
> >   * tasks. The reference is dropped in __memcg_kmem_uncharge_page
> >   */
> >
> > please
>
> I can do that, but keep in mind this piece of code is going away soon =)

Yes I have noticed that and replied to myself that it is not necessary.

--
Michal Hocko
SUSE Labs

Subject: Re: [PATCH v4 09/14] memcg: kmem accounting lifecycle management
Posted by Michal Hocko on Fri, 12 Oct 2012 08:41:01 GMT

On Fri 12-10-12 11:47:17, Glauber Costa wrote:
> On 10/11/2012 05:11 PM, Michal Hocko wrote:
> > On Mon 08-10-12 14:06:15, Glauber Costa wrote:
> >> Because kmem charges can outlive the cgroup, we need to make sure that
> >> we won't free the memcg structure while charges are still in flight.
> >> For reviewing simplicity, the charge functions will issue
> >> mem_cgroup_get() at every charge, and mem_cgroup_put() at every
> >> uncharge.
> >>
> >> This can get expensive, however, and we can do better. mem_cgroup_get()
> >> only really needs to be issued once: when the first limit is set. In the
> >> same spirit, we only need to issue mem_cgroup_put() when the last charge
> >> is gone.
> >>
> >> We'll need an extra bit in kmem_accounted for that: KMEM_ACCOUNTED_DEAD.
> >> it will be set when the cgroup dies, if there are charges in the group.
> >> If there aren't, we can proceed right away.
> >>
> >> Our uncharge function will have to test that bit every time the charges
> >> drop to 0. Because that is not the likely output of
> >> res_counter_uncharge, this should not impose a big hit on us: it is
> >> certainly much better than a reference count decrease at every
> >> operation.
> >>
> >> [v3: merged all lifecycle related patches in one]
> >>
> >> Signed-off-by: Glauber Costa <glommer@parallels.com>
> >> CC: Kamezawa Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com>
> >> CC: Christoph Lameter <cl@linux.com>
> >> CC: Pekka Enberg <penberg@cs.helsinki.fi>
> >> CC: Michal Hocko <mhocko@suse.cz>
> >> CC: Johannes Weiner <hannes@cmpxchg.org>
> >> CC: Suleiman Souhlal <suleiman@google.com>
> >
> > OK, I like the optimization. I have just one comment to the
> > memcg_kmem_dead naming but other than that
> >
> > Acked-by: Michal Hocko <mhocko@suse.cz>
> >
> > [...]
> >> +static bool memcg_kmem_dead(struct mem_cgroup *memcg)
> >
> > The name is tricky because it doesn't tell you that it clears the flag
> > which made me scratch my head when reading comment in kmem_cgroup_destroy
> >
> memcg_kmem_finally_kill_that_bastard() ?

memcg_kmem_test_and_clear_dead? I know long but at least clear that the
flag is cleared. Or just open code it.
--
Michal Hocko
SUSE Labs

Subject: Re: [PATCH v4 14/14] Add documentation about the kmem controller
Posted by Michal Hocko on Fri, 12 Oct 2012 08:44:32 GMT

View Forum Message <> Reply to Message

On Fri 12-10-12 11:53:23, Glauber Costa wrote:
> On 10/11/2012 06:35 PM, Michal Hocko wrote:
> > On Mon 08-10-12 14:06:20, Glauber Costa wrote:
[...]
> >> Kernel memory limits are not imposed for the root cgroup. Usage for the root
> >> -cgroup may or may not be accounted.
> >> +cgroup may or may not be accounted. The memory used is accumulated into
> >> +memory.kmem.usage_in_bytes, or in a separate counter when it makes sense.
> >
> > Which separate counter? Is this about tcp kmem?
> >
>
> So far, yes, this is the only case that makes sense, and the fewer the
> better. In any case it exists, and I wanted to be generic.

Add (currently tcp) or something similar

[...]
> >> +   Kernel memory is effectively set as a percentage of the user memory. This
> >
> > not a percentage it is subset of the user memory
> >
> Well, this is semantics. I can change, but for me it makes a lot of
> sense to think of it in terms of a percentage, because it is easy to
> administer. You don't actually write a percentage, which I tried to
> clarify by using the term "effective set as a percentage".

I can still see somebody reading this and wondering why echo 50 > ...limit
didn't set a percentage...

--
Michal Hocko
SUSE Labs

Subject: Re: [PATCH v4 06/14] memcg: kmem controller infrastructure

Posted by Glauber Costa on Fri, 12 Oct 2012 08:44:57 GMT
View Forum Message <> Reply to Message

On 10/12/2012 12:39 PM, Michal Hocko wrote:
> On Fri 12-10-12 11:45:46, Glauber Costa wrote:
>> On 10/11/2012 04:42 PM, Michal Hocko wrote:
>>> On Mon 08-10-12 14:06:12, Glauber Costa wrote:
> [...]
>>>> + /*
>>>> + * Conditions under which we can wait for the oom_killer.
>>>> + * __GFP_NORETRY should be masked by __mem_cgroup_try_charge,
>>>> + * but there is no harm in being explicit here
>>>> + */
>>>> + may_oom = (gfp & __GFP_WAIT) && !(gfp & __GFP_NORETRY);
>>>
>>> Well we _have to_ check __GFP_NORETRY here because if we don't then we
>>> can end up in OOM. mem_cgroup_do_charge returns CHARGE_NOMEM for
>>> __GFP_NORETRY (without doing any reclaim) and of oom==true we decrement
>>> oom retries counter and eventually hit OOM killer. So the comment is
>>> misleading.
>>
>> I will update. What i understood from your last message is that we don't
>> really need to, because try_charge will do it.
>
> IIRC I just said it couldn't happen before because migration doesn't go
> through charge and thp disable oom by default.
>

I had it changed to:

```
        /*
         * Conditions under which we can wait for the oom_killer.
         * We have to be able to wait, but also, if we can't retry,
         * we obviously shouldn't go mess with oom.
         */
        may_oom = (gfp & __GFP_WAIT) && !(gfp & __GFP_NORETRY);
```

>>>> +
>>>> + _memcg = memcg;
>>>> + ret = __mem_cgroup_try_charge(NULL, gfp, size >> PAGE_SHIFT,
>>>> +   &_memcg, may_oom);
>>>> +
>>>> + if (!ret) {
>>>> + ret = res_counter_charge(&memcg->kmem, size, &fail_res);
>>>
>>> Now that I'm thinking about the charging ordering we should charge the
>>> kmem first because we would like to hit kmem limit before we hit u+k
>>> limit, don't we.
>>> Say that you have kmem limit 10M and the total limit 50M. Current `u'

>>> would be 40M and this charge would cause kmem to hit the `k' limit. I
>>> think we should fail to charge kmem before we go to u+k and potentially
>>> reclaim/oom.
>>> Or has this been alredy discussed and I just do not remember?
>>>
>> This has never been discussed as far as I remember. We charged u first
>> since day0, and you are so far the first one to raise it...
>>
>> One of the things in favor of charging 'u' first is that
>> mem_cgroup_try_charge is already equipped to make a lot of decisions,
>> like when to allow reclaim, when to bypass charges, and it would be good
>> if we can reuse all that.
>
> Hmm, I think that we should prevent from those decisions if kmem charge
> would fail anyway (especially now when we do not have targeted slab
> reclaim).
>

Let's revisit this discussion when we do have targeted reclaim. For now,
I'll agree that charging kmem first would be acceptable.

This will only make a difference when K < U anyway.

Subject: Re: [PATCH v4 06/14] memcg: kmem controller infrastructure
Posted by Michal Hocko on Fri, 12 Oct 2012 08:57:40 GMT
View Forum Message <> Reply to Message

On Fri 12-10-12 12:44:57, Glauber Costa wrote:
> On 10/12/2012 12:39 PM, Michal Hocko wrote:
> > On Fri 12-10-12 11:45:46, Glauber Costa wrote:
> >> On 10/11/2012 04:42 PM, Michal Hocko wrote:
> >>> On Mon 08-10-12 14:06:12, Glauber Costa wrote:
> > [...]
> >>>> + /*
> >>>> + * Conditions under which we can wait for the oom_killer.
> >>>> + * __GFP_NORETRY should be masked by __mem_cgroup_try_charge,
> >>>> + * but there is no harm in being explicit here
> >>>> + */
> >>>> + may_oom = (gfp & __GFP_WAIT) && !(gfp & __GFP_NORETRY);
> >>>
> >>> Well we _have to_ check __GFP_NORETRY here because if we don't then we
> >>> can end up in OOM. mem_cgroup_do_charge returns CHARGE_NOMEM for
> >>> __GFP_NORETRY (without doing any reclaim) and of oom==true we decrement
> >>> oom retries counter and eventually hit OOM killer. So the comment is
> >>> misleading.
> >>
> >> I will update. What i understood from your last message is that we don't

> >> really need to, because try_charge will do it.
> >
> > IIRC I just said it couldn't happen before because migration doesn't go
> > through charge and thp disable oom by default.
> >
>
> I had it changed to:
>
>   /*
>   * Conditions under which we can wait for the oom_killer.
>   * We have to be able to wait, but also, if we can't retry,
>   * we obviously shouldn't go mess with oom.
>   */
>   may_oom = (gfp & __GFP_WAIT) && !(gfp & __GFP_NORETRY);

OK

>
> >>>> +
> >>>> + _memcg = memcg;
> >>>> + ret = __mem_cgroup_try_charge(NULL, gfp, size >> PAGE_SHIFT,
> >>>> +   &_memcg, may_oom);
> >>>> +
> >>>> + if (!ret) {
> >>>> + ret = res_counter_charge(&memcg->kmem, size, &fail_res);
> >>>
> >>> Now that I'm thinking about the charging ordering we should charge the
> >>> kmem first because we would like to hit kmem limit before we hit u+k
> >>> limit, don't we.
> >>> Say that you have kmem limit 10M and the total limit 50M. Current `u'
> >>> would be 40M and this charge would cause kmem to hit the `k' limit. I
> >>> think we should fail to charge kmem before we go to u+k and potentially
> >>> reclaim/oom.
> >>> Or has this been alredy discussed and I just do not remember?
> >>>
> >> This has never been discussed as far as I remember. We charged u first
> >> since day0, and you are so far the first one to raise it...
> >>
> >> One of the things in favor of charging 'u' first is that
> >> mem_cgroup_try_charge is already equipped to make a lot of decisions,
> >> like when to allow reclaim, when to bypass charges, and it would be good
> >> if we can reuse all that.
> >
> > Hmm, I think that we should prevent from those decisions if kmem charge
> > would fail anyway (especially now when we do not have targeted slab
> > reclaim).
> >
>

> Let's revisit this discussion when we do have targeted reclaim. For now,
> I'll agree that charging kmem first would be acceptable.
>
> This will only make a difference when K < U anyway.

Yes and it should work as advertised (aka hit the k limit first).

You can stick my Acked-by then.
--
Michal Hocko
SUSE Labs

Subject: Re: [PATCH v4 06/14] memcg: kmem controller infrastructure
Posted by Glauber Costa on Fri, 12 Oct 2012 09:13:04 GMT

View Forum Message <> Reply to Message

On 10/12/2012 12:57 PM, Michal Hocko wrote:
> On Fri 12-10-12 12:44:57, Glauber Costa wrote:
>> On 10/12/2012 12:39 PM, Michal Hocko wrote:
>>> On Fri 12-10-12 11:45:46, Glauber Costa wrote:
>>>> On 10/11/2012 04:42 PM, Michal Hocko wrote:
>>>>> On Mon 08-10-12 14:06:12, Glauber Costa wrote:
>>> [...]
>>>>>> + /*
>>>>>> + * Conditions under which we can wait for the oom_killer.
>>>>>> + * __GFP_NORETRY should be masked by __mem_cgroup_try_charge,
>>>>>> + * but there is no harm in being explicit here
>>>>>> + */
>>>>>> + may_oom = (gfp & __GFP_WAIT) && !(gfp & __GFP_NORETRY);
>>>>>
>>>>> Well we _have to_ check __GFP_NORETRY here because if we don't then we
>>>>> can end up in OOM. mem_cgroup_do_charge returns CHARGE_NOMEM for
>>>>> __GFP_NORETRY (without doing any reclaim) and of oom==true we decrement
>>>>> oom retries counter and eventually hit OOM killer. So the comment is
>>>>> misleading.
>>>>
>>>> I will update. What i understood from your last message is that we don't
>>>> really need to, because try_charge will do it.
>>>
>>> IIRC I just said it couldn't happen before because migration doesn't go
>>> through charge and thp disable oom by default.
>>>
>>
>> I had it changed to:
>>
>>   /*
>>   * Conditions under which we can wait for the oom_killer.

```
>>         * We have to be able to wait, but also, if we can't retry,
>>         * we obviously shouldn't go mess with oom.
>>         */
>>        may_oom = (gfp & __GFP_WAIT) && !(gfp & __GFP_NORETRY);
>
> OK
>
>>
>>>>>> +
>>>>>> + _memcg = memcg;
>>>>>> + ret = __mem_cgroup_try_charge(NULL, gfp, size >> PAGE_SHIFT,
>>>>>> +         &_memcg, may_oom);
>>>>>> +
>>>>>> + if (!ret) {
>>>>>> +  ret = res_counter_charge(&memcg->kmem, size, &fail_res);
>>>>>
>>>>> Now that I'm thinking about the charging ordering we should charge the
>>>>> kmem first because we would like to hit kmem limit before we hit u+k
>>>>> limit, don't we.
>>>>> Say that you have kmem limit 10M and the total limit 50M. Current `u'
>>>>> would be 40M and this charge would cause kmem to hit the `k' limit. I
>>>>> think we should fail to charge kmem before we go to u+k and potentially
>>>>> reclaim/oom.
>>>>> Or has this been alredy discussed and I just do not remember?
>>>>>
>>>> This has never been discussed as far as I remember. We charged u first
>>>> since day0, and you are so far the first one to raise it...
>>>>
>>>> One of the things in favor of charging 'u' first is that
>>>> mem_cgroup_try_charge is already equipped to make a lot of decisions,
>>>> like when to allow reclaim, when to bypass charges, and it would be good
>>>> if we can reuse all that.
>>>
>>> Hmm, I think that we should prevent from those decisions if kmem charge
>>> would fail anyway (especially now when we do not have targeted slab
>>> reclaim).
>>>
>>
>> Let's revisit this discussion when we do have targeted reclaim. For now,
>> I'll agree that charging kmem first would be acceptable.
>>
>> This will only make a difference when K < U anyway.
>
> Yes and it should work as advertised (aka hit the k limit first).
>
```
Just so we don't ping-pong in another submission:

I changed memcontrol.h's memcg_kmem_newpage_charge to include:

```
        /* If the test is dying, just let it go. */
        if (unlikely(test_thread_flag(TIF_MEMDIE)
                     || fatal_signal_pending(current)))
                return true;
```

I'm also attaching the proposed code in memcontrol.c

File Attachments

1) chch.patch, downloaded 401 times

Subject: Re: [PATCH v4 06/14] memcg: kmem controller infrastructure
Posted by Michal Hocko on Fri, 12 Oct 2012 09:47:26 GMT
View Forum Message <> Reply to Message

On Fri 12-10-12 13:13:04, Glauber Costa wrote:
[...]

```
> Just so we don't ping-pong in another submission:
>
> I changed memcontrol.h's memcg_kmem_newpage_charge to include:
>
>         /* If the test is dying, just let it go. */
>         if (unlikely(test_thread_flag(TIF_MEMDIE)
>                      || fatal_signal_pending(current)))
>                 return true;
```

OK

```
> diff --git a/mm/memcontrol.c b/mm/memcontrol.c
> index c32aaaf..72cf189 100644
> --- a/mm/memcontrol.c
> +++ b/mm/memcontrol.c
> +static int memcg_charge_kmem(struct mem_cgroup *memcg, gfp_t gfp, u64 size)
> +{
> + struct res_counter *fail_res;
> + struct mem_cgroup *_memcg;
> + int ret = 0;
> + bool may_oom;
> +
> + ret = res_counter_charge(&memcg->kmem, size, &fail_res);
> + if (ret)
> +  return ret;
> +
> + /*
> +  * Conditions under which we can wait for the oom_killer.
> +  * We have to be able to wait, but also, if we can't retry,
> +  * we obviously shouldn't go mess with oom.
> +  */
> + may_oom = (gfp & __GFP_WAIT) && !(gfp & __GFP_NORETRY);
> +
> + _memcg = memcg;
> + ret = __mem_cgroup_try_charge(NULL, gfp, size >> PAGE_SHIFT,
> +          &_memcg, may_oom);
> +
> + if (ret == -EINTR)  {
> +  /*
> +   * __mem_cgroup_try_charge() chosed to bypass to root due to
> +   * OOM kill or fatal signal.  Since our only options are to
> +   * either fail the allocation or charge it to this cgroup, do
> +   * it as a temporary condition. But we can't fail. From a
> +   * kmem/slab perspective, the cache has already been selected,
> +   * by mem_cgroup_get_kmem_cache(), so it is too late to change
> +   * our minds. This condition will only trigger if the task
> +   * entered memcg_charge_kmem in a sane state, but was
> +   * OOM-killed.  during __mem_cgroup_try_charge. Tasks that are
> +   * already dying when the allocation triggers should have been
> +   * already directed to the root cgroup.
> +   */
> +  res_counter_charge_nofail(&memcg->res, size, &fail_res);
> +  if (do_swap_account)
> +   res_counter_charge_nofail(&memcg->memsw, size,
> +        &fail_res);
> +  ret = 0;
> + } else if (ret)
> +  res_counter_uncharge(&memcg->kmem, size);
> +
> + return ret;
> +}
```

OK

--
Michal Hocko
SUSE Labs

Subject: Re: [PATCH v4 02/14] memcg: Reclaim when more than one page needed.

Posted by KAMEZAWA Hiroyuki on Tue, 16 Oct 2012 03:22:34 GMT

View Forum Message <> Reply to Message

(2012/10/08 19:06), Glauber Costa wrote:
> From: Suleiman Souhlal <ssouhlal@FreeBSD.org>
>
> mem_cgroup_do_charge() was written before kmem accounting, and expects

> three cases: being called for 1 page, being called for a stock of 32
> pages, or being called for a hugepage. If we call for 2 or 3 pages (and
> both the stack and several slabs used in process creation are such, at
> least with the debug options I had), it assumed it's being called for
> stock and just retried without reclaiming.
>
> Fix that by passing down a minsize argument in addition to the csize.
>
> And what to do about that (csize == PAGE_SIZE && ret) retry? If it's
> needed at all (and presumably is since it's there, perhaps to handle
> races), then it should be extended to more than PAGE_SIZE, yet how far?
> And should there be a retry count limit, of what? For now retry up to
> COSTLY_ORDER (as page_alloc.c does) and make sure not to do it if
> __GFP_NORETRY.
>
> [v4: fixed nr pages calculation pointed out by Christoph Lameter]
>
> Signed-off-by: Suleiman Souhlal <suleiman@google.com>
> Signed-off-by: Glauber Costa <glommer@parallels.com>
> Reviewed-by: Kamezawa Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com>
> Acked-by: Michal Hocko <mhocko@suse.cz>
> Acked-by: Johannes Weiner <hannes@cmpxchg.org>

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

Subject: Re: [PATCH v4 06/14] memcg: kmem controller infrastructure
Posted by KAMEZAWA Hiroyuki on Tue, 16 Oct 2012 08:00:03 GMT
View Forum Message <> Reply to Message

(2012/10/12 18:13), Glauber Costa wrote:
> On 10/12/2012 12:57 PM, Michal Hocko wrote:
>> On Fri 12-10-12 12:44:57, Glauber Costa wrote:
>>> On 10/12/2012 12:39 PM, Michal Hocko wrote:
>>>> On Fri 12-10-12 11:45:46, Glauber Costa wrote:
>>>>> On 10/11/2012 04:42 PM, Michal Hocko wrote:
>>>>>> On Mon 08-10-12 14:06:12, Glauber Costa wrote:
>>>> [...]
>>>>>>> + /*
>>>>>>> + * Conditions under which we can wait for the oom_killer.
>>>>>>> + * __GFP_NORETRY should be masked by __mem_cgroup_try_charge,
>>>>>>> + * but there is no harm in being explicit here
>>>>>>> + */
>>>>>>> + may_oom = (gfp & __GFP_WAIT) && !(gfp & __GFP_NORETRY);
>>>>>>
>>>>>> Well we _have to_ check __GFP_NORETRY here because if we don't then we
>>>>>> can end up in OOM. mem_cgroup_do_charge returns CHARGE_NOMEM for
>>>>>> __GFP_NORETRY (without doing any reclaim) and of oom==true we decrement

```
>>>>>> oom retries counter and eventually hit OOM killer. So the comment is
>>>>>> misleading.
>>>>>
>>>>> I will update. What i understood from your last message is that we don't
>>>>> really need to, because try_charge will do it.
>>>>
>>>> IIRC I just said it couldn't happen before because migration doesn't go
>>>> through charge and thp disable oom by default.
>>>>
>>>
>>> I had it changed to:
>>>
>>>         /*
>>>          * Conditions under which we can wait for the oom_killer.
>>>          * We have to be able to wait, but also, if we can't retry,
>>>          * we obviously shouldn't go mess with oom.
>>>          */
>>>         may_oom = (gfp & __GFP_WAIT) && !(gfp & __GFP_NORETRY);
>>
>> OK
>>
>>>
>>>>>>> +
>>>>>>> + _memcg = memcg;
>>>>>>> + ret = __mem_cgroup_try_charge(NULL, gfp, size >> PAGE_SHIFT,
>>>>>>> +         &_memcg, may_oom);
>>>>>>> +
>>>>>>> + if (!ret) {
>>>>>>> +  ret = res_counter_charge(&memcg->kmem, size, &fail_res);
>>>>>>
>>>>>> Now that I'm thinking about the charging ordering we should charge the
>>>>>> kmem first because we would like to hit kmem limit before we hit u+k
>>>>>> limit, don't we.
>>>>>> Say that you have kmem limit 10M and the total limit 50M. Current `u'
>>>>>> would be 40M and this charge would cause kmem to hit the `k' limit. I
>>>>>> think we should fail to charge kmem before we go to u+k and potentially
>>>>>> reclaim/oom.
>>>>>> Or has this been alredy discussed and I just do not remember?
>>>>>>
>>>>> This has never been discussed as far as I remember. We charged u first
>>>>> since day0, and you are so far the first one to raise it...
>>>>>
>>>>> One of the things in favor of charging 'u' first is that
>>>>> mem_cgroup_try_charge is already equipped to make a lot of decisions,
>>>>> like when to allow reclaim, when to bypass charges, and it would be good
>>>>> if we can reuse all that.
>>>>
>>>> Hmm, I think that we should prevent from those decisions if kmem charge
```

```
>>>> would fail anyway (especially now when we do not have targeted slab
>>>> reclaim).
>>>>
>>>
>>> Let's revisit this discussion when we do have targeted reclaim. For now,
>>> I'll agree that charging kmem first would be acceptable.
>>>
>>> This will only make a difference when K < U anyway.
>>
>> Yes and it should work as advertised (aka hit the k limit first).
>>
> Just so we don't ping-pong in another submission:
>
> I changed memcontrol.h's memcg_kmem_newpage_charge to include:
>
>         /* If the test is dying, just let it go. */
>         if (unlikely(test_thread_flag(TIF_MEMDIE)
>                         || fatal_signal_pending(current)))
>                 return true;
>
>
> I'm also attaching the proposed code in memcontrol.c
>
> +static int memcg_charge_kmem(struct mem_cgroup *memcg, gfp_t gfp, u64 size)
> +{
> + struct res_counter *fail_res;
> + struct mem_cgroup *_memcg;
> + int ret = 0;
> + bool may_oom;
> +
> + ret = res_counter_charge(&memcg->kmem, size, &fail_res);
> + if (ret)
> +  return ret;
> +
> + /*
> +  * Conditions under which we can wait for the oom_killer.
> +  * We have to be able to wait, but also, if we can't retry,
> +  * we obviously shouldn't go mess with oom.
> +  */
> + may_oom = (gfp & __GFP_WAIT) && !(gfp & __GFP_NORETRY);
> +
> + _memcg = memcg;
> + ret = __mem_cgroup_try_charge(NULL, gfp, size >> PAGE_SHIFT,
> +         &_memcg, may_oom);
> +
> + if (ret == -EINTR)  {
> +  /*
> +   * __mem_cgroup_try_charge() chosed to bypass to root due to
> +   * OOM kill or fatal signal.  Since our only options are to
> +   * either fail the allocation or charge it to this cgroup, do
> +   * it as a temporary condition. But we can't fail. From a
> +   * kmem/slab perspective, the cache has already been selected,
> +   * by mem_cgroup_get_kmem_cache(), so it is too late to change
> +   * our minds. This condition will only trigger if the task
> +   * entered memcg_charge_kmem in a sane state, but was
> +   * OOM-killed.  during __mem_cgroup_try_charge. Tasks that are
> +   * already dying when the allocation triggers should have been
> +   * already directed to the root cgroup.
> +   */
> +  res_counter_charge_nofail(&memcg->res, size, &fail_res);
> +  if (do_swap_account)
> +   res_counter_charge_nofail(&memcg->memsw, size,
> +        &fail_res);
> +  ret = 0;
> + } else if (ret)
> +  res_counter_uncharge(&memcg->kmem, size);
> +
> + return ret;
> +}
```

seems ok to me. but we'll need a patch to hide the usage > limit situation from users.

Thanks,
-Kame

Subject: Re: [PATCH v4 08/14] res_counter: return amount of charges after res_counter_uncharge
Posted by KAMEZAWA Hiroyuki on Tue, 16 Oct 2012 08:20:34 GMT

View Forum Message <> Reply to Message

(2012/10/08 19:06), Glauber Costa wrote:
> It is useful to know how many charges are still left after a call to
> res_counter_uncharge. While it is possible to issue a res_counter_read
> after uncharge, this can be racy.
>
> If we need, for instance, to take some action when the counters drop
> down to 0, only one of the callers should see it. This is the same
> semantics as the atomic variables in the kernel.
>
> Since the current return value is void, we don't need to worry about
> anything breaking due to this change: nobody relied on that, and only
> users appearing from now on will be checking this value.
>
> Signed-off-by: Glauber Costa <glommer@parallels.com>

> CC: Michal Hocko <mhocko@suse.cz>
> CC: Johannes Weiner <hannes@cmpxchg.org>
> CC: Suleiman Souhlal <suleiman@google.com>
> CC: Kamezawa Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com>
> ---
```
>   Documentation/cgroups/resource_counter.txt |  7 ++++---
>   include/linux/res_counter.h                | 12 +++++++-----
>   kernel/res_counter.c                       | 20 +++++++++++++-------
>   3 files changed, 24 insertions(+), 15 deletions(-)
```

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

Subject: Re: [PATCH v4 09/14] memcg: kmem accounting lifecycle management
Posted by KAMEZAWA Hiroyuki on Tue, 16 Oct 2012 08:41:32 GMT
View Forum Message <> Reply to Message

(2012/10/12 17:41), Michal Hocko wrote:
> On Fri 12-10-12 11:47:17, Glauber Costa wrote:
>> On 10/11/2012 05:11 PM, Michal Hocko wrote:
>>> On Mon 08-10-12 14:06:15, Glauber Costa wrote:
>>>> Because kmem charges can outlive the cgroup, we need to make sure that
>>>> we won't free the memcg structure while charges are still in flight.
>>>> For reviewing simplicity, the charge functions will issue
>>>> mem_cgroup_get() at every charge, and mem_cgroup_put() at every
>>>> uncharge.
>>>>
>>>> This can get expensive, however, and we can do better. mem_cgroup_get()
>>>> only really needs to be issued once: when the first limit is set. In the
>>>> same spirit, we only need to issue mem_cgroup_put() when the last charge
>>>> is gone.
>>>>
>>>> We'll need an extra bit in kmem_accounted for that: KMEM_ACCOUNTED_DEAD.
>>>> it will be set when the cgroup dies, if there are charges in the group.
>>>> If there aren't, we can proceed right away.
>>>>
>>>> Our uncharge function will have to test that bit every time the charges
>>>> drop to 0. Because that is not the likely output of
>>>> res_counter_uncharge, this should not impose a big hit on us: it is
>>>> certainly much better than a reference count decrease at every
>>>> operation.
>>>>
>>>> [v3: merged all lifecycle related patches in one]
>>>>
>>>> Signed-off-by: Glauber Costa <glommer@parallels.com>
>>>> CC: Kamezawa Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com>
>>>> CC: Christoph Lameter <cl@linux.com>
>>>> CC: Pekka Enberg <penberg@cs.helsinki.fi>

>>>> CC: Michal Hocko <mhocko@suse.cz>
>>>> CC: Johannes Weiner <hannes@cmpxchg.org>
>>>> CC: Suleiman Souhlal <suleiman@google.com>
>>>
>>> OK, I like the optimization. I have just one comment to the
>>> memcg_kmem_dead naming but other than that
>>>
>>> Acked-by: Michal Hocko <mhocko@suse.cz>
>>>
>>> [...]
>>>> +static bool memcg_kmem_dead(struct mem_cgroup *memcg)
>>>
>>> The name is tricky because it doesn't tell you that it clears the flag
>>> which made me scratch my head when reading comment in kmem_cgroup_destroy
>>>
>> memcg_kmem_finally_kill_that_bastard() ?
>
> memcg_kmem_test_and_clear_dead? I know long but at least clear that the
> flag is cleared. Or just open code it.
>

I agree. Ack by me with that naming.

Thanks,
-Kame

Subject: Re: [PATCH v4 10/14] memcg: use static branches when code not in use
Posted by KAMEZAWA Hiroyuki on Tue, 16 Oct 2012 08:48:29 GMT
View Forum Message <> Reply to Message

(2012/10/12 16:47), Glauber Costa wrote:
> On 10/11/2012 05:40 PM, Michal Hocko wrote:
>> On Mon 08-10-12 14:06:16, Glauber Costa wrote:
>>> We can use static branches to patch the code in or out when not used.
>>>
>>> Because the _ACTIVE bit on kmem_accounted is only set after the
>>> increment is done, we guarantee that the root memcg will always be
>>> selected for kmem charges until all call sites are patched (see
>>> memcg_kmem_enabled). This guarantees that no mischarges are applied.
>>>
>>> static branch decrement happens when the last reference count from the
>>> kmem accounting in memcg dies. This will only happen when the charges
>>> drop down to 0.
>>>
>>> When that happen, we need to disable the static branch only on those
>>> memcgs that enabled it. To achieve this, we would be forced to
>>> complicate the code by keeping track of which memcgs were the ones

>>> that actually enabled limits, and which ones got it from its parents.
>>>
>>> It is a lot simpler just to do static_key_slow_inc() on every child
>>> that is accounted.
>>>
>>> [v4: adapted this patch to the changes in kmem_accounted]
>>>
>>> Signed-off-by: Glauber Costa <glommer@parallels.com>
>>> CC: Kamezawa Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com>
>>> CC: Christoph Lameter <cl@linux.com>
>>> CC: Pekka Enberg <penberg@cs.helsinki.fi>
>>> CC: Michal Hocko <mhocko@suse.cz>
>>> CC: Johannes Weiner <hannes@cmpxchg.org>
>>> CC: Suleiman Souhlal <suleiman@google.com>
>>
>> Looks reasonable to me
>> Acked-by: Michal Hocko <mhocko@suse.cz>
>>
>> Just a little nit.
>>
>> [...]
>>

```
>>> diff --git a/mm/memcontrol.c b/mm/memcontrol.c
>>> index 634c7b5..724a08b 100644
>>> --- a/mm/memcontrol.c
>>> +++ b/mm/memcontrol.c
>>> @@ -344,11 +344,15 @@ struct mem_cgroup {
>>>   /* internal only representation about the status of kmem accounting. */
>>>   enum {
>>>    KMEM_ACCOUNTED_ACTIVE = 0, /* accounted by this cgroup itself */
>>> + KMEM_ACCOUNTED_ACTIVATED, /* static key enabled. */
>>>    KMEM_ACCOUNTED_DEAD, /* dead memcg, pending kmem charges */
>>>   };
>>>
>>> -/* first bit */
>>> -#define KMEM_ACCOUNTED_MASK 0x1
>>> +/*
>>> + * first two bits. We account when limit is on, but only after
>>> + * call sites are patched
>>> + */
>>> +#define KMEM_ACCOUNTED_MASK 0x3
```

>>
>> The names are long but why not use KMEM_ACCOUNTED_ACTIVE*
>> #define KMEM_ACCOUNTED_MASK 1<<KMEM_ACCOUNTED_ACTIVE |
1<<KMEM_ACCOUNTED_ACTIVATED
>>
> Because the names are long! =)
>

please use "long" macros ;) it's not bad.

Anyway,

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

Subject: Re: [PATCH v4 14/14] Add documentation about the kmem controller
Posted by KAMEZAWA Hiroyuki on Wed, 17 Oct 2012 07:29:50 GMT
View Forum Message <> Reply to Message

(2012/10/08 19:06), Glauber Costa wrote:
> Signed-off-by: Glauber Costa <glommer@parallels.com>
> ---
>   Documentation/cgroups/memory.txt | 55 +++-
>   1 file changed, 54 insertions(+), 1 deletion(-)
>

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