Subject: [PATCH 00/23] slab+slub accounting for memcg
Posted by Glauber Costa on Fri, 20 Apr 2012 21:57:08 GMT
View Forum Message <> Reply to Message

Hi,

This is my current attempt at getting the kmem controller
into a mergeable state. IMHO, all the important bits are there, and it should't
change *that* much from now on. I am, however, expecting at least a couple more
interactions before we sort all the edges out.

This series works for both the slub and the slab. One of my main goals was to
make sure that the interfaces we are creating actually makes sense for both
allocators.

I did some adaptations to the slab-specific patches, but the bulk of it
comes from Suleiman's patches. I did the best to use his patches
as-is where possible so to keep authorship information. When not possible,
I tried to be fair and quote it in the commit message.

In this series, all existing caches are created per-memcg after its first hit.
The main reason is, during discussions in the memory summit we came into
agreement that the fragmentation problems that could arise from creating all
of them are mitigated by the typically small quantity of caches in the system
(order of a few megabytes total for sparsely used caches).
The lazy creation from Suleiman is kept, although a bit modified. For instance,
I now use a locked scheme instead of cmpxcgh to make sure cache creation won't
fail due to duplicates, which simplifies things by quite a bit.

The slub is a bit more complex than what I came up with in my slub-only
series. The reason is we did not need to use the cache-selection logic
in the allocator itself - it was done by the cache users. But since now
we are lazy creating all caches, this is simply no longer doable.

I am leaving destruction of caches out of the series, although most
of the infrastructure for that is here, since we did it in earlier
series. This is basically because right now Kame is reworking it for
user memcg, and I like the new proposed behavior a lot more. We all seemed
to have agreed that reclaim is an interesting problem by itself, and
is not included in this already too complicated series. Please note
that this is still marked as experimental, so we have so room. A proper
shrinker implementation is a hard requirement to take the kmem controller
out of the experimental state.

I am also not including documentation, but it should only be a matter
of merging what we already wrote in earlier series plus some additions.

Glauber Costa (19):

slub: don't create a copy of the name string in kmem_cache_create
slub: always get the cache from its page in kfree
slab: rename gfpflags to allocflags
slab: use obj_size field of struct kmem_cache when not debugging
change defines to an enum
don't force return value checking in res_counter_charge_nofail
kmem slab accounting basic infrastructure
slab/slub: struct memcg_params
slub: consider a memcg parameter in kmem_create_cache
slab: pass memcg parameter to kmem_cache_create
slub: create duplicate cache
slub: provide kmalloc_no_account
slab: create duplicate cache
slab: provide kmalloc_no_account
kmem controller charge/uncharge infrastructure
slub: charge allocation to a memcg
slab: per-memcg accounting of slab caches
memcg: disable kmem code when not in use.
slub: create slabinfo file for memcg

Suleiman Souhlal (4):
memcg: Make it possible to use the stock for more than one page.
memcg: Reclaim when more than one page needed.
memcg: Track all the memcg children of a kmem_cache.
memcg: Per-memcg memory.kmem.slabinfo file.

include/linux/memcontrol.h | 87 ++++++
include/linux/res_counter.h | 2 +-
include/linux/slab.h | 26 ++
include/linux/slab_def.h | 77 ++++++-
include/linux/slub_def.h | 36 +++-
init/Kconfig | 2 +-
mm/memcontrol.c | 607 ++--
mm/slab.c | 390 ++++++++++++++++++++++++++++++++++++++

registering caches have to keep a copy themselves anyway, since code
needs to work on all allocators.

Having slab create a copy of it as well may very well be the right
thing to do: but at this point, the callers are already there

My motivation for it comes from the kmem slab cache controller for
memcg. Because we create duplicate caches, having a more consistent
behavior here really helps.

I am sending the patch, however, more to probe on your opinion about
it. If you guys agree, but don't want to merge it - since it is not
fixing anything, nor improving any situation etc, I am more than happy
to carry it in my series until it gets merged (fingers crossed).

Signed-off-by: Glauber Costa <glommer@parallels.com>
CC: Christoph Lameter <cl@linux.com>
CC: Pekka Enberg <penberg@cs.helsinki.fi>
---
 mm/slub.c | 14 ++----------
 1 files changed, 2 insertions(+), 12 deletions(-)

diff --git a/mm/slub.c b/mm/slub.c
index ffe13fd..af8cee9 100644
--- a/mm/slub.c
+++ b/mm/slub.c
@@ -3925,7 +3925,6 @@ struct kmem_cache *kmem_cache_create(const char *name, size_t
size,
   size_t align, unsigned long flags, void (*ctor)(void *))
 {
  struct kmem_cache *s;
- char *n;

  if (WARN_ON(!name))
   return NULL;
@@ -3949,26 +3948,20 @@ struct kmem_cache *kmem_cache_create(const char *name, size_t
size,
   return s;
  }

- n = kstrdup(name, GFP_KERNEL);
- if (!n)
- goto err;
-
  s = kmalloc(kmem_size, GFP_KERNEL);
  if (s) {
- if (kmem_cache_open(s, n,
+ if (kmem_cache_open(s, name,

```
     size, align, flags, ctor)) {
    list_add(&s->list, &slab_caches);
    up_write(&slub_lock);
    if (sysfs_slab_add(s)) {
     down_write(&slub_lock);
     list_del(&s->list);
-    kfree(n);
     kfree(s);
     goto err;
    }
    return s;
   }
-  kfree(n);
   kfree(s);
  }
 err:
@@ -5212,7 +5205,6 @@ static void kmem_cache_release(struct kobject *kobj)
 {
  struct kmem_cache *s = to_slab(kobj);

- kfree(s->name);
  kfree(s);
 }

@@ -5318,11 +5310,9 @@ static int sysfs_slab_add(struct kmem_cache *s)
   return err;
  }
  kobject_uevent(&s->kobj, KOBJ_ADD);
- if (!unmergeable) {
+ if (!unmergeable)
   /* Setup first alias */
   sysfs_slab_alias(s, s->name);
-  kfree(name);
- }
  return 0;
 }

--
1.7.7.6
```

Subject: [PATCH 02/23] slub: always get the cache from its page in kfree
Posted by Glauber Costa on Fri, 20 Apr 2012 21:57:10 GMT
View Forum Message <> Reply to Message

struct page already have this information. If we start chaining
caches, this information will always be more trustworthy than
whatever is passed into the function

```
Signed-off-by: Glauber Costa <glommer@parallels.com>
---
 mm/slub.c |    2 +-
 1 files changed, 1 insertions(+), 1 deletions(-)

diff --git a/mm/slub.c b/mm/slub.c
index af8cee9..2652e7c 100644
--- a/mm/slub.c
+++ b/mm/slub.c
@@ -2600,7 +2600,7 @@ void kmem_cache_free(struct kmem_cache *s, void *x)

  page = virt_to_head_page(x);

- slab_free(s, page, x, _RET_IP_);
+ slab_free(page->slab, page, x, _RET_IP_);

  trace_kmem_cache_free(_RET_IP_, x);
 }
--
1.7.7.6
```

Subject: [PATCH 03/23] slab: rename gfpflags to allocflags
Posted by Glauber Costa on Fri, 20 Apr 2012 21:57:11 GMT
View Forum Message <> Reply to Message

```
A consistent name with slub saves us an acessor function.
In both caches, this field represents the same thing. We would
like to use it from the mem_cgroup code.

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

diff --git a/include/linux/slab_def.h b/include/linux/slab_def.h
index fbd1117..d41effe 100644
--- a/include/linux/slab_def.h
+++ b/include/linux/slab_def.h
@@ -39,7 +39,7 @@ struct kmem_cache {
  unsigned int gfporder;

  /* force GFP flags, e.g. GFP_DMA */
- gfp_t gfpflags;
+ gfp_t allocflags;
```

```
  size_t colour;   /* cache colouring range */
  unsigned int colour_off; /* colour offset */
diff --git a/mm/slab.c b/mm/slab.c
index e901a36..c6e5ab8 100644
--- a/mm/slab.c
+++ b/mm/slab.c
@@ -1798,7 +1798,7 @@ static void *kmem_getpages(struct kmem_cache *cachep, gfp_t flags,
int nodeid)
  flags |= __GFP_COMP;
 #endif

- flags |= cachep->gfpflags;
+ flags |= cachep->allocflags;
  if (cachep->flags & SLAB_RECLAIM_ACCOUNT)
   flags |= __GFP_RECLAIMABLE;

@@ -2508,9 +2508,9 @@ kmem_cache_create (const char *name, size_t size, size_t align,
  cachep->colour = left_over / cachep->colour_off;
  cachep->slab_size = slab_size;
  cachep->flags = flags;
- cachep->gfpflags = 0;
+ cachep->allocflags = 0;
  if (CONFIG_ZONE_DMA_FLAG && (flags & SLAB_CACHE_DMA))
-  cachep->gfpflags |= GFP_DMA;
+  cachep->allocflags |= GFP_DMA;
  cachep->buffer_size = size;
  cachep->reciprocal_buffer_size = reciprocal_value(size);

@@ -2857,9 +2857,9 @@ static void kmem_flagcheck(struct kmem_cache *cachep, gfp_t flags)
 {
  if (CONFIG_ZONE_DMA_FLAG) {
   if (flags & GFP_DMA)
-   BUG_ON(!(cachep->gfpflags & GFP_DMA));
+   BUG_ON(!(cachep->allocflags & GFP_DMA));
   else
-   BUG_ON(cachep->gfpflags & GFP_DMA);
+   BUG_ON(cachep->allocflags & GFP_DMA);
  }
 }

--
1.7.7.6
```

Subject: [PATCH 04/23] memcg: Make it possible to use the stock for more than one page.
Posted by Glauber Costa on Fri, 20 Apr 2012 21:57:12 GMT

From: Suleiman Souhlal <ssouhlal@FreeBSD.org>

Signed-off-by: Suleiman Souhlal <suleiman@google.com>
---
 mm/memcontrol.c | 18 +++++++++---------
 1 files changed, 9 insertions(+), 9 deletions(-)

```
diff --git a/mm/memcontrol.c b/mm/memcontrol.c
index 932a734..4b94b2d 100644
--- a/mm/memcontrol.c
+++ b/mm/memcontrol.c
@@ -1998,19 +1998,19 @@ 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.
+ * Try to consume stocked charge on this cpu. If success, nr_pages pages are
+ * 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.
  */
-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;

  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);
@@ -2309,7 +2309,7 @@ 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 {
@@ -2334,7 +2334,7 @@ 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
--
1.7.7.6
```

Subject: [PATCH 05/23] memcg: Reclaim when more than one page needed.
Posted by Glauber Costa on Fri, 20 Apr 2012 21:57:13 GMT
View Forum Message <> Reply to Message

From: Suleiman Souhlal <ssouhlal@FreeBSD.org>

mem_cgroup_do_charge() was written before slab 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 pages (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), stay safe with a cond_resched(),
and make sure not to do it if __GFP_NORETRY.

Signed-off-by: Suleiman Souhlal <suleiman@google.com>
---
 mm/memcontrol.c | 18 ++++++++++++-------
 1 files changed, 11 insertions(+), 7 deletions(-)

diff --git a/mm/memcontrol.c b/mm/memcontrol.c
index 4b94b2d..cbffc4c 100644
--- a/mm/memcontrol.c
+++ b/mm/memcontrol.c
@@ -2187,7 +2187,8 @@ 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;
@@ -2210,18 +2211,18 @@ 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;
@@ -2234,8 +2235,10 @@ 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 <= (PAGE_SIZE << PAGE_ALLOC_COSTLY_ORDER) && ret) {
+  cond_resched();
   return CHARGE_RETRY;
+ }

  /*
   * At task move, charge accounts can be doubly counted. So, it's
@@ -2369,7 +2372,8 @@ 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;
--
1.7.7.6
```

Subject: [PATCH 06/23] slab: use obj_size field of struct kmem_cache when not debugging
Posted by Glauber Costa on Fri, 20 Apr 2012 21:57:14 GMT
View Forum Message <> Reply to Message

The kmem controller needs to keep track of the object size of
a cache so it can later on create a per-memcg duplicate. Logic
to keep track of that already exists, but it is only enable while
debugging.

This patch makes it also available when the kmem controller code
is compiled in.

Signed-off-by: Glauber Costa <glommer@parallels.com>
CC: Christoph Lameter <cl@linux.com>
CC: Pekka Enberg <penberg@cs.helsinki.fi>
---
 include/linux/slab_def.h | 4 +++-
 mm/slab.c | 37 ++++++

```
--- a/mm/slab.c
+++ b/mm/slab.c
@@ -413,8 +413,28 @@ static void kmem_list3_init(struct kmem_list3 *parent)
 #define STATS_INC_FREEMISS(x) do { } while (0)
 #endif

-#if DEBUG
+#if defined(CONFIG_DEBUG_SLAB) || defined(CONFIG_CGROUP_MEM_RES_CTLR_KMEM)
+static int obj_size(struct kmem_cache *cachep)
+{
+ return cachep->obj_size;
+}
+static void set_obj_size(struct kmem_cache *cachep, int size)
+{
+ cachep->obj_size = size;
+}
+
+#else
+static int obj_size(struct kmem_cache *cachep)
+{
+ return cachep->buffer_size;
+}
+
+static void set_obj_size(struct kmem_cache *cachep, int size)
+{
+}
+#endif

+#if DEBUG
 /*
  * memory layout of objects:
  * 0  : objp
@@ -433,11 +453,6 @@ static int obj_offset(struct kmem_cache *cachep)
  return cachep->obj_offset;
 }

-static int obj_size(struct kmem_cache *cachep)
-{
- return cachep->obj_size;
-}
-
 static unsigned long long *dbg_redzone1(struct kmem_cache *cachep, void *objp)
 {
  BUG_ON(!(cachep->flags & SLAB_RED_ZONE));
@@ -465,7 +480,6 @@ static void **dbg_userword(struct kmem_cache *cachep, void *objp)
 #else

 #define obj_offset(x)   0
```

```
-#define obj_size(cachep)  (cachep->buffer_size)
 #define dbg_redzone1(cachep, objp) ({BUG(); (unsigned long long *)NULL;})
 #define dbg_redzone2(cachep, objp) ({BUG(); (unsigned long long *)NULL;})
 #define dbg_userword(cachep, objp) ({BUG(); (void **)NULL;})
@@ -1555,9 +1569,9 @@ void __init kmem_cache_init(void)
   */
  cache_cache.buffer_size = offsetof(struct kmem_cache, array[nr_cpu_ids]) +
       nr_node_ids * sizeof(struct kmem_list3 *);
-#if DEBUG
- cache_cache.obj_size = cache_cache.buffer_size;
-#endif
+
+ set_obj_size(&cache_cache, cache_cache.buffer_size);
+
  cache_cache.buffer_size = ALIGN(cache_cache.buffer_size,
      cache_line_size());
  cache_cache.reciprocal_buffer_size =
@@ -2418,8 +2432,9 @@ kmem_cache_create (const char *name, size_t size, size_t align,
   goto oops;

  cachep->nodelists = (struct kmem_list3 **)&cachep->array[nr_cpu_ids];
+
+ set_obj_size(cachep, size);
 #if DEBUG
- cachep->obj_size = size;

  /*
   * Both debugging options require word-alignment which is calculated
--
1.7.7.6
```

Subject: [PATCH 07/23] change defines to an enum
Posted by Glauber Costa on Fri, 20 Apr 2012 21:57:15 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.

Signed-off-by: Glauber Costa <glommer@parallels.com>
CC: Michal Hocko <mhocko@suse.cz>
CC: Kamezawa Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com>
CC: Johannes Weiner <hannes@cmpxchg.org>
---
 mm/memcontrol.c | 9 ++++++---
 1 files changed, 6 insertions(+), 3 deletions(-)

```
diff --git a/mm/memcontrol.c b/mm/memcontrol.c
index cbffc4c..2810228 100644
--- a/mm/memcontrol.c
+++ b/mm/memcontrol.c
@@ -374,9 +374,12 @@ 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)
--
1.7.7.6
```

Subject: [PATCH 08/23] don't force return value checking in res_counter_charge_nofail

Posted by Glauber Costa on Fri, 20 Apr 2012 21:57:16 GMT

View Forum Message <> Reply to Message

Since we will succeed with the allocation no matter what, there
isn't the need to use __must_check with it. It can very well
be optional.

Signed-off-by: Glauber Costa <glommer@parallels.com>
CC: Kamezawa Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com>
CC: Johannes Weiner <hannes@cmpxchg.org>
CC: Michal Hocko <mhocko@suse.cz>
---
 include/linux/res_counter.h | 2 +-
 1 files changed, 1 insertions(+), 1 deletions(-)

```
diff --git a/include/linux/res_counter.h b/include/linux/res_counter.h
index da81af0..f7621cf 100644
--- a/include/linux/res_counter.h
+++ b/include/linux/res_counter.h
@@ -119,7 +119,7 @@ int __must_check res_counter_charge_locked(struct res_counter
*counter,
   unsigned long val);
```

```
 int __must_check res_counter_charge(struct res_counter *counter,
   unsigned long val, struct res_counter **limit_fail_at);
-int __must_check res_counter_charge_nofail(struct res_counter *counter,
+int res_counter_charge_nofail(struct res_counter *counter,
   unsigned long val, struct res_counter **limit_fail_at);

 /*
--
1.7.7.6
```

Subject: [PATCH 09/23] kmem slab accounting basic infrastructure
Posted by Glauber Costa on Fri, 20 Apr 2012 21:57:17 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.

The code is not enabled until a limit is set. This can be tested by the flag "kmem_accounted". This means that after the patch is applied, no behavioral changes exists for whoever is still using memcg to control their memory usage.

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)

Signed-off-by: Glauber Costa <glommer@parallels.com>
CC: Michal Hocko <mhocko@suse.cz>
CC: Kamezawa Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com>
CC: Johannes Weiner <hannes@cmpxchg.org>
---
 mm/memcontrol.c | 80 ++-
 1 files changed, 79 insertions(+), 1 deletions(-)

diff --git a/mm/memcontrol.c b/mm/memcontrol.c

```
index 2810228..36f1e6b 100644
--- a/mm/memcontrol.c
+++ b/mm/memcontrol.c
@@ -252,6 +252,10 @@ 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.
  */
@@ -266,6 +270,7 @@ struct mem_cgroup {
  * Should the accounting and control be hierarchical, per subtree?
  */
 bool use_hierarchy;
+bool kmem_accounted;

 bool  oom_lock;
 atomic_t under_oom;
@@ -378,6 +383,7 @@ enum res_type {
 _MEM,
 _MEMSWAP,
 _OOM_TYPE,
+_KMEM,
 };

 #define MEMFILE_PRIVATE(x, val) (((x) << 16) | (val))
@@ -1470,6 +1476,10 @@ 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));
 }

 /*
@@ -3914,6 +3924,11 @@ static ssize_t mem_cgroup_read(struct cgroup *cont, struct cftype
*cft,
  else
   val = res_counter_read_u64(&memcg->memsw, name);
  break;
+#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
+ case _KMEM:
```

```
+  val = res_counter_read_u64(&memcg->kmem, name);
+  break;
+#endif
  default:
   BUG();
  }
@@ -3951,8 +3966,26 @@ 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);
+#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
+  else if (type == _KMEM) {
+   ret = res_counter_set_limit(&memcg->kmem, val);
+   if (ret)
+    break;
+   /*
+    * Once enabled, can't be disabled. We could in theory
+    * disable it if we haven't yet created any caches, or
+    * if we can shrink them all to death.
+    *
+    * But it is not worth the trouble
+    */
+   if (!memcg->kmem_accounted && val != RESOURCE_MAX)
+    memcg->kmem_accounted = true;
+  }
+#endif
+  else
+   return -EINVAL;
   break;
  case RES_SOFT_LIMIT:
   ret = res_counter_memparse_write_strategy(buffer, &val);
@@ -4017,12 +4050,20 @@ static int mem_cgroup_reset(struct cgroup *cont, unsigned int
event)
  case RES_MAX_USAGE:
   if (type == _MEM)
    res_counter_reset_max(&memcg->res);
+#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
+  else if (type == _KMEM)
+   res_counter_reset_max(&memcg->kmem);
+#endif
   else
    res_counter_reset_max(&memcg->memsw);
   break;
  case RES_FAILCNT:
   if (type == _MEM)
```

```
    res_counter_reset_failcnt(&memcg->res);
+#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
+  else if (type == _KMEM)
+   res_counter_reset_failcnt(&memcg->kmem);
+#endif
   else
    res_counter_reset_failcnt(&memcg->memsw);
   break;
@@ -4647,6 +4688,33 @@ static int mem_control_numa_stat_open(struct inode *unused, struct
file *file)
 #endif /* CONFIG_NUMA */

 #ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
+static struct cftype kmem_cgroup_files[] = {
+ {
+  .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,
+ },
+ {},
+};
+
 static int memcg_init_kmem(struct mem_cgroup *memcg, struct cgroup_subsys *ss)
 {
  return mem_cgroup_sockets_init(memcg, ss);
@@ -4654,6 +4722,7 @@ static int memcg_init_kmem(struct mem_cgroup *memcg, struct
cgroup_subsys *ss)

 static void kmem_cgroup_destroy(struct mem_cgroup *memcg)
 {
```

```
+ BUG_ON(res_counter_read_u64(&memcg->kmem, RES_USAGE) != 0);
  mem_cgroup_sockets_destroy(memcg);
 }
 #else
@@ -4979,6 +5048,12 @@ mem_cgroup_create(struct cgroup *cont)
   int cpu;
   enable_swap_cgroup();
   parent = NULL;
+
+#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
+  WARN_ON(cgroup_add_cftypes(&mem_cgroup_subsys,
+        kmem_cgroup_files));
+#endif
+
   if (mem_cgroup_soft_limit_tree_init())
    goto free_out;
   root_mem_cgroup = memcg;
@@ -4997,6 +5072,7 @@ 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.
@@ -5007,6 +5083,7 @@ mem_cgroup_create(struct cgroup *cont)
  } else {
   res_counter_init(&memcg->res, NULL);
   res_counter_init(&memcg->memsw, NULL);
+  res_counter_init(&memcg->kmem, NULL);
  }
  memcg->last_scanned_node = MAX_NUMNODES;
  INIT_LIST_HEAD(&memcg->oom_notify);
@@ -5014,6 +5091,7 @@ mem_cgroup_create(struct cgroup *cont)
  if (parent)
   memcg->swappiness = mem_cgroup_swappiness(parent);
  atomic_set(&memcg->refcnt, 1);
+ memcg->kmem_accounted = false;
  memcg->move_charge_at_immigrate = 0;
  mutex_init(&memcg->thresholds_lock);
  spin_lock_init(&memcg->move_lock);
--
1.7.7.6
```

Subject: [PATCH 10/23] slab/slub: struct memcg_params
Posted by Glauber Costa on Fri, 20 Apr 2012 21:57:18 GMT

For the kmem slab controller, we need to record some extra
information in the kmem_cache structure.

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>
CC: Suleiman Souhlal <suleiman@google.com>
---
 include/linux/slab.h | 15 +++++++++++++++
 include/linux/slab_def.h | 4 ++++
 include/linux/slub_def.h | 3 +++
 3 files changed, 22 insertions(+), 0 deletions(-)

diff --git a/include/linux/slab.h b/include/linux/slab.h
index a595dce..a5127e1 100644
--- a/include/linux/slab.h
+++ b/include/linux/slab.h
@@ -153,6 +153,21 @@ unsigned int kmem_cache_size(struct kmem_cache *);
 #define ARCH_SLAB_MINALIGN __alignof__(unsigned long long)
 #endif

+#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
+struct mem_cgroup_cache_params {
+ struct mem_cgroup *memcg;
+ int id;
+
+#ifdef CONFIG_SLAB
+ /* Original cache parameters, used when creating a memcg cache */
+ size_t orig_align;
+ atomic_t refcnt;
+
+#endif
+ struct list_head destroyed_list; /* Used when deleting cpuset cache */
+};
+#endif
+
 /*
  * Common kmalloc functions provided by all allocators
  */
diff --git a/include/linux/slab_def.h b/include/linux/slab_def.h
index cba3139..06e4a3e 100644
--- a/include/linux/slab_def.h
+++ b/include/linux/slab_def.h
@@ -83,6 +83,10 @@ struct kmem_cache {

```
  int obj_size;
 #endif

+#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
+ struct mem_cgroup_cache_params memcg_params;
+#endif
+
 /* 6) per-cpu/per-node data, touched during every alloc/free */
  /*
   * We put array[] at the end of kmem_cache, because we want to size
diff --git a/include/linux/slub_def.h b/include/linux/slub_def.h
index c2f8c8b..5f5e942 100644
--- a/include/linux/slub_def.h
+++ b/include/linux/slub_def.h
@@ -102,6 +102,9 @@ struct kmem_cache {
 #ifdef CONFIG_SYSFS
  struct kobject kobj; /* For sysfs */
 #endif
+#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
+ struct mem_cgroup_cache_params memcg_params;
+#endif

 #ifdef CONFIG_NUMA
  /*
--
1.7.7.6
```

Subject: [PATCH 11/23] slub: consider a memcg parameter in kmem_create_cache
Posted by Glauber Costa on Fri, 20 Apr 2012 21:57:19 GMT
View Forum Message <> Reply to Message

Allow a memcg parameter to be passed during cache creation.
The slub allocator will only merge caches that belong to
the same memcg.

Default function is created as a wrapper, passing NULL
to the memcg version. We only merge caches that belong
to the same memcg.

>From the memcontrol.c side, 3 helper functions are created:

 1) memcg_css_id: because slub needs a unique cache name
    for sysfs. Since this is visible, but not the canonical
    location for slab data, the cache name is not used, the
    css_id should suffice.

 2) mem_cgroup_register_cache: is responsible for assigning

a unique index to each cache, and other general purpose
setup. The index is only assigned for the root caches. All
others are assigned index == -1.

3) mem_cgroup_release_cache: can be called from the root cache
destruction, and will release the index for other caches.

This index mechanism was developed by Suleiman Souhlal.

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>
CC: Suleiman Souhlal <suleiman@google.com>
---
include/linux/memcontrol.h | 14 +++++++++++++
include/linux/slab.h | 6 ++++++
mm/memcontrol.c | 29 +++++++++++++++++++++++++++++
mm/slub.c | 31 +++++++++++++++++++++++++++++++----
4 files changed, 76 insertions(+), 4 deletions(-)

diff --git a/include/linux/memcontrol.h b/include/linux/memcontrol.h
index f94efd2..99e14b9 100644
--- a/include/linux/memcontrol.h
+++ b/include/linux/memcontrol.h
@@ -26,6 +26,7 @@ struct mem_cgroup;
struct page_cgroup;
struct page;
struct mm_struct;
+struct kmem_cache;

/* Stats that can be updated by kernel. */
enum mem_cgroup_page_stat_item {
@@ -440,7 +441,20 @@ struct sock;
#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
void sock_update_memcg(struct sock *sk);
void sock_release_memcg(struct sock *sk);
+int memcg_css_id(struct mem_cgroup *memcg);
+void mem_cgroup_register_cache(struct mem_cgroup *memcg,
+ struct kmem_cache *s);
+void mem_cgroup_release_cache(struct kmem_cache *cachep);
#else
+static inline void mem_cgroup_register_cache(struct mem_cgroup *memcg,
+ struct kmem_cache *s)
+{
+}

```
+
+static inline void mem_cgroup_release_cache(struct kmem_cache *cachep)
+{
+}
+
 static inline void sock_update_memcg(struct sock *sk)
 {
 }
diff --git a/include/linux/slab.h b/include/linux/slab.h
index a5127e1..c7a7e05 100644
--- a/include/linux/slab.h
+++ b/include/linux/slab.h
@@ -321,6 +321,12 @@ extern void *__kmalloc_track_caller(size_t, gfp_t, unsigned long);
  __kmalloc(size, flags)
 #endif /* DEBUG_SLAB */

+#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
+#define MAX_KMEM_CACHE_TYPES 400
+#else
+#define MAX_KMEM_CACHE_TYPES 0
+#endif /* CONFIG_CGROUP_MEM_RES_CTLR_KMEM */
+
 #ifdef CONFIG_NUMA
 /*
  * kmalloc_node_track_caller is a special version of kmalloc_node that
diff --git a/mm/memcontrol.c b/mm/memcontrol.c
index 36f1e6b..0015ed0 100644
--- a/mm/memcontrol.c
+++ b/mm/memcontrol.c
@@ -323,6 +323,11 @@ struct mem_cgroup {
 #endif
 };

+int memcg_css_id(struct mem_cgroup *memcg)
+{
+ return css_id(&memcg->css);
+}
+
 /* Stuffs for move charges at task migration. */
 /*
  * Types of charges to be moved. "move_charge_at_immitgrate" is treated as a
@@ -461,6 +466,30 @@ struct cg_proto *tcp_proto_cgroup(struct mem_cgroup *memcg)
 }
 EXPORT_SYMBOL(tcp_proto_cgroup);
 #endif /* CONFIG_INET */
+
+/* Bitmap used for allocating the cache id numbers. */
+static DECLARE_BITMAP(cache_types, MAX_KMEM_CACHE_TYPES);
```

```
+
+void mem_cgroup_register_cache(struct mem_cgroup *memcg,
+          struct kmem_cache *cachep)
+{
+ int id = -1;
+
+ cachep->memcg_params.memcg = memcg;
+
+ if (!memcg) {
+  id = find_first_zero_bit(cache_types, MAX_KMEM_CACHE_TYPES);
+  BUG_ON(id < 0 || id >= MAX_KMEM_CACHE_TYPES);
+  __set_bit(id, cache_types);
+ } else
+  INIT_LIST_HEAD(&cachep->memcg_params.destroyed_list);
+ cachep->memcg_params.id = id;
+}
+
+void mem_cgroup_release_cache(struct kmem_cache *cachep)
+{
+ __clear_bit(cachep->memcg_params.id, cache_types);
+}
 #endif /* CONFIG_CGROUP_MEM_RES_CTLR_KMEM */

 static void drain_all_stock_async(struct mem_cgroup *memcg);
diff --git a/mm/slub.c b/mm/slub.c
index 2652e7c..86e40cc 100644
--- a/mm/slub.c
+++ b/mm/slub.c
@@ -32,6 +32,7 @@
 #include <linux/prefetch.h>

 #include <trace/events/kmem.h>
+#include <linux/memcontrol.h>

 /*
  * Lock order:
@@ -3880,7 +3881,7 @@ static int slab_unmergeable(struct kmem_cache *s)
  return 0;
 }

-static struct kmem_cache *find_mergeable(size_t size,
+static struct kmem_cache *find_mergeable(struct mem_cgroup *memcg, size_t size,
   size_t align, unsigned long flags, const char *name,
   void (*ctor)(void *))
 {
@@ -3916,21 +3917,29 @@ static struct kmem_cache *find_mergeable(size_t size,
   if (s->size - size >= sizeof(void *))
    continue;
```

```
+  if (memcg && s->memcg_params.memcg != memcg)
+   continue;
+
   return s;
  }
  return NULL;
 }

-struct kmem_cache *kmem_cache_create(const char *name, size_t size,
-  size_t align, unsigned long flags, void (*ctor)(void *))
+struct kmem_cache *
+kmem_cache_create_memcg(struct mem_cgroup *memcg, const char *name, size_t size,
+   size_t align, unsigned long flags, void (*ctor)(void *))
 {
  struct kmem_cache *s;

  if (WARN_ON(!name))
   return NULL;

+#ifndef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
+ WARN_ON(memcg != NULL);
+#endif
+
  down_write(&slub_lock);
- s = find_mergeable(size, align, flags, name, ctor);
+ s = find_mergeable(memcg, size, align, flags, name, ctor);
  if (s) {
   s->refcount++;
   /*
@@ -3954,12 +3963,15 @@ struct kmem_cache *kmem_cache_create(const char *name, size_t
size,
     size, align, flags, ctor)) {
    list_add(&s->list, &slab_caches);
    up_write(&slub_lock);
+   mem_cgroup_register_cache(memcg, s);
    if (sysfs_slab_add(s)) {
     down_write(&slub_lock);
     list_del(&s->list);
     kfree(s);
     goto err;
    }
+   if (memcg)
+    s->refcount++;
    return s;
   }
   kfree(s);
@@ -3973,6 +3985,12 @@ err:
```

```
   s = NULL;
  return s;
 }
+
+struct kmem_cache *kmem_cache_create(const char *name, size_t size,
+  size_t align, unsigned long flags, void (*ctor)(void *))
+{
+ return kmem_cache_create_memcg(NULL, name, size, align, flags, ctor);
+}
 EXPORT_SYMBOL(kmem_cache_create);

 #ifdef CONFIG_SMP
@@ -5265,6 +5283,11 @@ static char *create_unique_id(struct kmem_cache *s)
  if (p != name + 1)
   *p++ = '-';
  p += sprintf(p, "%07d", s->size);
+
+#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
+ if (s->memcg_params.memcg)
+  p += sprintf(p, "-%08d", memcg_css_id(s->memcg_params.memcg));
+#endif
  BUG_ON(p > name + ID_STR_LENGTH - 1);
  return name;
 }
--
1.7.7.6
```

Subject: [PATCH 12/23] slab: pass memcg parameter to kmem_cache_create
Posted by Glauber Costa on Sun, 22 Apr 2012 23:53:29 GMT
View Forum Message <> Reply to Message

Allow a memcg parameter to be passed during cache creation.

Default function is created as a wrapper, passing NULL
to the memcg version. We only merge caches that belong
to the same memcg.

This code was mostly written by Suleiman Souhlal and
only adapted to my patchset, plus a couple of simplifications

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>
CC: Suleiman Souhlal <suleiman@google.com>

```
---
 mm/slab.c |   38 ++++++++++++++++++++++++++++++++++++++---------
 1 files changed, 29 insertions(+), 9 deletions(-)

diff --git a/mm/slab.c b/mm/slab.c
index a0d51dd..362bb6e 100644
--- a/mm/slab.c
+++ b/mm/slab.c
@@ -2287,14 +2287,15 @@ static int __init_refok setup_cpu_cache(struct kmem_cache
*cachep, gfp_t gfp)
  * cacheline.  This can be beneficial if you're counting cycles as closely
  * as davem.
  */
-struct kmem_cache *
-kmem_cache_create (const char *name, size_t size, size_t align,
- unsigned long flags, void (*ctor)(void *))
+static struct kmem_cache *
+__kmem_cache_create(struct mem_cgroup *memcg, const char *name, size_t size,
+      size_t align, unsigned long flags, void (*ctor)(void *))
 {
- size_t left_over, slab_size, ralign;
+ size_t left_over, orig_align, ralign, slab_size;
  struct kmem_cache *cachep = NULL, *pc;
  gfp_t gfp;

+ orig_align = align;
  /*
   * Sanity checks... these are all serious usage bugs.
   */
@@ -2311,7 +2312,6 @@ kmem_cache_create (const char *name, size_t size, size_t align,
   */
  if (slab_is_available()) {
   get_online_cpus();
-  mutex_lock(&cache_chain_mutex);
  }

  list_for_each_entry(pc, &cache_chain, next) {
@@ -2331,9 +2331,9 @@ kmem_cache_create (const char *name, size_t size, size_t align,
    continue;
   }

-  if (!strcmp(pc->name, name)) {
+  if (!strcmp(pc->name, name) && !memcg) {
    printk(KERN_ERR
-          "kmem_cache_create: duplicate cache %s\n", name);
+   "kmem_cache_create: duplicate cache %s\n", name);
    dump_stack();
    goto oops;
```

```
   }
@@ -2434,6 +2434,9 @@ kmem_cache_create (const char *name, size_t size, size_t align,
  cachep->nodelists = (struct kmem_list3 **)&cachep->array[nr_cpu_ids];

  set_obj_size(cachep, size);
+#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
+ cachep->memcg_params.orig_align = orig_align;
+#endif
 #if DEBUG

  /*
@@ -2541,7 +2544,12 @@ kmem_cache_create (const char *name, size_t size, size_t align,
   BUG_ON(ZERO_OR_NULL_PTR(cachep->slabp_cache));
  }
  cachep->ctor = ctor;
- cachep->name = name;
+ cachep->name = (char *)name;
+
+#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
+ mem_cgroup_register_cache(memcg, cachep);
+ atomic_set(&cachep->memcg_params.refcnt, 1);
+#endif

  if (setup_cpu_cache(cachep, gfp)) {
   __kmem_cache_destroy(cachep);
@@ -2566,11 +2574,23 @@ oops:
   panic("kmem_cache_create(): failed to create slab `%s'\n",
         name);
  if (slab_is_available()) {
-  mutex_unlock(&cache_chain_mutex);
   put_online_cpus();
  }
  return cachep;
 }
+
+struct kmem_cache *
+kmem_cache_create(const char *name, size_t size, size_t align,
+    unsigned long flags, void (*ctor)(void *))
+{
+ struct kmem_cache *cachep;
+
+ mutex_lock(&cache_chain_mutex);
+ cachep = __kmem_cache_create(NULL, name, size, align, flags, ctor);
+ mutex_unlock(&cache_chain_mutex);
+
+ return cachep;
+}
 EXPORT_SYMBOL(kmem_cache_create);
```

```
 #if DEBUG
--
1.7.7.6
```

Subject: [PATCH 13/23] slub: create duplicate cache
Posted by Glauber Costa on Sun, 22 Apr 2012 23:53:30 GMT
View Forum Message <> Reply to Message

This patch provides kmem_cache_dup(), that duplicates
a cache for a memcg, preserving its creation properties.
Object size, alignment and flags are all respected.

When a duplicate cache is created, the parent cache cannot
be destructed during the child lifetime. To assure this,
its reference count is increased if the cache creation
succeeds.

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>
CC: Suleiman Souhlal <suleiman@google.com>

```
 include/linux/memcontrol.h |    3 +++
 include/linux/slab.h       |    3 +++
 mm/memcontrol.c            |   44 ++++++++++++++++++++++++++++++++++++++++++++
 mm/slub.c                  |   37 +++++++++++++++++++++++++++++++++++++
 4 files changed, 87 insertions(+), 0 deletions(-)

diff --git a/include/linux/memcontrol.h b/include/linux/memcontrol.h
index 99e14b9..493ecdd 100644
--- a/include/linux/memcontrol.h
+++ b/include/linux/memcontrol.h
@@ -445,6 +445,9 @@ int memcg_css_id(struct mem_cgroup *memcg);
 void mem_cgroup_register_cache(struct mem_cgroup *memcg,
                struct kmem_cache *s);
 void mem_cgroup_release_cache(struct kmem_cache *cachep);
+extern char *mem_cgroup_cache_name(struct mem_cgroup *memcg,
+       struct kmem_cache *cachep);
+
 #else
 static inline void mem_cgroup_register_cache(struct mem_cgroup *memcg,
                struct kmem_cache *s)
diff --git a/include/linux/slab.h b/include/linux/slab.h
```

```
index c7a7e05..909b508 100644
--- a/include/linux/slab.h
+++ b/include/linux/slab.h
@@ -323,6 +323,9 @@ extern void *__kmalloc_track_caller(size_t, gfp_t, unsigned long);

 #ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
 #define MAX_KMEM_CACHE_TYPES 400
+extern struct kmem_cache *kmem_cache_dup(struct mem_cgroup *memcg,
+      struct kmem_cache *cachep);
+void kmem_cache_drop_ref(struct kmem_cache *cachep);
 #else
 #define MAX_KMEM_CACHE_TYPES 0
 #endif /* CONFIG_CGROUP_MEM_RES_CTLR_KMEM */
diff --git a/mm/memcontrol.c b/mm/memcontrol.c
index 0015ed0..e881d83 100644
--- a/mm/memcontrol.c
+++ b/mm/memcontrol.c
@@ -467,6 +467,50 @@ struct cg_proto *tcp_proto_cgroup(struct mem_cgroup *memcg)
 EXPORT_SYMBOL(tcp_proto_cgroup);
 #endif /* CONFIG_INET */

+/*
+ * This is to prevent races againt the kmalloc cache creations.
+ * Should never be used outside the core memcg code. Therefore,
+ * copy it here, instead of letting it in lib/
+ */
+static char *kasprintf_no_account(gfp_t gfp, const char *fmt, ...)
+{
+ unsigned int len;
+ char *p = NULL;
+ va_list ap, aq;
+
+ va_start(ap, fmt);
+ va_copy(aq, ap);
+ len = vsnprintf(NULL, 0, fmt, aq);
+ va_end(aq);
+
+ p = kmalloc_no_account(len+1, gfp);
+ if (!p)
+  goto out;
+
+ vsnprintf(p, len+1, fmt, ap);
+
+out:
+ va_end(ap);
+ return p;
+}
+
```

```
+char *mem_cgroup_cache_name(struct mem_cgroup *memcg, struct kmem_cache *cachep)
+{
+ char *name;
+ struct dentry *dentry = memcg->css.cgroup->dentry;
+
+ BUG_ON(dentry == NULL);
+
+ /* Preallocate the space for "dead" at the end */
+ name = kasprintf_no_account(GFP_KERNEL, "%s(%d:%s)dead",
+     cachep->name, css_id(&memcg->css), dentry->d_name.name);
+
+ if (name)
+  /* Remove "dead" */
+  name[strlen(name) - 4] = '\0';
+ return name;
+}
+
 /* Bitmap used for allocating the cache id numbers. */
 static DECLARE_BITMAP(cache_types, MAX_KMEM_CACHE_TYPES);

diff --git a/mm/slub.c b/mm/slub.c
index 86e40cc..2285a96 100644
--- a/mm/slub.c
+++ b/mm/slub.c
@@ -3993,6 +3993,43 @@ struct kmem_cache *kmem_cache_create(const char *name, size_t
size,
 }
 EXPORT_SYMBOL(kmem_cache_create);

+#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
+struct kmem_cache *kmem_cache_dup(struct mem_cgroup *memcg,
+      struct kmem_cache *s)
+{
+ char *name;
+ struct kmem_cache *new;
+
+ name = mem_cgroup_cache_name(memcg, s);
+ if (!name)
+  return NULL;
+
+ new = kmem_cache_create_memcg(memcg, name, s->objsize, s->align,
+          s->allocflags, s->ctor);
+
+ /*
+  * We increase the reference counter in the parent cache, to
+  * prevent it from being deleted. If kmem_cache_destroy() is
+  * called for the root cache before we call it for a child cache,
+  * it will be queued for destruction when we finally drop the
```

```
+  * reference on the child cache.
+  */
+ if (new) {
+  down_write(&slub_lock);
+  s->refcount++;
+  up_write(&slub_lock);
+ }
+
+ return new;
+}
+
+void kmem_cache_drop_ref(struct kmem_cache *s)
+{
+ BUG_ON(s->memcg_params.id != -1);
+ kmem_cache_destroy(s);
+}
+#endif
+
 #ifdef CONFIG_SMP
 /*
  * Use the cpu notifier to insure that the cpu slabs are flushed when
--
1.7.7.6
```

Subject: [PATCH 14/23] slub: provide kmalloc_no_account
Posted by Glauber Costa on Sun, 22 Apr 2012 23:53:31 GMT
View Forum Message <> Reply to Message

Some allocations need to be accounted to the root memcg regardless
of their context. One trivial example, is the allocations we do
during the memcg slab cache creation themselves. Strictly speaking,
they could go to the parent, but it is way easier to bill them to
the root cgroup.

Only generic kmalloc allocations are allowed to be bypassed.

The function is not exported, because drivers code should always
be accounted.

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>
CC: Suleiman Souhlal <suleiman@google.com>

```
 include/linux/slub_def.h |    1 +
 mm/slub.c                |   21 ++++++++++++++++++++
 2 files changed, 22 insertions(+), 0 deletions(-)

diff --git a/include/linux/slub_def.h b/include/linux/slub_def.h
index 5f5e942..9a8000a 100644
--- a/include/linux/slub_def.h
+++ b/include/linux/slub_def.h
@@ -221,6 +221,7 @@ static __always_inline struct kmem_cache *kmalloc_slab(size_t size)
 }

 void *kmem_cache_alloc(struct kmem_cache *, gfp_t);
+void *kmalloc_no_account(size_t size, gfp_t);
 void *__kmalloc(size_t size, gfp_t flags);

 static __always_inline void *
diff --git a/mm/slub.c b/mm/slub.c
index 2285a96..d754b06 100644
--- a/mm/slub.c
+++ b/mm/slub.c
@@ -3359,6 +3359,27 @@ void *__kmalloc(size_t size, gfp_t flags)
 }
 EXPORT_SYMBOL(__kmalloc);

+void *kmalloc_no_account(size_t size, gfp_t flags)
+{
+ struct kmem_cache *s;
+ void *ret;
+
+ if (unlikely(size > SLUB_MAX_SIZE))
+  return kmalloc_large(size, flags);
+
+ s = get_slab(size, flags);
+
+ if (unlikely(ZERO_OR_NULL_PTR(s)))
+  return s;
+
+ ret = slab_alloc(s, flags, NUMA_NO_NODE, _RET_IP_);
+
+ trace_kmalloc(_RET_IP_, ret, size, s->size, flags);
+
+ return ret;
+}
+EXPORT_SYMBOL(kmalloc_no_account);
+
 #ifdef CONFIG_NUMA
 static void *kmalloc_large_node(size_t size, gfp_t flags, int node)
 {
```

--
1.7.7.6

Subject: [PATCH 15/23] slab: create duplicate cache
Posted by Glauber Costa on Sun, 22 Apr 2012 23:53:32 GMT
View Forum Message <> Reply to Message

This patch provides kmem_cache_dup(), that duplicates
a cache for a memcg, preserving its creation properties.
Object size, alignment and flags are all respected.
An exception is the SLAB_PANIC flag, since cache creation
inside a memcg should not be fatal.

This code is mostly written by Suleiman Souhlal,
with some adaptations and simplifications by me.

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>
CC: Suleiman Souhlal <suleiman@google.com>
---
 mm/slab.c | 36 ++++++++++++++++++++++++++++++++++++
 1 files changed, 36 insertions(+), 0 deletions(-)

```
diff --git a/mm/slab.c b/mm/slab.c
index 362bb6e..c4ef684 100644
--- a/mm/slab.c
+++ b/mm/slab.c
@@ -301,6 +301,8 @@ static void free_block(struct kmem_cache *cachep, void **objpp, int len,
    int node);
 static int enable_cpucache(struct kmem_cache *cachep, gfp_t gfp);
 static void cache_reap(struct work_struct *unused);
+static int do_tune_cpucache(struct kmem_cache *cachep, int limit,
+       int batchcount, int shared, gfp_t gfp);

 /*
  * This function must be completely optimized away if a constant is passed to
@@ -2593,6 +2595,40 @@ kmem_cache_create(const char *name, size_t size, size_t align,
 }
 EXPORT_SYMBOL(kmem_cache_create);

+#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
+struct kmem_cache *
+kmem_cache_dup(struct mem_cgroup *memcg, struct kmem_cache *cachep)
```

```
+{
+ struct kmem_cache *new;
+ int flags;
+ char *name;
+
+ name = mem_cgroup_cache_name(memcg, cachep);
+ if (!name)
+  return NULL;
+
+ flags = cachep->flags & ~SLAB_PANIC;
+ mutex_lock(&cache_chain_mutex);
+ new = __kmem_cache_create(memcg, name, obj_size(cachep),
+     cachep->memcg_params.orig_align, flags, cachep->ctor);
+
+ if (new == NULL) {
+  mutex_unlock(&cache_chain_mutex);
+  kfree(name);
+  return NULL;
+ }
+
+ if ((cachep->limit != new->limit) ||
+     (cachep->batchcount != new->batchcount) ||
+     (cachep->shared != new->shared))
+  do_tune_cpucache(new, cachep->limit, cachep->batchcount,
+      cachep->shared, GFP_KERNEL);
+ mutex_unlock(&cache_chain_mutex);
+
+ return new;
+}
+#endif /* CONFIG_CGROUP_MEM_RES_CTLR_KMEM */
+
 #if DEBUG
 static void check_irq_off(void)
 {
--
1.7.7.6
```

Subject: [PATCH 16/23] slab: provide kmalloc_no_account
Posted by Glauber Costa on Sun, 22 Apr 2012 23:53:33 GMT
View Forum Message <> Reply to Message

Some allocations need to be accounted to the root memcg regardless
of their context. One trivial example, is the allocations we do
during the memcg slab cache creation themselves. Strictly speaking,
they could go to the parent, but it is way easier to bill them to
the root cgroup.

Only generic kmalloc allocations are allowed to be bypassed.

The function is not exported, because drivers code should always
be accounted.

This code is mosly written by Suleiman Souhlal.

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>
CC: Suleiman Souhlal <suleiman@google.com>
---
 include/linux/slab_def.h | 1 +
 mm/slab.c | 23 +++++++++++++++++++++++
 2 files changed, 24 insertions(+), 0 deletions(-)

```
diff --git a/include/linux/slab_def.h b/include/linux/slab_def.h
index 06e4a3e..54d25d7 100644
--- a/include/linux/slab_def.h
+++ b/include/linux/slab_def.h
@@ -114,6 +114,7 @@ extern struct cache_sizes malloc_sizes[];

 void *kmem_cache_alloc(struct kmem_cache *, gfp_t);
 void *__kmalloc(size_t size, gfp_t flags);
+void *kmalloc_no_account(size_t size, gfp_t flags);

 #ifdef CONFIG_TRACING
 extern void *kmem_cache_alloc_trace(size_t size,
diff --git a/mm/slab.c b/mm/slab.c
index c4ef684..13948c3 100644
--- a/mm/slab.c
+++ b/mm/slab.c
@@ -3960,6 +3960,29 @@ void *__kmalloc(size_t size, gfp_t flags)
 }
 EXPORT_SYMBOL(__kmalloc);

+static __always_inline void *__do_kmalloc_no_account(size_t size, gfp_t flags,
+           void *caller)
+{
+ struct kmem_cache *cachep;
+ void *ret;
+
+ cachep = __find_general_cachep(size, flags);
+ if (unlikely(ZERO_OR_NULL_PTR(cachep)))
+  return cachep;
```

```
+
+ ret = __cache_alloc(cachep, flags, caller);
+ trace_kmalloc((unsigned long)caller, ret, size,
+       cachep->buffer_size, flags);
+
+ return ret;
+}
+
+void *kmalloc_no_account(size_t size, gfp_t flags)
+{
+ return __do_kmalloc_no_account(size, flags,
+          __builtin_return_address(0));
+}
+
 void *__kmalloc_track_caller(size_t size, gfp_t flags, unsigned long caller)
 {
  return __do_kmalloc(size, flags, (void *)caller);
--
1.7.7.6
```

Subject: [PATCH 17/23] kmem controller charge/uncharge infrastructure
Posted by Glauber Costa on Sun, 22 Apr 2012 23:53:34 GMT
View Forum Message <> Reply to Message

With all the dependencies already in place, this patch introduces
the charge/uncharge functions for the slab cache accounting in memcg.

Before we can charge a cache, we need to select the right cache.
This is done by using the function __mem_cgroup_get_kmem_cache().

If we should use the root kmem cache, this function tries to detect
that and return as early as possible.

The charge and uncharge functions comes in two flavours:
 * __mem_cgroup_(un)charge_slab(), that assumes the allocation is
   a slab page, and
 * __mem_cgroup_(un)charge_kmem(), that does not. This later exists
   because the slub allocator draws the larger kmalloc allocations
   from the page allocator.

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

Because the slub allocator tends to inline the allocations whenever
it can, those functions need to be exported so modules can make use
of it properly.

I apologize in advance to the reviewers. This patch is quite big, but
I was not able to split it any further due to all the dependencies
between the code.

This code is inspired by the code written by Suleiman Souhlal,
but heavily changed.

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>
CC: Suleiman Souhlal <suleiman@google.com>
---
 include/linux/memcontrol.h | 68 ++++++++
 init/Kconfig | 2 +-
 mm/memcontrol.c | 373 ++-
 3 files changed, 441 insertions(+), 2 deletions(-)

diff --git a/include/linux/memcontrol.h b/include/linux/memcontrol.h
index 493ecdd..c1c1302 100644
--- a/include/linux/memcontrol.h
+++ b/include/linux/memcontrol.h
@@ -448,6 +448,21 @@ void mem_cgroup_release_cache(struct kmem_cache *cachep);
 extern char *mem_cgroup_cache_name(struct mem_cgroup *memcg,
 		struct kmem_cache *cachep);

+void mem_cgroup_flush_cache_create_queue(void);
+void mem_cgroup_remove_child_kmem_cache(struct kmem_cache *cachep, int id);
+bool __mem_cgroup_charge_slab(struct kmem_cache *cachep, gfp_t gfp,
+		size_t size);
+void __mem_cgroup_uncharge_slab(struct kmem_cache *cachep, size_t size);
+
+bool __mem_cgroup_charge_kmem(gfp_t gfp, size_t size);
+void __mem_cgroup_uncharge_kmem(size_t size);
+
+struct kmem_cache *
+__mem_cgroup_get_kmem_cache(struct kmem_cache *cachep, gfp_t gfp);
+
+#define mem_cgroup_kmem_on 1
+
+void mem_cgroup_destroy_cache(struct kmem_cache *cachep);
 #else
 static inline void mem_cgroup_register_cache(struct mem_cgroup *memcg,
 		struct kmem_cache *s)
@@ -464,6 +479,59 @@ static inline void sock_update_memcg(struct sock *sk)

```
 static inline void sock_release_memcg(struct sock *sk)
 {
 }
+
+static inline void
+mem_cgroup_flush_cache_create_queue(void)
+{
+}
+
+static inline void mem_cgroup_destroy_cache(struct kmem_cache *cachep)
+{
+}
+
+#define mem_cgroup_kmem_on 0
+#define __mem_cgroup_get_kmem_cache(a, b) a
+#define __mem_cgroup_charge_slab(a, b, c) false
+#define __mem_cgroup_charge_kmem(a, b) false
+#define __mem_cgroup_uncharge_slab(a, b)
+#define __mem_cgroup_uncharge_kmem(b)
 #endif /* CONFIG_CGROUP_MEM_RES_CTLR_KMEM */
+static __always_inline struct kmem_cache *
+mem_cgroup_get_kmem_cache(struct kmem_cache *cachep, gfp_t gfp)
+{
+ if (mem_cgroup_kmem_on)
+  return __mem_cgroup_get_kmem_cache(cachep, gfp);
+ return cachep;
+}
+
+static __always_inline bool
+mem_cgroup_charge_slab(struct kmem_cache *cachep, gfp_t gfp, size_t size)
+{
+ if (mem_cgroup_kmem_on)
+  return __mem_cgroup_charge_slab(cachep, gfp, size);
+ return true;
+}
+
+static __always_inline void
+mem_cgroup_uncharge_slab(struct kmem_cache *cachep, size_t size)
+{
+ if (mem_cgroup_kmem_on)
+  __mem_cgroup_uncharge_slab(cachep, size);
+}
+
+static __always_inline
+bool mem_cgroup_charge_kmem(gfp_t gfp, size_t size)
+{
+ if (mem_cgroup_kmem_on)
+  return __mem_cgroup_charge_kmem(gfp, size);
```

```
+ return true;
+}
+
+static __always_inline
+void mem_cgroup_uncharge_kmem(size_t size)
+{
+ if (mem_cgroup_kmem_on)
+  __mem_cgroup_uncharge_kmem(size);
+}
 #endif /* _LINUX_MEMCONTROL_H */

diff --git a/init/Kconfig b/init/Kconfig
index 72f33fa..071b7e3 100644
--- a/init/Kconfig
+++ b/init/Kconfig
@@ -696,7 +696,7 @@ config CGROUP_MEM_RES_CTLR_SWAP_ENABLED
    then swapaccount=0 does the trick).
 config CGROUP_MEM_RES_CTLR_KMEM
  bool "Memory Resource Controller Kernel Memory accounting (EXPERIMENTAL)"
- depends on CGROUP_MEM_RES_CTLR && EXPERIMENTAL
+ depends on CGROUP_MEM_RES_CTLR && EXPERIMENTAL && !SLOB
  default n
  help
    The Kernel Memory extension for Memory Resource Controller can limit
diff --git a/mm/memcontrol.c b/mm/memcontrol.c
index e881d83..ae61e99 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
@@ -321,6 +325,11 @@ struct mem_cgroup {
 #ifdef CONFIG_INET
  struct tcp_memcontrol tcp_mem;
 #endif
+
+#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
+ /* Slab accounting */
+ struct kmem_cache *slabs[MAX_KMEM_CACHE_TYPES];
+#endif
```

```
 };

 int memcg_css_id(struct mem_cgroup *memcg)
@@ -414,6 +423,9 @@ static void mem_cgroup_put(struct mem_cgroup *memcg);
 #include <net/ip.h>

 static bool mem_cgroup_is_root(struct mem_cgroup *memcg);
+static int memcg_charge_kmem(struct mem_cgroup *memcg, gfp_t gfp, s64 delta);
+static void memcg_uncharge_kmem(struct mem_cgroup *memcg, s64 delta);
+
 void sock_update_memcg(struct sock *sk)
 {
  if (mem_cgroup_sockets_enabled) {
@@ -513,6 +525,13 @@ char *mem_cgroup_cache_name(struct mem_cgroup *memcg, struct
kmem_cache *cachep)

 /* Bitmap used for allocating the cache id numbers. */
 static DECLARE_BITMAP(cache_types, MAX_KMEM_CACHE_TYPES);
+static DEFINE_MUTEX(memcg_cache_mutex);
+
+static inline bool mem_cgroup_kmem_enabled(struct mem_cgroup *memcg)
+{
+ return !mem_cgroup_disabled() && memcg &&
+        !mem_cgroup_is_root(memcg) && memcg->kmem_accounted;
+}

 void mem_cgroup_register_cache(struct mem_cgroup *memcg,
          struct kmem_cache *cachep)
@@ -534,6 +553,300 @@ void mem_cgroup_release_cache(struct kmem_cache *cachep)
 {
  __clear_bit(cachep->memcg_params.id, cache_types);
 }
+
+static struct kmem_cache *memcg_create_kmem_cache(struct mem_cgroup *memcg,
+        struct kmem_cache *cachep)
+{
+ struct kmem_cache *new_cachep;
+ int idx;
+
+ BUG_ON(!mem_cgroup_kmem_enabled(memcg));
+
+ idx = cachep->memcg_params.id;
+
+ mutex_lock(&memcg_cache_mutex);
+ new_cachep = memcg->slabs[idx];
+ if (new_cachep)
+  goto out;
+
```

```
+ new_cachep = kmem_cache_dup(memcg, cachep);
+
+ if (new_cachep == NULL) {
+  new_cachep = cachep;
+  goto out;
+ }
+
+ mem_cgroup_get(memcg);
+ memcg->slabs[idx] = new_cachep;
+ new_cachep->memcg_params.memcg = memcg;
+out:
+ mutex_unlock(&memcg_cache_mutex);
+ return new_cachep;
+}
+
+struct create_work {
+ struct mem_cgroup *memcg;
+ struct kmem_cache *cachep;
+ struct list_head list;
+};
+
+/* Use a single spinlock for destruction and creation, not a frequent op */
+static DEFINE_SPINLOCK(cache_queue_lock);
+static LIST_HEAD(create_queue);
+static LIST_HEAD(destroyed_caches);
+
+static void kmem_cache_destroy_work_func(struct work_struct *w)
+{
+ struct kmem_cache *cachep;
+ char *name;
+
+ spin_lock_irq(&cache_queue_lock);
+ while (!list_empty(&destroyed_caches)) {
+  cachep = container_of(list_first_entry(&destroyed_caches,
+      struct mem_cgroup_cache_params, destroyed_list), struct
+      kmem_cache, memcg_params);
+  name = (char *)cachep->name;
+  list_del(&cachep->memcg_params.destroyed_list);
+  spin_unlock_irq(&cache_queue_lock);
+  synchronize_rcu();
+  kmem_cache_destroy(cachep);
+  kfree(name);
+  spin_lock_irq(&cache_queue_lock);
+ }
+ spin_unlock_irq(&cache_queue_lock);
+}
+static DECLARE_WORK(kmem_cache_destroy_work, kmem_cache_destroy_work_func);
+
```

```
+void mem_cgroup_destroy_cache(struct kmem_cache *cachep)
+{
+ unsigned long flags;
+
+ BUG_ON(cachep->memcg_params.id != -1);
+
+ /*
+  * We have to defer the actual destroying to a workqueue, because
+  * we might currently be in a context that cannot sleep.
+  */
+ spin_lock_irqsave(&cache_queue_lock, flags);
+ list_add(&cachep->memcg_params.destroyed_list, &destroyed_caches);
+ spin_unlock_irqrestore(&cache_queue_lock, flags);
+
+ schedule_work(&kmem_cache_destroy_work);
+}
+
+
+/*
+ * Flush the queue of kmem_caches to create, because we're creating a cgroup.
+ *
+ * We might end up flushing other cgroups' creation requests as well, but
+ * they will just get queued again next time someone tries to make a slab
+ * allocation for them.
+ */
+void mem_cgroup_flush_cache_create_queue(void)
+{
+ struct create_work *cw, *tmp;
+ unsigned long flags;
+
+ spin_lock_irqsave(&cache_queue_lock, flags);
+ list_for_each_entry_safe(cw, tmp, &create_queue, list) {
+  list_del(&cw->list);
+  kfree(cw);
+ }
+ spin_unlock_irqrestore(&cache_queue_lock, flags);
+}
+
+static void memcg_create_cache_work_func(struct work_struct *w)
+{
+ struct kmem_cache *cachep;
+ struct create_work *cw;
+
+ spin_lock_irq(&cache_queue_lock);
+ while (!list_empty(&create_queue)) {
+  cw = list_first_entry(&create_queue, struct create_work, list);
+  list_del(&cw->list);
+  spin_unlock_irq(&cache_queue_lock);
```

```
+  cachep = memcg_create_kmem_cache(cw->memcg, cw->cachep);
+  if (cachep == NULL)
+   printk(KERN_ALERT
+   "%s: Couldn't create memcg-cache for %s memcg %s\n",
+   __func__, cw->cachep->name,
+   cw->memcg->css.cgroup->dentry->d_name.name);
+  /* Drop the reference gotten when we enqueued. */
+  css_put(&cw->memcg->css);
+  kfree(cw);
+  spin_lock_irq(&cache_queue_lock);
+ }
+ spin_unlock_irq(&cache_queue_lock);
+}
+
+static DECLARE_WORK(memcg_create_cache_work, memcg_create_cache_work_func);
+
+/*
+ * Enqueue the creation of a per-memcg kmem_cache.
+ * Called with rcu_read_lock.
+ */
+static void memcg_create_cache_enqueue(struct mem_cgroup *memcg,
+          struct kmem_cache *cachep)
+{
+ struct create_work *cw;
+ unsigned long flags;
+
+ spin_lock_irqsave(&cache_queue_lock, flags);
+ list_for_each_entry(cw, &create_queue, list) {
+  if (cw->memcg == memcg && cw->cachep == cachep) {
+   spin_unlock_irqrestore(&cache_queue_lock, flags);
+   return;
+  }
+ }
+ spin_unlock_irqrestore(&cache_queue_lock, flags);
+
+ /* The corresponding put will be done in the workqueue. */
+ if (!css_tryget(&memcg->css))
+  return;
+
+ cw = kmalloc_no_account(sizeof(struct create_work), GFP_NOWAIT);
+ if (cw == NULL) {
+  css_put(&memcg->css);
+  return;
+ }
+
+ cw->memcg = memcg;
+ cw->cachep = cachep;
+ spin_lock_irqsave(&cache_queue_lock, flags);
```

```
+ list_add_tail(&cw->list, &create_queue);
+ spin_unlock_irqrestore(&cache_queue_lock, flags);
+
+ schedule_work(&memcg_create_cache_work);
+}
+
+/*
+ * Return the kmem_cache we're supposed to use for a slab allocation.
+ * If we are in interrupt context or otherwise have an allocation that
+ * can't fail, we return the original cache.
+ * Otherwise, we will try to use the current memcg's version of the cache.
+ *
+ * If the cache does not exist yet, if we are the first user of it,
+ * we either create it immediately, if possible, or create it asynchronously
+ * in a workqueue.
+ * In the latter case, we will let the current allocation go through with
+ * the original cache.
+ *
+ * This function returns with rcu_read_lock() held.
+ */
+struct kmem_cache *__mem_cgroup_get_kmem_cache(struct kmem_cache *cachep,
+          gfp_t gfp)
+{
+ struct mem_cgroup *memcg;
+ int idx;
+
+ gfp |=  cachep->allocflags;
+
+ if ((current->mm == NULL))
+  return cachep;
+
+ if (cachep->memcg_params.memcg)
+  return cachep;
+
+ idx = cachep->memcg_params.id;
+ VM_BUG_ON(idx == -1);
+
+ memcg = mem_cgroup_from_task(current);
+ if (!mem_cgroup_kmem_enabled(memcg))
+  return cachep;
+
+ if (rcu_access_pointer(memcg->slabs[idx]) == NULL) {
+  memcg_create_cache_enqueue(memcg, cachep);
+  return cachep;
+ }
+
+ return rcu_dereference(memcg->slabs[idx]);
+}
```

```
+EXPORT_SYMBOL(__mem_cgroup_get_kmem_cache);
+
+void mem_cgroup_remove_child_kmem_cache(struct kmem_cache *cachep, int id)
+{
+ rcu_assign_pointer(cachep->memcg_params.memcg->slabs[id], NULL);
+}
+
+bool __mem_cgroup_charge_kmem(gfp_t gfp, size_t size)
+{
+ struct mem_cgroup *memcg;
+ bool ret = true;
+
+ rcu_read_lock();
+ memcg = mem_cgroup_from_task(current);
+
+ if (!mem_cgroup_kmem_enabled(memcg))
+  goto out;
+
+ mem_cgroup_get(memcg);
+ ret = memcg_charge_kmem(memcg, gfp, size) == 0;
+ if (ret)
+  mem_cgroup_put(memcg);
+out:
+ rcu_read_unlock();
+ return ret;
+}
+EXPORT_SYMBOL(__mem_cgroup_charge_kmem);
+
+void __mem_cgroup_uncharge_kmem(size_t size)
+{
+ struct mem_cgroup *memcg;
+
+ rcu_read_lock();
+ memcg = mem_cgroup_from_task(current);
+
+ if (!mem_cgroup_kmem_enabled(memcg))
+  goto out;
+
+ mem_cgroup_put(memcg);
+ memcg_uncharge_kmem(memcg, size);
+out:
+ rcu_read_unlock();
+}
+EXPORT_SYMBOL(__mem_cgroup_uncharge_kmem);
+
+bool __mem_cgroup_charge_slab(struct kmem_cache *cachep, gfp_t gfp, size_t size)
+{
+ struct mem_cgroup *memcg;
```

```
+ bool ret = true;
+
+ rcu_read_lock();
+ memcg = cachep->memcg_params.memcg;
+ if (!mem_cgroup_kmem_enabled(memcg))
+  goto out;
+
+ ret = memcg_charge_kmem(memcg, gfp, size) == 0;
+out:
+ rcu_read_unlock();
+ return ret;
+}
+EXPORT_SYMBOL(__mem_cgroup_charge_slab);
+
+void __mem_cgroup_uncharge_slab(struct kmem_cache *cachep, size_t size)
+{
+ struct mem_cgroup *memcg;
+
+ rcu_read_lock();
+ memcg = cachep->memcg_params.memcg;
+
+ if (!mem_cgroup_kmem_enabled(memcg)) {
+  rcu_read_unlock();
+  return;
+ }
+ rcu_read_unlock();
+
+ memcg_uncharge_kmem(memcg, size);
+}
+EXPORT_SYMBOL(__mem_cgroup_uncharge_slab);
+
+static void memcg_slab_init(struct mem_cgroup *memcg)
+{
+ int i;
+
+ for (i = 0; i < MAX_KMEM_CACHE_TYPES; i++)
+  rcu_assign_pointer(memcg->slabs[i], NULL);
+}
 #endif /* CONFIG_CGROUP_MEM_RES_CTLR_KMEM */

 static void drain_all_stock_async(struct mem_cgroup *memcg);
@@ -4790,7 +5103,11 @@ static struct cftype kmem_cgroup_files[] = {

 static int memcg_init_kmem(struct mem_cgroup *memcg, struct cgroup_subsys *ss)
 {
- return mem_cgroup_sockets_init(memcg, ss);
+ int ret = mem_cgroup_sockets_init(memcg, ss);
+
```

```
+ if (!ret)
+  memcg_slab_init(memcg);
+ return ret;
 };

 static void kmem_cgroup_destroy(struct mem_cgroup *memcg)
@@ -5805,3 +6122,57 @@ static int __init enable_swap_account(char *s)
 __setup("swapaccount=", enable_swap_account);

 #endif
+
+#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
+int memcg_charge_kmem(struct mem_cgroup *memcg, gfp_t gfp, s64 delta)
+{
+ struct res_counter *fail_res;
+ struct mem_cgroup *_memcg;
+ int may_oom, ret;
+ bool nofail = false;
+
+ may_oom = (gfp & __GFP_WAIT) && (gfp & __GFP_FS) &&
+     !(gfp & __GFP_NORETRY);
+
+ ret = 0;
+
+ if (!memcg)
+  return ret;
+
+ _memcg = memcg;
+ ret = __mem_cgroup_try_charge(NULL, gfp, delta / PAGE_SIZE,
+     &_memcg, may_oom);
+ if (ret == -ENOMEM)
+  return ret;
+ else if ((ret == -EINTR) || (ret && (gfp & __GFP_NOFAIL)))  {
+  nofail = true;
+  /*
+   * __mem_cgroup_try_charge() chose 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, force the
+   * change, going above the limit if needed.
+   */
+  res_counter_charge_nofail(&memcg->res, delta, &fail_res);
+ }
+
+ if (nofail)
+  res_counter_charge_nofail(&memcg->kmem, delta, &fail_res);
+ else
+  ret = res_counter_charge(&memcg->kmem, delta, &fail_res);
```

```
+
+ if (ret)
+  res_counter_uncharge(&memcg->res, delta);
+
+ return ret;
+}
+
+void memcg_uncharge_kmem(struct mem_cgroup *memcg, s64 delta)
+{
+ if (!memcg)
+  return;
+
+ res_counter_uncharge(&memcg->kmem, delta);
+ res_counter_uncharge(&memcg->res, delta);
+}
+#endif /* CONFIG_CGROUP_MEM_RES_CTLR_KMEM */
--
1.7.7.6
```

Subject: [PATCH 18/23] slub: charge allocation to a memcg
Posted by Glauber Costa on Sun, 22 Apr 2012 23:53:35 GMT
View Forum Message <> Reply to Message

This patch charges allocation of a slab object to a particular
memcg.

The cache is selected with mem_cgroup_get_kmem_cache(),
which is the biggest overhead we pay here, because
it happens at all allocations. However, other than forcing
a function call, this function is not very expensive, and
try to return as soon as we realize we are not a memcg cache.

The charge/uncharge functions are heavier, but are only called
for new page allocations.

The kmalloc_no_account variant is patched so the base
function is used and we don't even try to do cache
selection.

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>
CC: Suleiman Souhlal <suleiman@google.com>

```
 include/linux/slub_def.h |   32 ++++++++++--
 mm/slub.c                |  124 +++++++++++++++++++++++++++++++++++++++++-----
 2 files changed, 138 insertions(+), 18 deletions(-)

diff --git a/include/linux/slub_def.h b/include/linux/slub_def.h
index 9a8000a..e75efcb 100644
--- a/include/linux/slub_def.h
+++ b/include/linux/slub_def.h
@@ -13,6 +13,7 @@
 #include <linux/kobject.h>

 #include <linux/kmemleak.h>
+#include <linux/memcontrol.h>

 enum stat_item {
  ALLOC_FASTPATH,  /* Allocation from cpu slab */
@@ -210,14 +211,21 @@ static __always_inline int kmalloc_index(size_t size)
  * This ought to end up with a global pointer to the right cache
  * in kmalloc_caches.
  */
-static __always_inline struct kmem_cache *kmalloc_slab(size_t size)
+static __always_inline struct kmem_cache *kmalloc_slab(gfp_t flags, size_t size)
 {
+ struct kmem_cache *s;
  int index = kmalloc_index(size);

  if (index == 0)
   return NULL;

- return kmalloc_caches[index];
+ s = kmalloc_caches[index];
+
+ rcu_read_lock();
+ s = mem_cgroup_get_kmem_cache(s, flags);
+ rcu_read_unlock();
+
+ return s;
 }

 void *kmem_cache_alloc(struct kmem_cache *, gfp_t);
@@ -225,13 +233,27 @@ void *kmalloc_no_account(size_t size, gfp_t);
 void *__kmalloc(size_t size, gfp_t flags);

 static __always_inline void *
-kmalloc_order(size_t size, gfp_t flags, unsigned int order)
+kmalloc_order_base(size_t size, gfp_t flags, unsigned int order)
 {
  void *ret = (void *) __get_free_pages(flags | __GFP_COMP, order);
```

```
  kmemleak_alloc(ret, size, 1, flags);
  return ret;
 }

+static __always_inline void *
+kmalloc_order(size_t size, gfp_t flags, unsigned int order)
+{
+ void *ret = NULL;
+
+ if (!mem_cgroup_charge_kmem(flags, size))
+  return NULL;
+
+ ret = kmalloc_order_base(size, flags, order);
+ if (!ret)
+  mem_cgroup_uncharge_kmem((1 << order) << PAGE_SHIFT);
+ return ret;
+}
+
 /**
  * Calling this on allocated memory will check that the memory
  * is expected to be in use, and print warnings if not.
@@ -276,7 +298,7 @@ static __always_inline void *kmalloc(size_t size, gfp_t flags)
    return kmalloc_large(size, flags);

   if (!(flags & SLUB_DMA)) {
-   struct kmem_cache *s = kmalloc_slab(size);
+   struct kmem_cache *s = kmalloc_slab(flags, size);

    if (!s)
     return ZERO_SIZE_PTR;
@@ -309,7 +331,7 @@ static __always_inline void *kmalloc_node(size_t size, gfp_t flags, int
node)
 {
  if (__builtin_constant_p(size) &&
   size <= SLUB_MAX_SIZE && !(flags & SLUB_DMA)) {
-   struct kmem_cache *s = kmalloc_slab(size);
+   struct kmem_cache *s = kmalloc_slab(flags, size);

   if (!s)
    return ZERO_SIZE_PTR;
diff --git a/mm/slub.c b/mm/slub.c
index d754b06..9b22139 100644
--- a/mm/slub.c
+++ b/mm/slub.c
@@ -1283,11 +1283,17 @@ static inline struct page *alloc_slab_page(gfp_t flags, int node,
  return alloc_pages_exact_node(node, flags, order);
 }
```

```
+static inline unsigned long size_in_bytes(unsigned int order)
+{
+ return (1 << order) << PAGE_SHIFT;
+}
+
 static struct page *allocate_slab(struct kmem_cache *s, gfp_t flags, int node)
 {
- struct page *page;
+ struct page *page = NULL;
  struct kmem_cache_order_objects oo = s->oo;
  gfp_t alloc_gfp;
+ unsigned int memcg_allowed = oo_order(oo);

  flags &= gfp_allowed_mask;

@@ -1296,13 +1302,29 @@ static struct page *allocate_slab(struct kmem_cache *s, gfp_t flags,
int node)

  flags |= s->allocflags;

- /*
-  * Let the initial higher-order allocation fail under memory pressure
-  * so we fall-back to the minimum order allocation.
-  */
- alloc_gfp = (flags | __GFP_NOWARN | __GFP_NORETRY) & ~__GFP_NOFAIL;
+ memcg_allowed = oo_order(oo);
+ if (!mem_cgroup_charge_slab(s, flags, size_in_bytes(memcg_allowed))) {
+
+  memcg_allowed = oo_order(s->min);
+  if (!mem_cgroup_charge_slab(s, flags,
+         size_in_bytes(memcg_allowed))) {
+   if (flags & __GFP_WAIT)
+    local_irq_disable();
+   return NULL;
+  }
+ }
+
+ if (memcg_allowed == oo_order(oo)) {
+  /*
+   * Let the initial higher-order allocation fail under memory
+   * pressure so we fall-back to the minimum order allocation.
+   */
+  alloc_gfp = (flags | __GFP_NOWARN | __GFP_NORETRY) &
+        ~__GFP_NOFAIL;
+
+  page = alloc_slab_page(alloc_gfp, node, oo);
+ }
```

```
- page = alloc_slab_page(alloc_gfp, node, oo);
  if (unlikely(!page)) {
   oo = s->min;
   /*
@@ -1313,13 +1335,23 @@ static struct page *allocate_slab(struct kmem_cache *s, gfp_t flags,
int node)

   if (page)
    stat(s, ORDER_FALLBACK);
+  /*
+   * We reserved more than we used, time to give it back
+   */
+  if (page && memcg_allowed != oo_order(oo)) {
+   unsigned long delta;
+   delta = memcg_allowed - oo_order(oo);
+   mem_cgroup_uncharge_slab(s, size_in_bytes(delta));
+  }
  }

  if (flags & __GFP_WAIT)
   local_irq_disable();

- if (!page)
+ if (!page) {
+  mem_cgroup_uncharge_slab(s, size_in_bytes(memcg_allowed));
   return NULL;
+ }

  if (kmemcheck_enabled
   && !(s->flags & (SLAB_NOTRACK | DEBUG_DEFAULT_FLAGS))) {
@@ -1393,6 +1425,24 @@ out:
  return page;
 }

+#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
+static unsigned long slab_nr_pages(struct kmem_cache *s)
+{
+ int node;
+ unsigned long nr_slabs = 0;
+
+ for_each_online_node(node) {
+  struct kmem_cache_node *n = get_node(s, node);
+
+  if (!n)
+   continue;
+  nr_slabs += atomic_long_read(&n->nr_slabs);
+ }
+
```

```
+ return nr_slabs << oo_order(s->oo);
+}
+#endif
+
 static void __free_slab(struct kmem_cache *s, struct page *page)
 {
  int order = compound_order(page);
@@ -1419,6 +1469,12 @@ static void __free_slab(struct kmem_cache *s, struct page *page)
  if (current->reclaim_state)
   current->reclaim_state->reclaimed_slab += pages;
  __free_pages(page, order);
+
+ mem_cgroup_uncharge_slab(s, (1 << order) << PAGE_SHIFT);
+#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
+ if (s->memcg_params.memcg && (slab_nr_pages(s) == 0))
+  mem_cgroup_destroy_cache(s);
+#endif
 }

 #define need_reserve_slab_rcu      \
@@ -2300,8 +2356,9 @@ new_slab:
  *
  * Otherwise we can simply pick the next object from the lockless free list.
  */
-static __always_inline void *slab_alloc(struct kmem_cache *s,
-  gfp_t gfpflags, int node, unsigned long addr)
+static __always_inline void *slab_alloc_base(struct kmem_cache *s,
+          gfp_t gfpflags, int node,
+          unsigned long addr)
 {
  void **object;
  struct kmem_cache_cpu *c;
@@ -2369,6 +2426,24 @@ redo:
  return object;
 }

+static __always_inline void *slab_alloc(struct kmem_cache *s,
+  gfp_t gfpflags, int node, unsigned long addr)
+{
+
+ if (slab_pre_alloc_hook(s, gfpflags))
+  return NULL;
+
+ if (in_interrupt() || (current == NULL) || (gfpflags & __GFP_NOFAIL))
+  goto kernel_alloc;
+
+ rcu_read_lock();
+ s = mem_cgroup_get_kmem_cache(s, gfpflags);
```

```
+ rcu_read_unlock();
+
+kernel_alloc:
+ return slab_alloc_base(s, gfpflags, node, addr);
+}
+
 void *kmem_cache_alloc(struct kmem_cache *s, gfp_t gfpflags)
 {
  void *ret = slab_alloc(s, gfpflags, NUMA_NO_NODE, _RET_IP_);
@@ -3194,6 +3269,13 @@ void kmem_cache_destroy(struct kmem_cache *s)
  s->refcount--;
  if (!s->refcount) {
   list_del(&s->list);
+#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
+  /* Not a memcg cache */
+  if (s->memcg_params.id != -1) {
+   mem_cgroup_release_cache(s);
+   mem_cgroup_flush_cache_create_queue();
+  }
+#endif
   up_write(&slub_lock);
   if (kmem_cache_close(s)) {
    printk(KERN_ERR "SLUB %s: %s called for cache that "
@@ -3273,6 +3355,7 @@ static struct kmem_cache *__init create_kmalloc_cache(const char
*name,
   goto panic;

  list_add(&s->list, &slab_caches);
+ mem_cgroup_register_cache(NULL, s);
  return s;

 panic:
@@ -3364,15 +3447,21 @@ void *kmalloc_no_account(size_t size, gfp_t flags)
  struct kmem_cache *s;
  void *ret;

- if (unlikely(size > SLUB_MAX_SIZE))
-  return kmalloc_large(size, flags);
+ if (unlikely(size > SLUB_MAX_SIZE)) {
+  unsigned int order = get_order(size);
+  ret = kmalloc_order_base(size, flags, order);
+#ifdef CONFIG_TRACING
+  trace_kmalloc(_RET_IP_, ret, size, PAGE_SIZE << order, flags);
+#endif
+  return ret;
+ }

  s = get_slab(size, flags);
```

```
  if (unlikely(ZERO_OR_NULL_PTR(s)))
   return s;

- ret = slab_alloc(s, flags, NUMA_NO_NODE, _RET_IP_);
+ ret = slab_alloc_base(s, flags, NUMA_NO_NODE, _RET_IP_);

  trace_kmalloc(_RET_IP_, ret, size, s->size, flags);

@@ -3387,10 +3476,17 @@ static void *kmalloc_large_node(size_t size, gfp_t flags, int node)
  void *ptr = NULL;

  flags |= __GFP_COMP | __GFP_NOTRACK;
+
+ if (!mem_cgroup_charge_kmem(flags, size))
+  goto out;
+
  page = alloc_pages_node(node, flags, get_order(size));
  if (page)
   ptr = page_address(page);
+ else
+  mem_cgroup_uncharge_kmem(size);

+out:
  kmemleak_alloc(ptr, size, 1, flags);
  return ptr;
 }
@@ -3938,8 +4034,10 @@ static struct kmem_cache *find_mergeable(struct mem_cgroup
*memcg, size_t size,
   if (s->size - size >= sizeof(void *))
    continue;

+#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
   if (memcg && s->memcg_params.memcg != memcg)
    continue;
+#endif

   return s;
  }
--
1.7.7.6
```

Subject: [PATCH 19/23] slab: per-memcg accounting of slab caches
Posted by Glauber Costa on Sun, 22 Apr 2012 23:53:36 GMT
View Forum Message <> Reply to Message

This patch charges allocation of a slab object to a particular

memcg.

The cache is selected with mem_cgroup_get_kmem_cache(),
which is the biggest overhead we pay here, because
it happens at all allocations. However, other than forcing
a function call, this function is not very expensive, and
try to return as soon as we realize we are not a memcg cache.

The charge/uncharge functions are heavier, but are only called
for new page allocations.

Code is heavily inspired by Suleiman's, with adaptations to
the patchset and minor simplifications by me.

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>
CC: Suleiman Souhlal <suleiman@google.com>
---
 include/linux/slab_def.h | 66 ++++++++++++++++++++++++++-
 mm/slab.c | 105 ++----
 2 files changed, 162 insertions(+), 9 deletions(-)

diff --git a/include/linux/slab_def.h b/include/linux/slab_def.h
index 54d25d7..c4f7e45 100644
--- a/include/linux/slab_def.h
+++ b/include/linux/slab_def.h
@@ -51,7 +51,7 @@ struct kmem_cache {
  void (*ctor)(void *obj);

 /* 4) cache creation/removal */
- const char *name;
+ char *name;
  struct list_head next;

 /* 5) statistics */
@@ -219,4 +219,68 @@ found:

 #endif /* CONFIG_NUMA */

+#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
+
+void kmem_cache_drop_ref(struct kmem_cache *cachep);
+
+static inline void

```
+kmem_cache_get_ref(struct kmem_cache *cachep)
+{
+ if (cachep->memcg_params.id == -1 &&
+     unlikely(!atomic_add_unless(&cachep->memcg_params.refcnt, 1, 0)))
+  BUG();
+}
+
+static inline void
+mem_cgroup_put_kmem_cache(struct kmem_cache *cachep)
+{
+ rcu_read_unlock();
+}
+
+static inline void
+mem_cgroup_kmem_cache_prepare_sleep(struct kmem_cache *cachep)
+{
+ /*
+  * Make sure the cache doesn't get freed while we have interrupts
+  * enabled.
+  */
+ kmem_cache_get_ref(cachep);
+ rcu_read_unlock();
+}
+
+static inline void
+mem_cgroup_kmem_cache_finish_sleep(struct kmem_cache *cachep)
+{
+ rcu_read_lock();
+ kmem_cache_drop_ref(cachep);
+}
+
+#else /* CONFIG_CGROUP_MEM_RES_CTLR_KMEM */
+
+static inline void
+kmem_cache_get_ref(struct kmem_cache *cachep)
+{
+}
+
+static inline void
+kmem_cache_drop_ref(struct kmem_cache *cachep)
+{
+}
+
+static inline void
+mem_cgroup_put_kmem_cache(struct kmem_cache *cachep)
+{
+}
+
```

```
+static inline void
+mem_cgroup_kmem_cache_prepare_sleep(struct kmem_cache *cachep)
+{
+}
+
+static inline void
+mem_cgroup_kmem_cache_finish_sleep(struct kmem_cache *cachep)
+{
+}
+#endif /* CONFIG_CGROUP_MEM_RES_CTLR_KMEM */
+
 #endif /* _LINUX_SLAB_DEF_H */
diff --git a/mm/slab.c b/mm/slab.c
index 13948c3..ac0916b 100644
--- a/mm/slab.c
+++ b/mm/slab.c
@@ -1818,20 +1818,28 @@ static void *kmem_getpages(struct kmem_cache *cachep, gfp_t
flags, int nodeid)
  if (cachep->flags & SLAB_RECLAIM_ACCOUNT)
   flags |= __GFP_RECLAIMABLE;

+ nr_pages = (1 << cachep->gfporder);
+ if (!mem_cgroup_charge_slab(cachep, flags, nr_pages * PAGE_SIZE))
+  return NULL;
+
  page = alloc_pages_exact_node(nodeid, flags | __GFP_NOTRACK, cachep->gfporder);
  if (!page) {
   if (!(flags & __GFP_NOWARN) && printk_ratelimit())
    slab_out_of_memory(cachep, flags, nodeid);
+
+  mem_cgroup_uncharge_slab(cachep, nr_pages * PAGE_SIZE);
   return NULL;
  }

- nr_pages = (1 << cachep->gfporder);
  if (cachep->flags & SLAB_RECLAIM_ACCOUNT)
   add_zone_page_state(page_zone(page),
    NR_SLAB_RECLAIMABLE, nr_pages);
  else
   add_zone_page_state(page_zone(page),
    NR_SLAB_UNRECLAIMABLE, nr_pages);
+
+ kmem_cache_get_ref(cachep);
+
  for (i = 0; i < nr_pages; i++)
   __SetPageSlab(page + i);

@@ -1864,6 +1872,8 @@ static void kmem_freepages(struct kmem_cache *cachep, void *addr)
```

```
  else
   sub_zone_page_state(page_zone(page),
     NR_SLAB_UNRECLAIMABLE, nr_freed);
+ mem_cgroup_uncharge_slab(cachep, i * PAGE_SIZE);
+ kmem_cache_drop_ref(cachep);
  while (i--) {
   BUG_ON(!PageSlab(page));
   __ClearPageSlab(page);
@@ -2823,12 +2833,28 @@ void kmem_cache_destroy(struct kmem_cache *cachep)
  if (unlikely(cachep->flags & SLAB_DESTROY_BY_RCU))
   rcu_barrier();

+#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
+ /* Not a memcg cache */
+ if (cachep->memcg_params.id != -1) {
+  mem_cgroup_release_cache(cachep);
+  mem_cgroup_flush_cache_create_queue();
+ }
+#endif
  __kmem_cache_destroy(cachep);
  mutex_unlock(&cache_chain_mutex);
  put_online_cpus();
 }
 EXPORT_SYMBOL(kmem_cache_destroy);

+#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
+void kmem_cache_drop_ref(struct kmem_cache *cachep)
+{
+ if (cachep->memcg_params.id == -1 &&
+     unlikely(atomic_dec_and_test(&cachep->memcg_params.refcnt)))
+  mem_cgroup_destroy_cache(cachep);
+}
+#endif /* CONFIG_CGROUP_MEM_RES_CTLR_KMEM */
+
 /*
  * Get the memory for a slab management obj.
  * For a slab cache when the slab descriptor is off-slab, slab descriptors
@@ -3028,8 +3054,10 @@ static int cache_grow(struct kmem_cache *cachep,

  offset *= cachep->colour_off;

- if (local_flags & __GFP_WAIT)
+ if (local_flags & __GFP_WAIT) {
   local_irq_enable();
+  mem_cgroup_kmem_cache_prepare_sleep(cachep);
+ }

  /*
```

```
   * The test for missing atomic flag is performed here, rather than
@@ -3058,8 +3086,10 @@ static int cache_grow(struct kmem_cache *cachep,

  cache_init_objs(cachep, slabp);

- if (local_flags & __GFP_WAIT)
+ if (local_flags & __GFP_WAIT) {
    local_irq_disable();
+  mem_cgroup_kmem_cache_finish_sleep(cachep);
+ }
  check_irq_off();
  spin_lock(&l3->list_lock);

@@ -3072,8 +3102,10 @@ static int cache_grow(struct kmem_cache *cachep,
 opps1:
  kmem_freepages(cachep, objp);
 failed:
- if (local_flags & __GFP_WAIT)
+ if (local_flags & __GFP_WAIT) {
    local_irq_disable();
+  mem_cgroup_kmem_cache_finish_sleep(cachep);
+ }
  return 0;
 }

@@ -3834,11 +3866,15 @@ static inline void __cache_free(struct kmem_cache *cachep, void
*objp,
  */
 void *kmem_cache_alloc(struct kmem_cache *cachep, gfp_t flags)
 {
- void *ret = __cache_alloc(cachep, flags, __builtin_return_address(0));
+ void *ret;
+
+ rcu_read_lock();
+ cachep = mem_cgroup_get_kmem_cache(cachep, flags);
+ rcu_read_unlock();
+ ret = __cache_alloc(cachep, flags, __builtin_return_address(0));

  trace_kmem_cache_alloc(_RET_IP_, ret,
          obj_size(cachep), cachep->buffer_size, flags);
-
  return ret;
 }
 EXPORT_SYMBOL(kmem_cache_alloc);
@@ -3849,6 +3885,10 @@ kmem_cache_alloc_trace(size_t size, struct kmem_cache *cachep,
gfp_t flags)
 {
  void *ret;
```

```
+ rcu_read_lock();
+ cachep = mem_cgroup_get_kmem_cache(cachep, flags);
+ rcu_read_unlock();
+
  ret = __cache_alloc(cachep, flags, __builtin_return_address(0));

  trace_kmalloc(_RET_IP_, ret,
@@ -3861,13 +3901,17 @@ EXPORT_SYMBOL(kmem_cache_alloc_trace);
 #ifdef CONFIG_NUMA
 void *kmem_cache_alloc_node(struct kmem_cache *cachep, gfp_t flags, int nodeid)
 {
- void *ret = __cache_alloc_node(cachep, flags, nodeid,
+ void *ret;
+
+ rcu_read_lock();
+ cachep = mem_cgroup_get_kmem_cache(cachep, flags);
+ rcu_read_unlock();
+ ret  = __cache_alloc_node(cachep, flags, nodeid,
            __builtin_return_address(0));

  trace_kmem_cache_alloc_node(_RET_IP_, ret,
         obj_size(cachep), cachep->buffer_size,
         flags, nodeid);
-
  return ret;
 }
 EXPORT_SYMBOL(kmem_cache_alloc_node);
@@ -3880,6 +3924,9 @@ void *kmem_cache_alloc_node_trace(size_t size,
 {
  void *ret;

+ rcu_read_lock();
+ cachep = mem_cgroup_get_kmem_cache(cachep, flags);
+ rcu_read_unlock();
  ret = __cache_alloc_node(cachep, flags, nodeid,
        __builtin_return_address(0));
  trace_kmalloc_node(_RET_IP_, ret,
@@ -4011,9 +4058,33 @@ void kmem_cache_free(struct kmem_cache *cachep, void *objp)

  local_irq_save(flags);
  debug_check_no_locks_freed(objp, obj_size(cachep));
+
+#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
+ {
+  struct kmem_cache *actual_cachep;
+
+  actual_cachep = virt_to_cache(objp);
```

```
+  if (actual_cachep != cachep) {
+   VM_BUG_ON(actual_cachep->memcg_params.id != -1);
+   cachep = actual_cachep;
+  }
+  /*
+   * Grab a reference so that the cache is guaranteed to stay
+   * around.
+   * If we are freeing the last object of a dead memcg cache,
+   * the kmem_cache_drop_ref() at the end of this function
+   * will end up freeing the cache.
+   */
+  kmem_cache_get_ref(cachep);
+ }
+#endif
+
  if (!(cachep->flags & SLAB_DEBUG_OBJECTS))
   debug_check_no_obj_freed(objp, obj_size(cachep));
  __cache_free(cachep, objp, __builtin_return_address(0));
+
+ kmem_cache_drop_ref(cachep);
+
  local_irq_restore(flags);

  trace_kmem_cache_free(_RET_IP_, objp);
@@ -4041,9 +4112,19 @@ void kfree(const void *objp)
  local_irq_save(flags);
  kfree_debugcheck(objp);
  c = virt_to_cache(objp);
+
+ /*
+  * Grab a reference so that the cache is guaranteed to stay around.
+  * If we are freeing the last object of a dead memcg cache, the
+  * kmem_cache_drop_ref() at the end of this function will end up
+  * freeing the cache.
+  */
+ kmem_cache_get_ref(c);
+
  debug_check_no_locks_freed(objp, obj_size(c));
  debug_check_no_obj_freed(objp, obj_size(c));
  __cache_free(c, (void *)objp, __builtin_return_address(0));
+ kmem_cache_drop_ref(c);
  local_irq_restore(flags);
 }
 EXPORT_SYMBOL(kfree);
@@ -4312,6 +4393,13 @@ static void cache_reap(struct work_struct *w)
  list_for_each_entry(searchp, &cache_chain, next) {
   check_irq_on();
```

```
+#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
+  /* For memcg caches, make sure we only reap the active ones. */
+  if (searchp->memcg_params.id == -1 &&
+      !atomic_add_unless(&searchp->memcg_params.refcnt, 1, 0))
+   continue;
+#endif
+
   /*
    * We only take the l3 lock if absolutely necessary and we
    * have established with reasonable certainty that
@@ -4344,6 +4432,7 @@ static void cache_reap(struct work_struct *w)
    STATS_ADD_REAPED(searchp, freed);
   }
 next:
+  kmem_cache_drop_ref(searchp);
   cond_resched();
  }
  check_irq_on();
--
1.7.7.6
```

Subject: [PATCH 20/23] memcg: disable kmem code when not in use.
Posted by Glauber Costa on Sun, 22 Apr 2012 23:53:37 GMT
View Forum Message <> Reply to Message

We can use jump labels to patch the code in or out
when not used.

Because the assignment: memcg->kmem_accounted = true
is done after the jump labels increment, we guarantee
that the root memcg will always be selected until
all call sites are patched (see mem_cgroup_kmem_enabled).
This guarantees that no mischarges are applied.

Jump label decrement happens when the last reference
count from the memcg dies. This will only happen when
the caches are all dead.

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>
CC: Suleiman Souhlal <suleiman@google.com>
---
 include/linux/memcontrol.h | 4 +++-

```
 mm/memcontrol.c           |   21 ++++++++++++++++++++-
 2 files changed, 23 insertions(+), 2 deletions(-)

diff --git a/include/linux/memcontrol.h b/include/linux/memcontrol.h
index c1c1302..25c4324 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/jump_label.h>

 struct mem_cgroup;
 struct page_cgroup;
@@ -460,7 +461,8 @@ void __mem_cgroup_uncharge_kmem(size_t size);
 struct kmem_cache *
 __mem_cgroup_get_kmem_cache(struct kmem_cache *cachep, gfp_t gfp);

-#define mem_cgroup_kmem_on 1
+extern struct static_key mem_cgroup_kmem_enabled_key;
+#define mem_cgroup_kmem_on static_key_false(&mem_cgroup_kmem_enabled_key)

 void mem_cgroup_destroy_cache(struct kmem_cache *cachep);
 #else
diff --git a/mm/memcontrol.c b/mm/memcontrol.c
index ae61e99..547b632 100644
--- a/mm/memcontrol.c
+++ b/mm/memcontrol.c
@@ -422,6 +422,10 @@ static void mem_cgroup_put(struct mem_cgroup *memcg);
 #include <net/sock.h>
 #include <net/ip.h>

+struct static_key mem_cgroup_kmem_enabled_key;
+/* so modules can inline the checks */
+EXPORT_SYMBOL(mem_cgroup_kmem_enabled_key);
+
 static bool mem_cgroup_is_root(struct mem_cgroup *memcg);
 static int memcg_charge_kmem(struct mem_cgroup *memcg, gfp_t gfp, s64 delta);
 static void memcg_uncharge_kmem(struct mem_cgroup *memcg, s64 delta);
@@ -468,6 +472,12 @@ void sock_release_memcg(struct sock *sk)
  }
 }

+static void disarm_static_keys(struct mem_cgroup *memcg)
+{
+ if (memcg->kmem_accounted)
+  static_key_slow_dec(&mem_cgroup_kmem_enabled_key);
```

```
+}
+
 #ifdef CONFIG_INET
 struct cg_proto *tcp_proto_cgroup(struct mem_cgroup *memcg)
 {
@@ -847,6 +857,10 @@ static void memcg_slab_init(struct mem_cgroup *memcg)
  for (i = 0; i < MAX_KMEM_CACHE_TYPES; i++)
   rcu_assign_pointer(memcg->slabs[i], NULL);
 }
+#else
+static inline void disarm_static_keys(struct mem_cgroup *memcg)
+{
+}
 #endif /* CONFIG_CGROUP_MEM_RES_CTLR_KMEM */

 static void drain_all_stock_async(struct mem_cgroup *memcg);
@@ -4366,8 +4380,12 @@ static int mem_cgroup_write(struct cgroup *cont, struct cftype *cft,
     *
     * But it is not worth the trouble
     */
-   if (!memcg->kmem_accounted && val != RESOURCE_MAX)
+   mutex_lock(&set_limit_mutex);
+   if (!memcg->kmem_accounted && val != RESOURCE_MAX) {
+    static_key_slow_inc(&mem_cgroup_kmem_enabled_key);
     memcg->kmem_accounted = true;
+   }
+   mutex_unlock(&set_limit_mutex);
   }
 #endif
   else
@@ -5349,6 +5367,7 @@ static void __mem_cgroup_put(struct mem_cgroup *memcg, int count)
 {
  if (atomic_sub_and_test(count, &memcg->refcnt)) {
   struct mem_cgroup *parent = parent_mem_cgroup(memcg);
+  disarm_static_keys(memcg);
   __mem_cgroup_free(memcg);
   if (parent)
    mem_cgroup_put(parent);
--
1.7.7.6
```

Subject: [PATCH 21/23] memcg: Track all the memcg children of a kmem_cache.
Posted by Glauber Costa on Sun, 22 Apr 2012 23:53:38 GMT
View Forum Message <> Reply to Message

From: Suleiman Souhlal <ssouhlal@FreeBSD.org>

This enables us to remove all the children of a kmem_cache being
destroyed, if for example the kernel module it's being used in
gets unloaded. Otherwise, the children will still point to the
destroyed parent.

We also use this to propagate /proc/slabinfo settings to all
the children of a cache, when, for example, changing its
batchsize.

Signed-off-by: Suleiman Souhlal <suleiman@google.com>
---
 include/linux/slab.h | 1 +
 mm/slab.c | 53 ++---
 2 files changed, 50 insertions(+), 4 deletions(-)

diff --git a/include/linux/slab.h b/include/linux/slab.h
index 909b508..0dc49fa 100644
--- a/include/linux/slab.h
+++ b/include/linux/slab.h
@@ -163,6 +163,7 @@ struct mem_cgroup_cache_params {
  size_t orig_align;
  atomic_t refcnt;

+ struct list_head sibling_list;
 #endif
  struct list_head destroyed_list; /* Used when deleting cpuset cache */
 };
diff --git a/mm/slab.c b/mm/slab.c
index ac0916b..86f2275 100644
--- a/mm/slab.c
+++ b/mm/slab.c
@@ -2561,6 +2561,7 @@ __kmem_cache_create(struct mem_cgroup *memcg, const char
*name, size_t size,
 #ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
  mem_cgroup_register_cache(memcg, cachep);
  atomic_set(&cachep->memcg_params.refcnt, 1);
+ INIT_LIST_HEAD(&cachep->memcg_params.sibling_list);
 #endif

  if (setup_cpu_cache(cachep, gfp)) {
@@ -2628,6 +2629,8 @@ kmem_cache_dup(struct mem_cgroup *memcg, struct kmem_cache
*cachep)
   return NULL;
  }

+ list_add(&new->memcg_params.sibling_list,
+   &cachep->memcg_params.sibling_list);
  if ((cachep->limit != new->limit) ||

```
      (cachep->batchcount != new->batchcount) ||
      (cachep->shared != new->shared))
@@ -2815,6 +2818,29 @@ void kmem_cache_destroy(struct kmem_cache *cachep)
 {
  BUG_ON(!cachep || in_interrupt());

+#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
+ /* Destroy all the children caches if we aren't a memcg cache */
+ if (cachep->memcg_params.id != -1) {
+  struct kmem_cache *c;
+  struct mem_cgroup_cache_params *p, *tmp;
+
+  mutex_lock(&cache_chain_mutex);
+  list_for_each_entry_safe(p, tmp,
+      &cachep->memcg_params.sibling_list, sibling_list) {
+   c = container_of(p, struct kmem_cache, memcg_params);
+   if (c == cachep)
+    continue;
+   mutex_unlock(&cache_chain_mutex);
+   BUG_ON(c->memcg_params.id != -1);
+   mem_cgroup_remove_child_kmem_cache(c,
+       cachep->memcg_params.id);
+   kmem_cache_destroy(c);
+   mutex_lock(&cache_chain_mutex);
+  }
+  mutex_unlock(&cache_chain_mutex);
+ }
+#endif /* CONFIG_CGROUP_MEM_RES_CTLR_KMEM */
+
  /* Find the cache in the chain of caches. */
  get_online_cpus();
  mutex_lock(&cache_chain_mutex);
@@ -2822,6 +2848,9 @@ void kmem_cache_destroy(struct kmem_cache *cachep)
   * the chain is never empty, cache_cache is never destroyed
   */
  list_del(&cachep->next);
+#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
+ list_del(&cachep->memcg_params.sibling_list);
+#endif
  if (__cache_shrink(cachep)) {
   slab_error(cachep, "Can't free all objects");
   list_add(&cachep->next, &cache_chain);
@@ -4644,11 +4673,27 @@ static ssize_t slabinfo_write(struct file *file, const char __user
*buffer,
    if (limit < 1 || batchcount < 1 ||
      batchcount > limit || shared < 0) {
     res = 0;
-   } else {
```

```
-    res = do_tune_cpucache(cachep, limit,
-            batchcount, shared,
-            GFP_KERNEL);
+    break;
    }
+
+   res = do_tune_cpucache(cachep, limit, batchcount,
+       shared, GFP_KERNEL);
+
+#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
+   {
+    struct kmem_cache *c;
+    struct mem_cgroup_cache_params *p;
+
+    list_for_each_entry(p,
+        &cachep->memcg_params.sibling_list,
+        sibling_list) {
+     c = container_of(p, struct kmem_cache,
+         memcg_params);
+     do_tune_cpucache(c, limit, batchcount,
+         shared, GFP_KERNEL);
+    }
+   }
+#endif
    break;
   }
  }
--
1.7.7.6
```

Subject: [PATCH 22/23] memcg: Per-memcg memory.kmem.slabinfo file.
Posted by Glauber Costa on Sun, 22 Apr 2012 23:53:39 GMT
View Forum Message <> Reply to Message

From: Suleiman Souhlal <ssouhlal@FreeBSD.org>

This file shows all the kmem_caches used by a memcg.

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

```
 include/linux/slab.h |    1 +
 mm/memcontrol.c      |   17 ++++++++++
 mm/slab.c            |   88 +++++++++++
```

```
index 0dc49fa..6932205 100644
--- a/include/linux/slab.h
+++ b/include/linux/slab.h
@@ -327,6 +327,7 @@ extern void *__kmalloc_track_caller(size_t, gfp_t, unsigned long);
 extern struct kmem_cache *kmem_cache_dup(struct mem_cgroup *memcg,
       struct kmem_cache *cachep);
 void kmem_cache_drop_ref(struct kmem_cache *cachep);
+int mem_cgroup_slabinfo(struct mem_cgroup *mem, struct seq_file *m);
 #else
 #define MAX_KMEM_CACHE_TYPES 0
 #endif /* CONFIG_CGROUP_MEM_RES_CTLR_KMEM */
diff --git a/mm/memcontrol.c b/mm/memcontrol.c
index 547b632..46ebd11 100644
--- a/mm/memcontrol.c
+++ b/mm/memcontrol.c
@@ -5092,6 +5092,19 @@ static int mem_control_numa_stat_open(struct inode *unused, struct
file *file)
 #endif /* CONFIG_NUMA */

 #ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
+static int mem_cgroup_slabinfo_show(struct cgroup *cgroup, struct cftype *ctf,
+       struct seq_file *m)
+{
+ struct mem_cgroup *mem;
+
+ mem  = mem_cgroup_from_cont(cgroup);
+
+ if (mem == root_mem_cgroup)
+  mem = NULL;
+
+ return mem_cgroup_slabinfo(mem, m);
+}
+
 static struct cftype kmem_cgroup_files[] = {
  {
   .name = "kmem.limit_in_bytes",
@@ -5116,6 +5129,10 @@ static struct cftype kmem_cgroup_files[] = {
   .trigger = mem_cgroup_reset,
   .read = mem_cgroup_read,
  },
+ {
+  .name = "kmem.slabinfo",
+  .read_seq_string = mem_cgroup_slabinfo_show,
+ },
  {},
 };

diff --git a/mm/slab.c b/mm/slab.c
```

```
index 86f2275..3e13fef 100644
--- a/mm/slab.c
+++ b/mm/slab.c
@@ -4518,21 +4519,26 @@ static void s_stop(struct seq_file *m, void *p)
  mutex_unlock(&cache_chain_mutex);
 }

-static int s_show(struct seq_file *m, void *p)
-{
- struct kmem_cache *cachep = list_entry(p, struct kmem_cache, next);
- struct slab *slabp;
+struct slab_counts {
  unsigned long active_objs;
+ unsigned long active_slabs;
+ unsigned long num_slabs;
+ unsigned long free_objects;
+ unsigned long shared_avail;
  unsigned long num_objs;
- unsigned long active_slabs = 0;
- unsigned long num_slabs, free_objects = 0, shared_avail = 0;
- const char *name;
- char *error = NULL;
- int node;
+};
+
+static char *
+get_slab_counts(struct kmem_cache *cachep, struct slab_counts *c)
+{
  struct kmem_list3 *l3;
+ struct slab *slabp;
+ char *error;
+ int node;
+
+ error = NULL;
+ memset(c, 0, sizeof(struct slab_counts));

- active_objs = 0;
- num_slabs = 0;
  for_each_online_node(node) {
   l3 = cachep->nodelists[node];
   if (!l3)
@@ -4544,31 +4550,43 @@ static int s_show(struct seq_file *m, void *p)
   list_for_each_entry(slabp, &l3->slabs_full, list) {
    if (slabp->inuse != cachep->num && !error)
     error = "slabs_full accounting error";
-   active_objs += cachep->num;
-   active_slabs++;
+   c->active_objs += cachep->num;
```

```
+   c->active_slabs++;
    }
    list_for_each_entry(slabp, &l3->slabs_partial, list) {
     if (slabp->inuse == cachep->num && !error)
      error = "slabs_partial inuse accounting error";
     if (!slabp->inuse && !error)
      error = "slabs_partial/inuse accounting error";
-    active_objs += slabp->inuse;
-    active_slabs++;
+    c->active_objs += slabp->inuse;
+    c->active_slabs++;
    }
    list_for_each_entry(slabp, &l3->slabs_free, list) {
     if (slabp->inuse && !error)
      error = "slabs_free/inuse accounting error";
-    num_slabs++;
+    c->num_slabs++;
    }
-   free_objects += l3->free_objects;
+   c->free_objects += l3->free_objects;
    if (l3->shared)
-    shared_avail += l3->shared->avail;
+    c->shared_avail += l3->shared->avail;

    spin_unlock_irq(&l3->list_lock);
   }
- num_slabs += active_slabs;
- num_objs = num_slabs * cachep->num;
- if (num_objs - active_objs != free_objects && !error)
+ c->num_slabs += c->active_slabs;
+ c->num_objs = c->num_slabs * cachep->num;
+
+ return error;
+}
+
+static int s_show(struct seq_file *m, void *p)
+{
+ struct kmem_cache *cachep = list_entry(p, struct kmem_cache, next);
+ struct slab_counts c;
+ const char *name;
+ char *error;
+
+ error = get_slab_counts(cachep, &c);
+ if (c.num_objs - c.active_objs != c.free_objects && !error)
    error = "free_objects accounting error";

   name = cachep->name;
@@ -4576,12 +4594,12 @@ static int s_show(struct seq_file *m, void *p)
```

```
   printk(KERN_ERR "slab: cache %s error: %s\n", name, error);

  seq_printf(m, "%-17s %6lu %6lu %6u %4u %4d",
-     name, active_objs, num_objs, cachep->buffer_size,
+     name, c.active_objs, c.num_objs, cachep->buffer_size,
      cachep->num, (1 << cachep->gfporder));
  seq_printf(m, " : tunables %4u %4u %4u",
      cachep->limit, cachep->batchcount, cachep->shared);
  seq_printf(m, " : slabdata %6lu %6lu %6lu",
-     active_slabs, num_slabs, shared_avail);
+     c.active_slabs, c.num_slabs, c.shared_avail);
 #if STATS
  {   /* list3 stats */
   unsigned long high = cachep->high_mark;
@@ -4615,6 +4633,30 @@ static int s_show(struct seq_file *m, void *p)
  return 0;
 }

+#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
+int mem_cgroup_slabinfo(struct mem_cgroup *memcg, struct seq_file *m)
+{
+ struct kmem_cache *cachep;
+ struct slab_counts c;
+
+ seq_printf(m, "# name            <active_objs> <num_objs> <objsize>\n");
+
+ mutex_lock(&cache_chain_mutex);
+ list_for_each_entry(cachep, &cache_chain, next) {
+  if (cachep->memcg_params.memcg != memcg)
+   continue;
+
+  get_slab_counts(cachep, &c);
+
+  seq_printf(m, "%-17s %6lu %6lu %6u\n", cachep->name,
+      c.active_objs, c.num_objs, cachep->buffer_size);
+ }
+ mutex_unlock(&cache_chain_mutex);
+
+ return 0;
+}
+#endif /* CONFIG_CGROUP_MEM_RES_CTLR_KMEM */
+
 /*
  * slabinfo_op - iterator that generates /proc/slabinfo
  *
diff --git a/mm/slub.c b/mm/slub.c
index 9b22139..1031d4d 100644
--- a/mm/slub.c
```

```
+++ b/mm/slub.c
@@ -4147,6 +4147,11 @@ void kmem_cache_drop_ref(struct kmem_cache *s)
  BUG_ON(s->memcg_params.id != -1);
  kmem_cache_destroy(s);
 }
+
+int mem_cgroup_slabinfo(struct mem_cgroup *memcg, struct seq_file *m)
+{
+ return 0;
+}
 #endif

 #ifdef CONFIG_SMP
--
1.7.7.6
```

Subject: [PATCH 23/23] slub: create slabinfo file for memcg
Posted by Glauber Costa on Sun, 22 Apr 2012 23:53:40 GMT
View Forum Message <> Reply to Message

This patch implements mem_cgroup_slabinfo() for the slub.
With that, we can also probe the used caches for it.

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>
CC: Suleiman Souhlal <suleiman@google.com>
---
 mm/slub.c | 27 +++++++++++++++++++++++++++
 1 files changed, 27 insertions(+), 0 deletions(-)

```
diff --git a/mm/slub.c b/mm/slub.c
index 1031d4d..495a4f1 100644
--- a/mm/slub.c
+++ b/mm/slub.c
@@ -4150,6 +4150,33 @@ void kmem_cache_drop_ref(struct kmem_cache *s)

 int mem_cgroup_slabinfo(struct mem_cgroup *memcg, struct seq_file *m)
 {
+ struct kmem_cache *s;
+ int node;
+ unsigned long nr_objs = 0;
+ unsigned long nr_free = 0;
+
```

```
+ seq_printf(m, "# name            <active_objs> <num_objs> <objsize>\n");
+
+ down_read(&slub_lock);
+ list_for_each_entry(s, &slab_caches, list) {
+  if (s->memcg_params.memcg != memcg)
+   continue;
+
+  for_each_online_node(node) {
+   struct kmem_cache_node *n = get_node(s, node);
+
+   if (!n)
+    continue;
+
+   nr_objs += atomic_long_read(&n->total_objects);
+   nr_free += count_partial(n, count_free);
+  }
+
+  seq_printf(m, "%-17s %6lu %6lu %6u\n", s->name,
+      nr_objs - nr_free, nr_objs, s->size);
+ }
+ up_read(&slub_lock);
+
  return 0;
 }
 #endif
--
1.7.7.6
```

Subject: Re: [PATCH 00/23] slab+slub accounting for memcg
Posted by Glauber Costa on Sun, 22 Apr 2012 23:59:15 GMT
View Forum Message <> Reply to Message

On 04/20/2012 06:57 PM, Glauber Costa wrote:
> Hi,
>
> This is my current attempt at getting the kmem controller
> into a mergeable state. IMHO, all the important bits are there, and it should't
> change *that* much from now on. I am, however, expecting at least a couple more
> interactions before we sort all the edges out.
>
> This series works for both the slub and the slab. One of my main goals was to
> make sure that the interfaces we are creating actually makes sense for both
> allocators.
>
> I did some adaptations to the slab-specific patches, but the bulk of it
> comes from Suleiman's patches. I did the best to use his patches
> as-is where possible so to keep authorship information. When not possible,

> I tried to be fair and quote it in the commit message.
>
> In this series, all existing caches are created per-memcg after its first hit.
> The main reason is, during discussions in the memory summit we came into
> agreement that the fragmentation problems that could arise from creating all
> of them are mitigated by the typically small quantity of caches in the system
> (order of a few megabytes total for sparsely used caches).
> The lazy creation from Suleiman is kept, although a bit modified. For instance,
> I now use a locked scheme instead of cmpxcgh to make sure cache creation won't
> fail due to duplicates, which simplifies things by quite a bit.
>
> The slub is a bit more complex than what I came up with in my slub-only
> series. The reason is we did not need to use the cache-selection logic
> in the allocator itself - it was done by the cache users. But since now
> we are lazy creating all caches, this is simply no longer doable.
>
> I am leaving destruction of caches out of the series, although most
> of the infrastructure for that is here, since we did it in earlier
> series. This is basically because right now Kame is reworking it for
> user memcg, and I like the new proposed behavior a lot more. We all seemed
> to have agreed that reclaim is an interesting problem by itself, and
> is not included in this already too complicated series. Please note
> that this is still marked as experimental, so we have so room. A proper
> shrinker implementation is a hard requirement to take the kmem controller
> out of the experimental state.
>
> I am also not including documentation, but it should only be a matter
> of merging what we already wrote in earlier series plus some additions.
>
> Glauber Costa (19):
> slub: don't create a copy of the name string in kmem_cache_create
> slub: always get the cache from its page in kfree
> slab: rename gfpflags to allocflags
> slab: use obj_size field of struct kmem_cache when not debugging
> change defines to an enum
> don't force return value checking in res_counter_charge_nofail
> kmem slab accounting basic infrastructure
> slab/slub: struct memcg_params
> slub: consider a memcg parameter in kmem_create_cache
> slab: pass memcg parameter to kmem_cache_create
> slub: create duplicate cache
> slub: provide kmalloc_no_account
> slab: create duplicate cache
> slab: provide kmalloc_no_account
> kmem controller charge/uncharge infrastructure
> slub: charge allocation to a memcg
> slab: per-memcg accounting of slab caches
> memcg: disable kmem code when not in use.

```
>    slub: create slabinfo file for memcg
>
> Suleiman Souhlal (4):
>    memcg: Make it possible to use the stock for more than one page.
>    memcg: Reclaim when more than one page needed.
>    memcg: Track all the memcg children of a kmem_cache.
>    memcg: Per-memcg memory.kmem.slabinfo file.
>
>   include/linux/memcontrol.h  |   87 ++++++
>   include/linux/res_counter.h |    2 +-
>   include/linux/slab.h        |   26 ++
>   include/linux/slab_def.h    |   77 ++++++-
>   include/linux/slub_def.h    |   36 +++-
>   init/Kconfig                |    2 +-
>   mm/memcontrol.c             |  607 ++++++++++++++++++++++++++++++++++++++++++--
>   mm/slab.c                   |  390 +++++++++++++++++++++++-----
>   mm/slub.c                   |  255 +++++++++++++++++--
>   9 files changed, 1364 insertions(+), 118 deletions(-)
>
```

All patches should be there now.

Sorry for the trouble.

Subject: Re: [PATCH 17/23] kmem controller charge/uncharge infrastructure
Posted by David Rientjes on Mon, 23 Apr 2012 22:25:59 GMT
View Forum Message <> Reply to Message

On Sun, 22 Apr 2012, Glauber Costa wrote:

```
> +/*
> + * Return the kmem_cache we're supposed to use for a slab allocation.
> + * If we are in interrupt context or otherwise have an allocation that
> + * can't fail, we return the original cache.
> + * Otherwise, we will try to use the current memcg's version of the cache.
> + *
> + * If the cache does not exist yet, if we are the first user of it,
> + * we either create it immediately, if possible, or create it asynchronously
> + * in a workqueue.
> + * In the latter case, we will let the current allocation go through with
> + * the original cache.
> + *
> + * This function returns with rcu_read_lock() held.
> + */
> +struct kmem_cache *__mem_cgroup_get_kmem_cache(struct kmem_cache *cachep,
> +         gfp_t gfp)
> +{
> + struct mem_cgroup *memcg;
```

> + int idx;
> +
> + gfp |= cachep->allocflags;
> +
> + if ((current->mm == NULL))
> + return cachep;
> +
> + if (cachep->memcg_params.memcg)
> + return cachep;
> +
> + idx = cachep->memcg_params.id;
> + VM_BUG_ON(idx == -1);
> +
> + memcg = mem_cgroup_from_task(current);
> + if (!mem_cgroup_kmem_enabled(memcg))
> + return cachep;
> +
> + if (rcu_access_pointer(memcg->slabs[idx]) == NULL) {
> + memcg_create_cache_enqueue(memcg, cachep);
> + return cachep;
> + }
> +
> + return rcu_dereference(memcg->slabs[idx]);
> +}
> +EXPORT_SYMBOL(__mem_cgroup_get_kmem_cache);
> +
> +void mem_cgroup_remove_child_kmem_cache(struct kmem_cache *cachep, int id)
> +{
> + rcu_assign_pointer(cachep->memcg_params.memcg->slabs[id], NULL);
> +}
> +
> +bool __mem_cgroup_charge_kmem(gfp_t gfp, size_t size)
> +{
> + struct mem_cgroup *memcg;
> + bool ret = true;
> +
> + rcu_read_lock();
> + memcg = mem_cgroup_from_task(current);

This seems horribly inconsistent with memcg charging of user memory since
it charges to p->mm->owner and you're charging to p. So a thread attached
to a memcg can charge user memory to one memcg while charging slab to
another memcg?

> +
> + if (!mem_cgroup_kmem_enabled(memcg))
> + goto out;
> +

> + mem_cgroup_get(memcg);
> + ret = memcg_charge_kmem(memcg, gfp, size) == 0;
> + if (ret)
> + mem_cgroup_put(memcg);
> +out:
> + rcu_read_unlock();
> + return ret;
> +}
> +EXPORT_SYMBOL(__mem_cgroup_charge_kmem);
> +
> +void __mem_cgroup_uncharge_kmem(size_t size)
> +{
> + struct mem_cgroup *memcg;
> +
> + rcu_read_lock();
> + memcg = mem_cgroup_from_task(current);
> +
> + if (!mem_cgroup_kmem_enabled(memcg))
> + goto out;
> +
> + mem_cgroup_put(memcg);
> + memcg_uncharge_kmem(memcg, size);
> +out:
> + rcu_read_unlock();
> +}
> +EXPORT_SYMBOL(__mem_cgroup_uncharge_kmem);

Subject: Re: [PATCH 11/23] slub: consider a memcg parameter in kmem_create_cache
Posted by Frederic Weisbecker on Tue, 24 Apr 2012 14:03:31 GMT
View Forum Message <> Reply to Message

On Fri, Apr 20, 2012 at 06:57:19PM -0300, Glauber Costa wrote:
> diff --git a/mm/slub.c b/mm/slub.c
> index 2652e7c..86e40cc 100644
> --- a/mm/slub.c
> +++ b/mm/slub.c
> @@ -32,6 +32,7 @@
> #include <linux/prefetch.h>
>
> #include <trace/events/kmem.h>
> +#include <linux/memcontrol.h>
>
> /*
>   * Lock order:
> @@ -3880,7 +3881,7 @@ static int slab_unmergeable(struct kmem_cache *s)
>   return 0;

> }
>
> -static struct kmem_cache *find_mergeable(size_t size,
> +static struct kmem_cache *find_mergeable(struct mem_cgroup *memcg, size_t size,
> size_t align, unsigned long flags, const char *name,
> void (*ctor)(void *))
> {
> @@ -3916,21 +3917,29 @@ static struct kmem_cache *find_mergeable(size_t size,
> if (s->size - size >= sizeof(void *))
> continue;
>
> + if (memcg && s->memcg_params.memcg != memcg)
> + continue;
> +

This probably won't build without CONFIG_CGROUP_MEM_RES_CTLR_KMEM ?

> return s;
> }
> return NULL;
> }
>
> -struct kmem_cache *kmem_cache_create(const char *name, size_t size,
> - size_t align, unsigned long flags, void (*ctor)(void *))
> +struct kmem_cache *
> +kmem_cache_create_memcg(struct mem_cgroup *memcg, const char *name, size_t size,

Does that build without CONFIG_CGROUP_MEM_RES_CTLR ?

> + size_t align, unsigned long flags, void (*ctor)(void *))
> {
> struct kmem_cache *s;
>
> if (WARN_ON(!name))
> return NULL;
>
> +#ifndef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
> + WARN_ON(memcg != NULL);
> +#endif
> +
> down_write(&slub_lock);
> - s = find_mergeable(size, align, flags, name, ctor);
> + s = find_mergeable(memcg, size, align, flags, name, ctor);
> if (s) {
> s->refcount++;
> /*
> @@ -3954,12 +3963,15 @@ struct kmem_cache *kmem_cache_create(const char *name,
size_t size,

>   size, align, flags, ctor)) {
>   list_add(&s->list, &slab_caches);
>   up_write(&slub_lock);
> +   mem_cgroup_register_cache(memcg, s);

How do you handle when the memcg cgroup gets destroyed? Also that means only one memcg cgroup can be accounted for a given slab cache? What if that memcg cgroup has children? Hmm, perhaps this is handled in a further patch in the series, I saw a patch title with "children" inside :)

Also my knowledge on memory allocators is near zero, so I may well be asking weird questions...

Subject: Re: [PATCH 13/23] slub: create duplicate cache
Posted by Frederic Weisbecker on Tue, 24 Apr 2012 14:18:48 GMT
View Forum Message <> Reply to Message

On Sun, Apr 22, 2012 at 08:53:30PM -0300, Glauber Costa wrote:
> This patch provides kmem_cache_dup(), that duplicates
> a cache for a memcg, preserving its creation properties.
> Object size, alignment and flags are all respected.
>
> When a duplicate cache is created, the parent cache cannot
> be destructed during the child lifetime. To assure this,
> its reference count is increased if the cache creation
> succeeds.
>
> 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>
> CC: Suleiman Souhlal <suleiman@google.com>
> ---
> include/linux/memcontrol.h | 3 +++
> include/linux/slab.h | 3 +++
> mm/memcontrol.c | 44 ++
> mm/slub.c | 37 +++++++++++++++++++++++++++++++++++++
> 4 files changed, 87 insertions(+), 0 deletions(-)
>
> diff --git a/include/linux/memcontrol.h b/include/linux/memcontrol.h
> index 99e14b9..493ecdd 100644
> --- a/include/linux/memcontrol.h
> +++ b/include/linux/memcontrol.h
> @@ -445,6 +445,9 @@ int memcg_css_id(struct mem_cgroup *memcg);
> void mem_cgroup_register_cache(struct mem_cgroup *memcg,

```
>            struct kmem_cache *s);
>  void mem_cgroup_release_cache(struct kmem_cache *cachep);
> +extern char *mem_cgroup_cache_name(struct mem_cgroup *memcg,
> +       struct kmem_cache *cachep);
> +
>  #else
>  static inline void mem_cgroup_register_cache(struct mem_cgroup *memcg,
>            struct kmem_cache *s)
> diff --git a/include/linux/slab.h b/include/linux/slab.h
> index c7a7e05..909b508 100644
> --- a/include/linux/slab.h
> +++ b/include/linux/slab.h
> @@ -323,6 +323,9 @@ extern void *__kmalloc_track_caller(size_t, gfp_t, unsigned long);
>
>  #ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
>  #define MAX_KMEM_CACHE_TYPES 400
> +extern struct kmem_cache *kmem_cache_dup(struct mem_cgroup *memcg,
> +      struct kmem_cache *cachep);
> +void kmem_cache_drop_ref(struct kmem_cache *cachep);
>  #else
>  #define MAX_KMEM_CACHE_TYPES 0
>  #endif /* CONFIG_CGROUP_MEM_RES_CTLR_KMEM */
> diff --git a/mm/memcontrol.c b/mm/memcontrol.c
> index 0015ed0..e881d83 100644
> --- a/mm/memcontrol.c
> +++ b/mm/memcontrol.c
> @@ -467,6 +467,50 @@ struct cg_proto *tcp_proto_cgroup(struct mem_cgroup *memcg)
>  EXPORT_SYMBOL(tcp_proto_cgroup);
>  #endif /* CONFIG_INET */
>
> +/*
> + * This is to prevent races againt the kmalloc cache creations.
> + * Should never be used outside the core memcg code. Therefore,
> + * copy it here, instead of letting it in lib/
> + */
> +static char *kasprintf_no_account(gfp_t gfp, const char *fmt, ...)
> +{
> + unsigned int len;
> + char *p = NULL;
> + va_list ap, aq;
> +
> + va_start(ap, fmt);
> + va_copy(aq, ap);
> + len = vsnprintf(NULL, 0, fmt, aq);
> + va_end(aq);
> +
> + p = kmalloc_no_account(len+1, gfp);
```

I can't seem to find kmalloc_no_account() in this patch or may be
I missed it in a previous one?

> + if (!p)
> + goto out;
> +
> + vsnprintf(p, len+1, fmt, ap);
> +
> +out:
> + va_end(ap);
> + return p;
> +}
> +
> +char *mem_cgroup_cache_name(struct mem_cgroup *memcg, struct kmem_cache *cachep)
> +{
> + char *name;
> + struct dentry *dentry = memcg->css.cgroup->dentry;
> +
> + BUG_ON(dentry == NULL);
> +
> + /* Preallocate the space for "dead" at the end */
> + name = kasprintf_no_account(GFP_KERNEL, "%s(%d:%s)dead",
> +   cachep->name, css_id(&memcg->css), dentry->d_name.name);
> +
> + if (name)
> + /* Remove "dead" */
> + name[strlen(name) - 4] = '\0';

Why this space for "dead" ? I can't seem to find a reference to that in
the kernel. Is it something I'm missing because of my lack of slab knowledge
or is it something needed in a further patch? In which case this should be
explained in the changelog.

> + return name;
> +}
> +
> /* Bitmap used for allocating the cache id numbers. */
> static DECLARE_BITMAP(cache_types, MAX_KMEM_CACHE_TYPES);
>
> diff --git a/mm/slub.c b/mm/slub.c
> index 86e40cc..2285a96 100644
> --- a/mm/slub.c
> +++ b/mm/slub.c
> @@ -3993,6 +3993,43 @@ struct kmem_cache *kmem_cache_create(const char *name,
size_t size,
> }
> EXPORT_SYMBOL(kmem_cache_create);
>

```
> +#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
> +struct kmem_cache *kmem_cache_dup(struct mem_cgroup *memcg,
> +      struct kmem_cache *s)
> +{
> + char *name;
> + struct kmem_cache *new;
> +
> + name = mem_cgroup_cache_name(memcg, s);
> + if (!name)
> +  return NULL;
> +
> + new = kmem_cache_create_memcg(memcg, name, s->objsize, s->align,
> +          s->allocflags, s->ctor);
> +
> + /*
> +  * We increase the reference counter in the parent cache, to
> +  * prevent it from being deleted. If kmem_cache_destroy() is
> +  * called for the root cache before we call it for a child cache,
> +  * it will be queued for destruction when we finally drop the
> +  * reference on the child cache.
> +  */
> + if (new) {
> +  down_write(&slub_lock);
> +  s->refcount++;
> +  up_write(&slub_lock);
> + }
> +
> + return new;
> +}
> +
> +void kmem_cache_drop_ref(struct kmem_cache *s)
> +{
> + BUG_ON(s->memcg_params.id != -1);
> + kmem_cache_destroy(s);
> +}
> +#endif
> +
>  #ifdef CONFIG_SMP
>  /*
>   * Use the cpu notifier to insure that the cpu slabs are flushed when
> --
> 1.7.7.6
>
> --
> To unsubscribe from this list: send the line "unsubscribe linux-kernel" in
> the body of a message to majordomo@vger.kernel.org
> More majordomo info at  http://vger.kernel.org/majordomo-info.html
> Please read the FAQ at  http://www.tux.org/lkml/
```

Subject: Re: [PATCH 17/23] kmem controller charge/uncharge infrastructure
Posted by Frederic Weisbecker on Tue, 24 Apr 2012 14:22:34 GMT
View Forum Message <> Reply to Message

On Mon, Apr 23, 2012 at 03:25:59PM -0700, David Rientjes wrote:
> On Sun, 22 Apr 2012, Glauber Costa wrote:
>
> > +/*
> > + * Return the kmem_cache we're supposed to use for a slab allocation.
> > + * If we are in interrupt context or otherwise have an allocation that
> > + * can't fail, we return the original cache.
> > + * Otherwise, we will try to use the current memcg's version of the cache.
> > + *
> > + * If the cache does not exist yet, if we are the first user of it,
> > + * we either create it immediately, if possible, or create it asynchronously
> > + * in a workqueue.
> > + * In the latter case, we will let the current allocation go through with
> > + * the original cache.
> > + *
> > + * This function returns with rcu_read_lock() held.
> > + */
> > +struct kmem_cache *__mem_cgroup_get_kmem_cache(struct kmem_cache *cachep,
> > +   gfp_t gfp)
> > +{
> > + struct mem_cgroup *memcg;
> > + int idx;
> > +
> > + gfp |= cachep->allocflags;
> > +
> > + if ((current->mm == NULL))
> > + return cachep;
> > +
> > + if (cachep->memcg_params.memcg)
> > + return cachep;
> > +
> > + idx = cachep->memcg_params.id;
> > + VM_BUG_ON(idx == -1);
> > +
> > + memcg = mem_cgroup_from_task(current);
> > + if (!mem_cgroup_kmem_enabled(memcg))
> > + return cachep;
> > +
> > + if (rcu_access_pointer(memcg->slabs[idx]) == NULL) {
> > + memcg_create_cache_enqueue(memcg, cachep);
> > + return cachep;
> > + }
> > +
> > + return rcu_dereference(memcg->slabs[idx]);
> > +}

> > +EXPORT_SYMBOL(__mem_cgroup_get_kmem_cache);
> > +
> > +void mem_cgroup_remove_child_kmem_cache(struct kmem_cache *cachep, int id)
> > +{
> > + rcu_assign_pointer(cachep->memcg_params.memcg->slabs[id], NULL);
> > +}
> > +
> > +bool __mem_cgroup_charge_kmem(gfp_t gfp, size_t size)
> > +{
> > + struct mem_cgroup *memcg;
> > + bool ret = true;
> > +
> > + rcu_read_lock();
> > + memcg = mem_cgroup_from_task(current);
>
> This seems horribly inconsistent with memcg charging of user memory since
> it charges to p->mm->owner and you're charging to p. So a thread attached
> to a memcg can charge user memory to one memcg while charging slab to
> another memcg?

Charging to the thread rather than the process seem to me the right behaviour:
you can have two threads of a same process attached to different cgroups.

Perhaps it is the user memory memcg that needs to be fixed?

>
> > +
> > + if (!mem_cgroup_kmem_enabled(memcg))
> > + goto out;
> > +
> > + mem_cgroup_get(memcg);
> > + ret = memcg_charge_kmem(memcg, gfp, size) == 0;
> > + if (ret)
> > + mem_cgroup_put(memcg);
> > +out:
> > + rcu_read_unlock();
> > + return ret;
> > +}
> > +EXPORT_SYMBOL(__mem_cgroup_charge_kmem);
> > +
> > +void __mem_cgroup_uncharge_kmem(size_t size)
> > +{
> > + struct mem_cgroup *memcg;
> > +
> > + rcu_read_lock();
> > + memcg = mem_cgroup_from_task(current);
> > +
> > + if (!mem_cgroup_kmem_enabled(memcg))

> > + goto out;
> > +
> > + mem_cgroup_put(memcg);
> > + memcg_uncharge_kmem(memcg, size);
> > +out:
> > + rcu_read_unlock();
> > +}
> > +EXPORT_SYMBOL(__mem_cgroup_uncharge_kmem);

Subject: Re: [PATCH 11/23] slub: consider a memcg parameter in kmem_create_cache
Posted by Glauber Costa on Tue, 24 Apr 2012 14:27:50 GMT
View Forum Message <> Reply to Message

On 04/24/2012 11:03 AM, Frederic Weisbecker wrote:
> On Fri, Apr 20, 2012 at 06:57:19PM -0300, Glauber Costa wrote:
>> diff --git a/mm/slub.c b/mm/slub.c
>> index 2652e7c..86e40cc 100644
>> --- a/mm/slub.c
>> +++ b/mm/slub.c
>> @@ -32,6 +32,7 @@
>> #include<linux/prefetch.h>
>>
>> #include<trace/events/kmem.h>
>> +#include<linux/memcontrol.h>
>>
>> /*
>>   * Lock order:
>> @@ -3880,7 +3881,7 @@ static int slab_unmergeable(struct kmem_cache *s)
>>   return 0;
>> }
>>
>> -static struct kmem_cache *find_mergeable(size_t size,
>> +static struct kmem_cache *find_mergeable(struct mem_cgroup *memcg, size_t size,
>>   size_t align, unsigned long flags, const char *name,
>>   void (*ctor)(void *))
>> {
>> @@ -3916,21 +3917,29 @@ static struct kmem_cache *find_mergeable(size_t size,
>>   if (s->size - size>= sizeof(void *))
>>   continue;
>>
>> + if (memcg&& s->memcg_params.memcg != memcg)
>> +   continue;
>> +
>
> This probably won't build without CONFIG_CGROUP_MEM_RES_CTLR_KMEM ?

Probably not, thanks.

>
>>   return s;
>>   }
>>   return NULL;
>>   }
>>
>> -struct kmem_cache *kmem_cache_create(const char *name, size_t size,
>> - size_t align, unsigned long flags, void (*ctor)(void *))
>> +struct kmem_cache *
>> +kmem_cache_create_memcg(struct mem_cgroup *memcg, const char *name, size_t size,
>
> Does that build without CONFIG_CGROUP_MEM_RES_CTLR ?
Yes, because MEM_RES_CTLR_KMEM is dependent on RES_CTLR.

>
>> +   size_t align, unsigned long flags, void (*ctor)(void *))
>>   {
>>   struct kmem_cache *s;
>>
>>   if (WARN_ON(!name))
>>   return NULL;
>>
>> +#ifndef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
>> + WARN_ON(memcg != NULL);
>> +#endif
>> +
>>   down_write(&slub_lock);
>> - s = find_mergeable(size, align, flags, name, ctor);
>> + s = find_mergeable(memcg, size, align, flags, name, ctor);
>>   if (s) {
>>   s->refcount++;
>>   /*
>> @@ -3954,12 +3963,15 @@ struct kmem_cache *kmem_cache_create(const char *name,
size_t size,
>>   size, align, flags, ctor)) {
>>   list_add(&s->list,&slab_caches);
>>   up_write(&slub_lock);
>> +   mem_cgroup_register_cache(memcg, s);
>
> How do you handle when the memcg cgroup gets destroyed?

I don't (yet), because - as mentioned in patch 0 - I decided to hold
those patches until I had a better idea about how would Kame's
pre_destroy() patches look like. I plan, however, to include it in the
next version.

The idea is basically to mark the caches as dead (answers another
question of yours), and wait until it runs out of objects. Talking
specifically about the slub, that happens when free_page() frees the
last page of the cache *and* its reference count goes down to zero
(kmem_cache_destroy() drops the refcnt, so it will mean that cgroup
destruction already called it)

When we have a shrinker - I don't plan to include a per-memcg shrinker
in the first merge, because let's face it, it is a hard problem in
itself that would be better thought separately - we can call the
shrinkers to force the objects to die earlier.

> Also that means only one
> memcg cgroup can be accounted for a given slab cache?

Not sure if I understand your question in an ambiguity-free way.
If you mean the situation in which two tasks touch the same object, then
yes, only one of them is accounted.

If you mean about types of cache, then no, each memcg can have it's own
version of the whole cache array.

> What if that memcg cgroup has
> children? Hmm, perhaps this is handled in a further patch in the series, I saw a
> patch title with "children" inside :)

then the children creates caches as well, as much as the parents.

Note that because of the delayed allocation mechanism, if the parent
serves only as a placeholder, and has no tasks inside it, then it will
never touch - and therefore never create - any cache.

Subject: Re: [PATCH 13/23] slub: create duplicate cache
Posted by Glauber Costa on Tue, 24 Apr 2012 14:37:59 GMT
View Forum Message <> Reply to Message

On 04/24/2012 11:18 AM, Frederic Weisbecker wrote:
> On Sun, Apr 22, 2012 at 08:53:30PM -0300, Glauber Costa wrote:
>> This patch provides kmem_cache_dup(), that duplicates
>> a cache for a memcg, preserving its creation properties.
>> Object size, alignment and flags are all respected.
>>
>> When a duplicate cache is created, the parent cache cannot
>> be destructed during the child lifetime. To assure this,
>> its reference count is increased if the cache creation
>> succeeds.

>>
>> 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>
>> CC: Suleiman Souhlal<suleiman@google.com>
>> ---
>> include/linux/memcontrol.h | 3 +++
>> include/linux/slab.h | 3 +++
>> mm/memcontrol.c | 44 ++
>> mm/slub.c | 37 +++++++++++++++++++++++++++++++++++++
>> 4 files changed, 87 insertions(+), 0 deletions(-)
>>
>> diff --git a/include/linux/memcontrol.h b/include/linux/memcontrol.h
>> index 99e14b9..493ecdd 100644
>> --- a/include/linux/memcontrol.h
>> +++ b/include/linux/memcontrol.h
>> @@ -445,6 +445,9 @@ int memcg_css_id(struct mem_cgroup *memcg);
>> void mem_cgroup_register_cache(struct mem_cgroup *memcg,
>>   struct kmem_cache *s);
>> void mem_cgroup_release_cache(struct kmem_cache *cachep);
>> +extern char *mem_cgroup_cache_name(struct mem_cgroup *memcg,
>> +   struct kmem_cache *cachep);
>> +
>> #else
>> static inline void mem_cgroup_register_cache(struct mem_cgroup *memcg,
>>   struct kmem_cache *s)
>> diff --git a/include/linux/slab.h b/include/linux/slab.h
>> index c7a7e05..909b508 100644
>> --- a/include/linux/slab.h
>> +++ b/include/linux/slab.h
>> @@ -323,6 +323,9 @@ extern void *__kmalloc_track_caller(size_t, gfp_t, unsigned long);
>>
>> #ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
>> #define MAX_KMEM_CACHE_TYPES 400
>> +extern struct kmem_cache *kmem_cache_dup(struct mem_cgroup *memcg,
>> +   struct kmem_cache *cachep);
>> +void kmem_cache_drop_ref(struct kmem_cache *cachep);
>> #else
>> #define MAX_KMEM_CACHE_TYPES 0
>> #endif /* CONFIG_CGROUP_MEM_RES_CTLR_KMEM */
>> diff --git a/mm/memcontrol.c b/mm/memcontrol.c
>> index 0015ed0..e881d83 100644
>> --- a/mm/memcontrol.c
>> +++ b/mm/memcontrol.c
>> @@ -467,6 +467,50 @@ struct cg_proto *tcp_proto_cgroup(struct mem_cgroup *memcg)

>> EXPORT_SYMBOL(tcp_proto_cgroup);
>> #endif /* CONFIG_INET */
>>
>> +/*
>> + * This is to prevent races againt the kmalloc cache creations.
>> + * Should never be used outside the core memcg code. Therefore,
>> + * copy it here, instead of letting it in lib/
>> + */
>> +static char *kasprintf_no_account(gfp_t gfp, const char *fmt, ...)
>> +{
>> + unsigned int len;
>> + char *p = NULL;
>> + va_list ap, aq;
>> +
>> + va_start(ap, fmt);
>> + va_copy(aq, ap);
>> + len = vsnprintf(NULL, 0, fmt, aq);
>> + va_end(aq);
>> +
>> + p = kmalloc_no_account(len+1, gfp);
>
> I can't seem to find kmalloc_no_account() in this patch or may be
> I missed it in a previous one?

It is in a previous one (actually two, one for the slab, one for the
slub). They are bundled in the cache creation, but I could separate it
for clarity, if you prefer.

>> + if (!p)
>> + goto out;
>> +
>> + vsnprintf(p, len+1, fmt, ap);
>> +
>> +out:
>> + va_end(ap);
>> + return p;
>> +}
>> +
>> +char *mem_cgroup_cache_name(struct mem_cgroup *memcg, struct kmem_cache *cachep)
>> +{
>> + char *name;
>> + struct dentry *dentry = memcg->css.cgroup->dentry;
>> +
>> + BUG_ON(dentry == NULL);
>> +
>> + /* Preallocate the space for "dead" at the end */
>> + name = kasprintf_no_account(GFP_KERNEL, "%s(%d:%s)dead",

>> +   cachep->name, css_id(&memcg->css), dentry->d_name.name);
>> +
>> + if (name)
>> + /* Remove "dead" */
>> + name[strlen(name) - 4] = '\0';
>
> Why this space for "dead" ?

Ok, sorry. Since I didn't include the destruction part, it got too easy
for whoever wasn't following the last discussion on this to get lost -
My bad. So here it is:

When we destroy the memcg, some objects may still hold the cache in
memory. It is like a reference count, in a sense, which each object
being a reference.

In typical cases, like non-shrinkable caches that has create - destroy
patterns, the caches will go away as soon as the tasks using them.

But in cache-like structure like the dentry cache, the objects may hang
around until a shrinker pass takes them out. And even then, some of them
will live on.

In this case, we will display them with "dead" in the name.

We could hide them, but then it gets weirder because it would be hard to
understand where is your used memory when you need to inspect your system.

Creating another file, slabinfo_deadcaches, and keeping the names, is
also a possibility, if people think that the string append is way too ugly.

Subject: Re: [PATCH 17/23] kmem controller charge/uncharge infrastructure
Posted by Glauber Costa on Tue, 24 Apr 2012 14:40:34 GMT
View Forum Message <> Reply to Message

On 04/24/2012 11:22 AM, Frederic Weisbecker wrote:
> On Mon, Apr 23, 2012 at 03:25:59PM -0700, David Rientjes wrote:
>> On Sun, 22 Apr 2012, Glauber Costa wrote:
>>
>>> +/*
>>> + * Return the kmem_cache we're supposed to use for a slab allocation.
>>> + * If we are in interrupt context or otherwise have an allocation that
>>> + * can't fail, we return the original cache.
>>> + * Otherwise, we will try to use the current memcg's version of the cache.
>>> + *
>>> + * If the cache does not exist yet, if we are the first user of it,
>>> + * we either create it immediately, if possible, or create it asynchronously

```
>>> + * in a workqueue.
>>> + * In the latter case, we will let the current allocation go through with
>>> + * the original cache.
>>> + *
>>> + * This function returns with rcu_read_lock() held.
>>> + */
>>> +struct kmem_cache *__mem_cgroup_get_kmem_cache(struct kmem_cache *cachep,
>>> +          gfp_t gfp)
>>> +{
>>> + struct mem_cgroup *memcg;
>>> + int idx;
>>> +
>>> + gfp |=  cachep->allocflags;
>>> +
>>> + if ((current->mm == NULL))
>>> +  return cachep;
>>> +
>>> + if (cachep->memcg_params.memcg)
>>> +  return cachep;
>>> +
>>> + idx = cachep->memcg_params.id;
>>> + VM_BUG_ON(idx == -1);
>>> +
>>> + memcg = mem_cgroup_from_task(current);
>>> + if (!mem_cgroup_kmem_enabled(memcg))
>>> +  return cachep;
>>> +
>>> + if (rcu_access_pointer(memcg->slabs[idx]) == NULL) {
>>> +  memcg_create_cache_enqueue(memcg, cachep);
>>> +  return cachep;
>>> + }
>>> +
>>> + return rcu_dereference(memcg->slabs[idx]);
>>> +}
>>> +EXPORT_SYMBOL(__mem_cgroup_get_kmem_cache);
>>> +
>>> +void mem_cgroup_remove_child_kmem_cache(struct kmem_cache *cachep, int id)
>>> +{
>>> + rcu_assign_pointer(cachep->memcg_params.memcg->slabs[id], NULL);
>>> +}
>>> +
>>> +bool __mem_cgroup_charge_kmem(gfp_t gfp, size_t size)
>>> +{
>>> + struct mem_cgroup *memcg;
>>> + bool ret = true;
>>> +
>>> + rcu_read_lock();
>>> + memcg = mem_cgroup_from_task(current);
```

>>
>> This seems horribly inconsistent with memcg charging of user memory since
>> it charges to p->mm->owner and you're charging to p. So a thread attached
>> to a memcg can charge user memory to one memcg while charging slab to
>> another memcg?
>
> Charging to the thread rather than the process seem to me the right behaviour:
> you can have two threads of a same process attached to different cgroups.
>
> Perhaps it is the user memory memcg that needs to be fixed?
>

Hi David,

I just saw all the answers, so I will bundle here since Frederic also
chimed in...

I think memcg is not necessarily wrong. That is because threads in a
process share an address space, and you will eventually need to map a
page to deliver it to userspace. The mm struct points you to the owner
of that.

But that is not necessarily true for things that live in the kernel
address space.

Do you view this differently ?

Subject: Re: [PATCH 17/23] kmem controller charge/uncharge infrastructure
Posted by David Rientjes on Tue, 24 Apr 2012 20:21:43 GMT
View Forum Message <> Reply to Message

On Tue, 24 Apr 2012, Frederic Weisbecker wrote:

> > This seems horribly inconsistent with memcg charging of user memory since
> > it charges to p->mm->owner and you're charging to p. So a thread attached
> > to a memcg can charge user memory to one memcg while charging slab to
> > another memcg?
>
> Charging to the thread rather than the process seem to me the right behaviour:
> you can have two threads of a same process attached to different cgroups.
>
> Perhaps it is the user memory memcg that needs to be fixed?
>

No, because memory is represented by mm_struct, not task_struct, so you
must charge to p->mm->owner to allow for moving threads amongst memcgs
later for memory.move_charge_at_immigrate. You shouldn't be able to

charge two different memcgs for memory represented by a single mm.

```
> > > +
> > > + if (!mem_cgroup_kmem_enabled(memcg))
> > > +  goto out;
> > > +
> > > + mem_cgroup_get(memcg);
> > > + ret = memcg_charge_kmem(memcg, gfp, size) == 0;
> > > + if (ret)
> > > +  mem_cgroup_put(memcg);
> > > +out:
> > > + rcu_read_unlock();
> > > + return ret;
> > > +}
> > > +EXPORT_SYMBOL(__mem_cgroup_charge_kmem);
> > > +
> > > +void __mem_cgroup_uncharge_kmem(size_t size)
> > > +{
> > > + struct mem_cgroup *memcg;
> > > +
> > > + rcu_read_lock();
> > > + memcg = mem_cgroup_from_task(current);
> > > +
> > > + if (!mem_cgroup_kmem_enabled(memcg))
> > > +  goto out;
> > > +
> > > + mem_cgroup_put(memcg);
> > > + memcg_uncharge_kmem(memcg, size);
> > > +out:
> > > + rcu_read_unlock();
> > > +}
> > > +EXPORT_SYMBOL(__mem_cgroup_uncharge_kmem);
```

Subject: Re: [PATCH 17/23] kmem controller charge/uncharge infrastructure
Posted by David Rientjes on Tue, 24 Apr 2012 20:25:57 GMT
View Forum Message <> Reply to Message

On Tue, 24 Apr 2012, Glauber Costa wrote:

> I think memcg is not necessarily wrong. That is because threads in a process
> share an address space, and you will eventually need to map a page to deliver
> it to userspace. The mm struct points you to the owner of that.
>
> But that is not necessarily true for things that live in the kernel address
> space.
>
> Do you view this differently ?

>

Yes, for user memory, I see charging to p->mm->owner as allowing that
process to eventually move and be charged to a different memcg and there's
no way to do proper accounting if the charge is split amongst different
memcgs because of thread membership to a set of memcgs. This is
consistent with charges for shared memory being moved when a thread
mapping it moves to a new memcg, as well.

Subject: Re: [PATCH 17/23] kmem controller charge/uncharge infrastructure
Posted by Glauber Costa on Tue, 24 Apr 2012 21:36:02 GMT
View Forum Message <> Reply to Message

On 04/24/2012 05:25 PM, David Rientjes wrote:
> On Tue, 24 Apr 2012, Glauber Costa wrote:
>
>> I think memcg is not necessarily wrong. That is because threads in a process
>> share an address space, and you will eventually need to map a page to deliver
>> it to userspace. The mm struct points you to the owner of that.
>>
>> But that is not necessarily true for things that live in the kernel address
>> space.
>>
>> Do you view this differently ?
>>
>
> Yes, for user memory, I see charging to p->mm->owner as allowing that
> process to eventually move and be charged to a different memcg and there's
> no way to do proper accounting if the charge is split amongst different
> memcgs because of thread membership to a set of memcgs. This is
> consistent with charges for shared memory being moved when a thread
> mapping it moves to a new memcg, as well.

But that's the problem.

When we are dealing with kernel memory, we are allocating a whole slab
page. It is essentially impossible to track, given a page, which task
allocated which object.

Subject: Re: [PATCH 17/23] kmem controller charge/uncharge infrastructure
Posted by David Rientjes on Tue, 24 Apr 2012 22:54:20 GMT
View Forum Message <> Reply to Message

On Tue, 24 Apr 2012, Glauber Costa wrote:

> > Yes, for user memory, I see charging to p->mm->owner as allowing that
> > process to eventually move and be charged to a different memcg and there's
> > no way to do proper accounting if the charge is split amongst different
> > memcgs because of thread membership to a set of memcgs. This is
> > consistent with charges for shared memory being moved when a thread
> > mapping it moves to a new memcg, as well.
>
> But that's the problem.
>
> When we are dealing with kernel memory, we are allocating a whole slab page.
> It is essentially impossible to track, given a page, which task allocated
> which object.
>

Right, so you have to make the distinction that slab charges cannot be migrated by memory.move_charge_at_immigrate (and it's not even specified to do anything beyond user pages in Documentation/cgroups/memory.txt), but it would be consistent to charge the same memcg for a process's slab allocations as the process's user allocations.

My response was why we shouldn't be charging user pages to mem_cgroup_from_task(current) rather than mem_cgroup_from_task(current->mm->owner) which is what is currently implemented.

If that can't be changed so that we can still migrate user memory amongst memcgs for memory.move_charge_at_immigrate, then it seems consistent to have all allocations done by a task to be charged to the same memcg.
Hence, I suggested current->mm->owner for slab charging as well.

Subject: Re: [PATCH 04/23] memcg: Make it possible to use the stock for more than one page.
Posted by KAMEZAWA Hiroyuki on Wed, 25 Apr 2012 00:59:48 GMT
View Forum Message <> Reply to Message

(2012/04/21 6:57), Glauber Costa wrote:

> From: Suleiman Souhlal <ssouhlal@FreeBSD.org>
>
> Signed-off-by: Suleiman Souhlal <suleiman@google.com>

ok, should work enough.

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

Subject: Re: [PATCH 05/23] memcg: Reclaim when more than one page needed.
Posted by KAMEZAWA Hiroyuki on Wed, 25 Apr 2012 01:16:42 GMT
View Forum Message <> Reply to Message

(2012/04/21 6:57), Glauber Costa wrote:

> From: Suleiman Souhlal <ssouhlal@FreeBSD.org>
>
> mem_cgroup_do_charge() was written before slab 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 pages (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?

IIRC, it was for preventing rapid OOM kill and reducing latency.

> And should there be a retry count limit, of what? For now retry up to
> COSTLY_ORDER (as page_alloc.c does), stay safe with a cond_resched(),
> and make sure not to do it if __GFP_NORETRY.
>
> Signed-off-by: Suleiman Souhlal <suleiman@google.com>

Hmm, maybe ok.

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

> ---
> mm/memcontrol.c | 18 ++++++++++++-------
> 1 files changed, 11 insertions(+), 7 deletions(-)
>
> diff --git a/mm/memcontrol.c b/mm/memcontrol.c
> index 4b94b2d..cbffc4c 100644
> --- a/mm/memcontrol.c
> +++ b/mm/memcontrol.c
> @@ -2187,7 +2187,8 @@ 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;
> @@ -2210,18 +2211,18 @@ 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;
> @@ -2234,8 +2235,10 @@ 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 <= (PAGE_SIZE << PAGE_ALLOC_COSTLY_ORDER) && ret) {
> + cond_resched();
>   return CHARGE_RETRY;
> + }
>
>   /*
>   * At task move, charge accounts can be doubly counted. So, it's
> @@ -2369,7 +2372,8 @@ 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;

Subject: Re: [PATCH 07/23] change defines to an enum
Posted by KAMEZAWA Hiroyuki on Wed, 25 Apr 2012 01:18:42 GMT
View Forum Message <> Reply to Message

(2012/04/21 6:57), Glauber Costa wrote:

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

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

Subject: Re: [PATCH 08/23] don't force return value checking in res_counter_charge_nofail
Posted by KAMEZAWA Hiroyuki on Wed, 25 Apr 2012 01:28:31 GMT
View Forum Message <> Reply to Message

(2012/04/21 6:57), Glauber Costa wrote:

> Since we will succeed with the allocation no matter what, there
> isn't the need to use __must_check with it. It can very well
> be optional.
>
> Signed-off-by: Glauber Costa <glommer@parallels.com>
> CC: Kamezawa Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com>
> CC: Johannes Weiner <hannes@cmpxchg.org>
> CC: Michal Hocko <mhocko@suse.cz>

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

Subject: Re: [PATCH 09/23] kmem slab accounting basic infrastructure
Posted by KAMEZAWA Hiroyuki on Wed, 25 Apr 2012 01:32:32 GMT

(2012/04/21 6:57), 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.
>
> The code is not enabled until a limit is set. This can be tested by the flag
> "kmem_accounted". This means that after the patch is applied, no behavioral
> changes exists for whoever is still using memcg to control their memory usage.
>

Hmm, res_counter never goes naeative ?

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

The code itself seems fine.

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

Subject: Re: [PATCH 11/23] slub: consider a memcg parameter in kmem_create_cache
Posted by KAMEZAWA Hiroyuki on Wed, 25 Apr 2012 01:38:59 GMT

(2012/04/21 6:57), Glauber Costa wrote:

> Allow a memcg parameter to be passed during cache creation.
> The slub allocator will only merge caches that belong to
> the same memcg.
>
> Default function is created as a wrapper, passing NULL
> to the memcg version. We only merge caches that belong
> to the same memcg.
>
>>From the memcontrol.c side, 3 helper functions are created:
>
> 1) memcg_css_id: because slub needs a unique cache name
>   for sysfs. Since this is visible, but not the canonical
>   location for slab data, the cache name is not used, the
>   css_id should suffice.
>
> 2) mem_cgroup_register_cache: is responsible for assigning
>   a unique index to each cache, and other general purpose
>   setup. The index is only assigned for the root caches. All
>   others are assigned index == -1.
>
> 3) mem_cgroup_release_cache: can be called from the root cache
>   destruction, and will release the index for other caches.
>
> This index mechanism was developed by Suleiman Souhlal.
>
> 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>
> CC: Suleiman Souhlal <suleiman@google.com>
> ---
> include/linux/memcontrol.h | 14 +++++++++++++
> include/linux/slab.h | 6 ++++++
> mm/memcontrol.c | 29 +++++++++++++++++++++++++++++
> mm/slub.c | 31 +++++++++++++++++++++++++++++++----
> 4 files changed, 76 insertions(+), 4 deletions(-)
>
> diff --git a/include/linux/memcontrol.h b/include/linux/memcontrol.h
> index f94efd2..99e14b9 100644
> --- a/include/linux/memcontrol.h
> +++ b/include/linux/memcontrol.h
> @@ -26,6 +26,7 @@ struct mem_cgroup;
> struct page_cgroup;
> struct page;
> struct mm_struct;

> +struct kmem_cache;
>
> /* Stats that can be updated by kernel. */
> enum mem_cgroup_page_stat_item {
> @@ -440,7 +441,20 @@ struct sock;
> #ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
> void sock_update_memcg(struct sock *sk);
> void sock_release_memcg(struct sock *sk);
> +int memcg_css_id(struct mem_cgroup *memcg);
> +void mem_cgroup_register_cache(struct mem_cgroup *memcg,
> +   struct kmem_cache *s);
> +void mem_cgroup_release_cache(struct kmem_cache *cachep);
> #else
> +static inline void mem_cgroup_register_cache(struct mem_cgroup *memcg,
> +   struct kmem_cache *s)
> +{
> +}
> +
> +static inline void mem_cgroup_release_cache(struct kmem_cache *cachep)
> +{
> +}
> +
> static inline void sock_update_memcg(struct sock *sk)
> {
> }
> diff --git a/include/linux/slab.h b/include/linux/slab.h
> index a5127e1..c7a7e05 100644
> --- a/include/linux/slab.h
> +++ b/include/linux/slab.h
> @@ -321,6 +321,12 @@ extern void *__kmalloc_track_caller(size_t, gfp_t, unsigned long);
>   __kmalloc(size, flags)
> #endif /* DEBUG_SLAB */
>
> +#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
> +#define MAX_KMEM_CACHE_TYPES 400
> +#else
> +#define MAX_KMEM_CACHE_TYPES 0
> +#endif /* CONFIG_CGROUP_MEM_RES_CTLR_KMEM */
> +

why 400 ?

> #ifdef CONFIG_NUMA
> /*
>   * kmalloc_node_track_caller is a special version of kmalloc_node that
> diff --git a/mm/memcontrol.c b/mm/memcontrol.c

> index 36f1e6b..0015ed0 100644
> --- a/mm/memcontrol.c
> +++ b/mm/memcontrol.c
> @@ -323,6 +323,11 @@ struct mem_cgroup {
> #endif
> };
>
> +int memcg_css_id(struct mem_cgroup *memcg)
> +{
> + return css_id(&memcg->css);
> +}
> +
> /* Stuffs for move charges at task migration. */
> /*
>   * Types of charges to be moved. "move_charge_at_immitgrate" is treated as a
> @@ -461,6 +466,30 @@ struct cg_proto *tcp_proto_cgroup(struct mem_cgroup *memcg)
> }
> EXPORT_SYMBOL(tcp_proto_cgroup);
> #endif /* CONFIG_INET */
> +
> +/* Bitmap used for allocating the cache id numbers. */
> +static DECLARE_BITMAP(cache_types, MAX_KMEM_CACHE_TYPES);
> +
> +void mem_cgroup_register_cache(struct mem_cgroup *memcg,
> +   struct kmem_cache *cachep)
> +{
> + int id = -1;
> +
> + cachep->memcg_params.memcg = memcg;
> +
> + if (!memcg) {
> + id = find_first_zero_bit(cache_types, MAX_KMEM_CACHE_TYPES);
> + BUG_ON(id < 0 || id >= MAX_KMEM_CACHE_TYPES);
> + __set_bit(id, cache_types);

No lock here ? you need find_first_zero_bit_and_set_atomic() or some.
Rather than that, I think you can use lib/idr.c::ida_simple_get().

> + } else
> + INIT_LIST_HEAD(&cachep->memcg_params.destroyed_list);
> + cachep->memcg_params.id = id;
> +}
> +
> +void mem_cgroup_release_cache(struct kmem_cache *cachep)
> +{
> + __clear_bit(cachep->memcg_params.id, cache_types);
> +}

> #endif /* CONFIG_CGROUP_MEM_RES_CTLR_KMEM */
>
> static void drain_all_stock_async(struct mem_cgroup *memcg);
> diff --git a/mm/slub.c b/mm/slub.c
> index 2652e7c..86e40cc 100644
> --- a/mm/slub.c
> +++ b/mm/slub.c
> @@ -32,6 +32,7 @@
> #include <linux/prefetch.h>
>
> #include <trace/events/kmem.h>
> +#include <linux/memcontrol.h>
>
> /*
>   * Lock order:
> @@ -3880,7 +3881,7 @@ static int slab_unmergeable(struct kmem_cache *s)
>   return 0;
> }
>
> -static struct kmem_cache *find_mergeable(size_t size,
> +static struct kmem_cache *find_mergeable(struct mem_cgroup *memcg, size_t size,
>   size_t align, unsigned long flags, const char *name,
>   void (*ctor)(void *))
> {
> @@ -3916,21 +3917,29 @@ static struct kmem_cache *find_mergeable(size_t size,
>   if (s->size - size >= sizeof(void *))
>   continue;
>
> + if (memcg && s->memcg_params.memcg != memcg)
> +   continue;
> +
>   return s;
>   }
>   return NULL;
> }
>
> -struct kmem_cache *kmem_cache_create(const char *name, size_t size,
> - size_t align, unsigned long flags, void (*ctor)(void *))
> +struct kmem_cache *
> +kmem_cache_create_memcg(struct mem_cgroup *memcg, const char *name, size_t size,
> +   size_t align, unsigned long flags, void (*ctor)(void *))
> {
>   struct kmem_cache *s;
>
>   if (WARN_ON(!name))
>   return NULL;
>
> +#ifndef CONFIG_CGROUP_MEM_RES_CTLR_KMEM

> + WARN_ON(memcg != NULL);
> +#endif

I'm sorry what's is this warning for ?

> @@ -5265,6 +5283,11 @@ static char *create_unique_id(struct kmem_cache *s)
> if (p != name + 1)
> *p++ = '-';
> p += sprintf(p, "%07d", s->size);
> +
> +#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
> + if (s->memcg_params.memcg)
> + p += sprintf(p, "-%08d", memcg_css_id(s->memcg_params.memcg));
> +#endif
> BUG_ON(p > name + ID_STR_LENGTH - 1);
> return name;
> }

So, you use 'id' in user interface. Should we provide 'id' as memory.id file ?

Thanks,
-Kame

Subject: Re: [PATCH 16/23] slab: provide kmalloc_no_account
Posted by KAMEZAWA Hiroyuki on Wed, 25 Apr 2012 01:44:35 GMT
View Forum Message <> Reply to Message

(2012/04/23 8:53), Glauber Costa wrote:

> Some allocations need to be accounted to the root memcg regardless
> of their context. One trivial example, is the allocations we do
> during the memcg slab cache creation themselves. Strictly speaking,
> they could go to the parent, but it is way easier to bill them to
> the root cgroup.
>
> Only generic kmalloc allocations are allowed to be bypassed.
>
> The function is not exported, because drivers code should always
> be accounted.
>
> This code is mosly written by Suleiman Souhlal.
>
> 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>
> CC: Suleiman Souhlal <suleiman@google.com>

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

Hmm...but can't we find the 'context' in automatic way ?

-Kame

> ---
> include/linux/slab_def.h | 1 +
> mm/slab.c | 23 ++++++++++++++++++++++
> 2 files changed, 24 insertions(+), 0 deletions(-)
>
> diff --git a/include/linux/slab_def.h b/include/linux/slab_def.h
> index 06e4a3e..54d25d7 100644
> --- a/include/linux/slab_def.h
> +++ b/include/linux/slab_def.h
> @@ -114,6 +114,7 @@ extern struct cache_sizes malloc_sizes[];
>
> void *kmem_cache_alloc(struct kmem_cache *, gfp_t);
> void *__kmalloc(size_t size, gfp_t flags);
> +void *kmalloc_no_account(size_t size, gfp_t flags);
>
> #ifdef CONFIG_TRACING
> extern void *kmem_cache_alloc_trace(size_t size,
> diff --git a/mm/slab.c b/mm/slab.c
> index c4ef684..13948c3 100644
> --- a/mm/slab.c
> +++ b/mm/slab.c
> @@ -3960,6 +3960,29 @@ void *__kmalloc(size_t size, gfp_t flags)
> }
> EXPORT_SYMBOL(__kmalloc);
>
> +static __always_inline void *__do_kmalloc_no_account(size_t size, gfp_t flags,
> +   void *caller)
> +{
> + struct kmem_cache *cachep;
> + void *ret;
> +
> + cachep = __find_general_cachep(size, flags);
> + if (unlikely(ZERO_OR_NULL_PTR(cachep)))
> + return cachep;

> +
> + ret = __cache_alloc(cachep, flags, caller);
> + trace_kmalloc((unsigned long)caller, ret, size,
> +   cachep->buffer_size, flags);
> +
> + return ret;
> +}
> +
> +void *kmalloc_no_account(size_t size, gfp_t flags)
> +{
> + return __do_kmalloc_no_account(size, flags,
> +   __builtin_return_address(0));
> +}
> +
> void *__kmalloc_track_caller(size_t size, gfp_t flags, unsigned long caller)
> {
>   return __do_kmalloc(size, flags, (void *)caller);

Subject: Re: [PATCH 17/23] kmem controller charge/uncharge infrastructure
Posted by KAMEZAWA Hiroyuki on Wed, 25 Apr 2012 01:56:16 GMT
View Forum Message <> Reply to Message

(2012/04/24 23:22), Frederic Weisbecker wrote:

> On Mon, Apr 23, 2012 at 03:25:59PM -0700, David Rientjes wrote:
>> On Sun, 22 Apr 2012, Glauber Costa wrote:
>>
>>> +/*
>>> + * Return the kmem_cache we're supposed to use for a slab allocation.
>>> + * If we are in interrupt context or otherwise have an allocation that
>>> + * can't fail, we return the original cache.
>>> + * Otherwise, we will try to use the current memcg's version of the cache.
>>> + *
>>> + * If the cache does not exist yet, if we are the first user of it,
>>> + * we either create it immediately, if possible, or create it asynchronously
>>> + * in a workqueue.
>>> + * In the latter case, we will let the current allocation go through with
>>> + * the original cache.
>>> + *
>>> + * This function returns with rcu_read_lock() held.
>>> + */
>>> +struct kmem_cache *__mem_cgroup_get_kmem_cache(struct kmem_cache *cachep,
>>> +   gfp_t gfp)
>>> +{
>>> + struct mem_cgroup *memcg;
>>> + int idx;
>>> +

```
>>> + gfp |=  cachep->allocflags;
>>> +
>>> + if ((current->mm == NULL))
>>> +  return cachep;
>>> +
>>> + if (cachep->memcg_params.memcg)
>>> +  return cachep;
>>> +
>>> + idx = cachep->memcg_params.id;
>>> + VM_BUG_ON(idx == -1);
>>> +
>>> + memcg = mem_cgroup_from_task(current);
>>> + if (!mem_cgroup_kmem_enabled(memcg))
>>> +  return cachep;
>>> +
>>> + if (rcu_access_pointer(memcg->slabs[idx]) == NULL) {
>>> +  memcg_create_cache_enqueue(memcg, cachep);
>>> +  return cachep;
>>> + }
>>> +
>>> + return rcu_dereference(memcg->slabs[idx]);
>>> +}
>>> +EXPORT_SYMBOL(__mem_cgroup_get_kmem_cache);
>>> +
>>> +void mem_cgroup_remove_child_kmem_cache(struct kmem_cache *cachep, int id)
>>> +{
>>> + rcu_assign_pointer(cachep->memcg_params.memcg->slabs[id], NULL);
>>> +}
>>> +
>>> +bool __mem_cgroup_charge_kmem(gfp_t gfp, size_t size)
>>> +{
>>> + struct mem_cgroup *memcg;
>>> + bool ret = true;
>>> +
>>> + rcu_read_lock();
>>> + memcg = mem_cgroup_from_task(current);
>>
>> This seems horribly inconsistent with memcg charging of user memory since
>> it charges to p->mm->owner and you're charging to p.  So a thread attached
>> to a memcg can charge user memory to one memcg while charging slab to
>> another memcg?
>
> Charging to the thread rather than the process seem to me the right behaviour:
> you can have two threads of a same process attached to different cgroups.
>
> Perhaps it is the user memory memcg that needs to be fixed?
>
```

There is a problem of OOM-Kill.
To free memory by killing process, 'mm' should be released by kill.
So, oom-killer just finds a leader of process.

Assume A process X consists of thread A, B and A is thread-group-leader.

Put thread A into cgroup/Gold
  thread B into cgroup/Silver.

If we do accounting based on threads, we can't do anything at OOM in cgroup/Silver.
An idea 'Killing thread-A to kill thread-B'..... breaks isolation.

As far as resources used by process, I think accounting should be done per process.
It's not tied to thread.

About kmem, if we count task_struct, page tables, etc...which can be freed by
OOM-Killer i.e. it's allocated for 'process', should be aware of OOM problem.
Using mm->owner makes sense to me until someone finds a great idea to handle
OOM situation rather than task killing.

Thanks,
-Kame

Subject: Re: [PATCH 16/23] slab: provide kmalloc_no_account
Posted by Glauber Costa on Wed, 25 Apr 2012 14:29:22 GMT
View Forum Message <> Reply to Message

On 04/24/2012 10:44 PM, KAMEZAWA Hiroyuki wrote:
> (2012/04/23 8:53), Glauber Costa wrote:
>
>> Some allocations need to be accounted to the root memcg regardless
>> of their context. One trivial example, is the allocations we do
>> during the memcg slab cache creation themselves. Strictly speaking,
>> they could go to the parent, but it is way easier to bill them to
>> the root cgroup.
>>
>> Only generic kmalloc allocations are allowed to be bypassed.
>>
>> The function is not exported, because drivers code should always
>> be accounted.
>>
>> This code is mosly written by Suleiman Souhlal.
>>
>> 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>
>> CC: Suleiman Souhlal<suleiman@google.com>
>
>
> Seems reasonable.
> Reviewed-by: KAMEZAWA Hiroyuki<kamezawa.hiroyu@jp.fujitsu.com>
>
> Hmm...but can't we find the 'context' in automatic way ?
>

Not that I can think of. Well, actually, not without adding some tests
to the allocation path I'd rather not (like testing for the return
address and then doing a table lookup, etc)

An option would be to store it in the task_struct. So we would allocate
as following:

memcg_skip_account_start(p);
do_a_bunch_of_allocations();
memcg_skip_account_stop(p);

The problem with that, is that it is quite easy to abuse.
but if we don't export that to modules, it would be acceptable.

Question is, given the fact that the number of kmalloc_no_account() is
expected to be really small, is it worth it?

Subject: Re: [PATCH 11/23] slub: consider a memcg parameter in kmem_create_cache
Posted by Glauber Costa on Wed, 25 Apr 2012 14:37:22 GMT
View Forum Message <> Reply to Message

On 04/24/2012 10:38 PM, KAMEZAWA Hiroyuki wrote:
> (2012/04/21 6:57), Glauber Costa wrote:
>
>> Allow a memcg parameter to be passed during cache creation.
>> The slub allocator will only merge caches that belong to
>> the same memcg.
>>
>> Default function is created as a wrapper, passing NULL
>> to the memcg version. We only merge caches that belong
>> to the same memcg.
>>
>> > From the memcontrol.c side, 3 helper functions are created:
>>
>> 1) memcg_css_id: because slub needs a unique cache name

>> for sysfs. Since this is visible, but not the canonical
>> location for slab data, the cache name is not used, the
>> css_id should suffice.
>>
>> 2) mem_cgroup_register_cache: is responsible for assigning
>> a unique index to each cache, and other general purpose
>> setup. The index is only assigned for the root caches. All
>> others are assigned index == -1.
>>
>> 3) mem_cgroup_release_cache: can be called from the root cache
>> destruction, and will release the index for other caches.
>>
>> This index mechanism was developed by Suleiman Souhlal.
>>
>> 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>
>> CC: Suleiman Souhlal<suleiman@google.com>
>> ---
>> include/linux/memcontrol.h | 14 +++++++++++++
>> include/linux/slab.h | 6 ++++++
>> mm/memcontrol.c | 29 ++++++++++++++++++++++++++++++
>> mm/slub.c | 31 ++++++++++++++++++++++++++++++----
>> 4 files changed, 76 insertions(+), 4 deletions(-)
>>
>> diff --git a/include/linux/memcontrol.h b/include/linux/memcontrol.h
>> index f94efd2..99e14b9 100644
>> --- a/include/linux/memcontrol.h
>> +++ b/include/linux/memcontrol.h
>> @@ -26,6 +26,7 @@ struct mem_cgroup;
>> struct page_cgroup;
>> struct page;
>> struct mm_struct;
>> +struct kmem_cache;
>>
>> /* Stats that can be updated by kernel. */
>> enum mem_cgroup_page_stat_item {
>> @@ -440,7 +441,20 @@ struct sock;
>> #ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
>> void sock_update_memcg(struct sock *sk);
>> void sock_release_memcg(struct sock *sk);
>> +int memcg_css_id(struct mem_cgroup *memcg);
>> +void mem_cgroup_register_cache(struct mem_cgroup *memcg,
>> + struct kmem_cache *s);
>> +void mem_cgroup_release_cache(struct kmem_cache *cachep);

>> #else
>> +static inline void mem_cgroup_register_cache(struct mem_cgroup *memcg,
>> +   struct kmem_cache *s)
>> +{
>> +}
>> +
>> +static inline void mem_cgroup_release_cache(struct kmem_cache *cachep)
>> +{
>> +}
>> +
>> static inline void sock_update_memcg(struct sock *sk)
>> {
>> }
>> diff --git a/include/linux/slab.h b/include/linux/slab.h
>> index a5127e1..c7a7e05 100644
>> --- a/include/linux/slab.h
>> +++ b/include/linux/slab.h
>> @@ -321,6 +321,12 @@ extern void *__kmalloc_track_caller(size_t, gfp_t, unsigned long);
>>   __kmalloc(size, flags)
>> #endif /* DEBUG_SLAB */
>>
>> +#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
>> +#define MAX_KMEM_CACHE_TYPES 400
>> +#else
>> +#define MAX_KMEM_CACHE_TYPES 0
>> +#endif /* CONFIG_CGROUP_MEM_RES_CTLR_KMEM */
>> +
>
>
> why 400 ?

Quite arbitrary. Just large enough to hold all caches there are
currently in a system + modules. (Right now I have around 140
in a normal fedora installation)

>> +/* Bitmap used for allocating the cache id numbers. */
>> +static DECLARE_BITMAP(cache_types, MAX_KMEM_CACHE_TYPES);
>> +
>> +void mem_cgroup_register_cache(struct mem_cgroup *memcg,
>> +   struct kmem_cache *cachep)
>> +{
>> + int id = -1;
>> +
>> + cachep->memcg_params.memcg = memcg;
>> +
>> + if (!memcg) {
>> + id = find_first_zero_bit(cache_types, MAX_KMEM_CACHE_TYPES);
>> + BUG_ON(id< 0 || id>= MAX_KMEM_CACHE_TYPES);

>> + __set_bit(id, cache_types);
>
>
> No lock here ? you need find_first_zero_bit_and_set_atomic() or some.
> Rather than that, I think you can use lib/idr.c::ida_simple_get().

This function is called from within kmem_cache_create(), that usually
already do locking. The slub, for instance, uses the slub_lock() for all
cache creation, and the slab do something quite similar. (All right, I
should have mentioned that in comments)

But as for idr, I don't think it is a bad idea. I will take a look.

>> @@ -3880,7 +3881,7 @@ static int slab_unmergeable(struct kmem_cache *s)
>>   return 0;
>>   }
>>
>> -static struct kmem_cache *find_mergeable(size_t size,
>> +static struct kmem_cache *find_mergeable(struct mem_cgroup *memcg, size_t size,
>>   size_t align, unsigned long flags, const char *name,
>>   void (*ctor)(void *))
>>   {
>> @@ -3916,21 +3917,29 @@ static struct kmem_cache *find_mergeable(size_t size,
>>   if (s->size - size>= sizeof(void *))
>>   continue;
>>
>> + if (memcg&& s->memcg_params.memcg != memcg)
>> +   continue;
>> +
>>   return s;
>>   }
>>   return NULL;
>>   }
>>
>> -struct kmem_cache *kmem_cache_create(const char *name, size_t size,
>> - size_t align, unsigned long flags, void (*ctor)(void *))
>> +struct kmem_cache *
>> +kmem_cache_create_memcg(struct mem_cgroup *memcg, const char *name, size_t size,
>> +   size_t align, unsigned long flags, void (*ctor)(void *))
>>   {
>>   struct kmem_cache *s;
>>
>>   if (WARN_ON(!name))
>>   return NULL;
>>
>> +#ifndef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
>> + WARN_ON(memcg != NULL);
>> +#endif

>
>
> I'm sorry what's is this warning for ?
this is inside ifndef (not defined), so this means anyone trying to pass
a memcg in that situation, is doing something really wrong.

I was actually going for BUG() on this one, but changed my mind

Thinking again, I could probably do this:

```
if (WARN_ON(memcg != NULL))
   memcg = NULL;
```

this way we can keep going without killing the kernel as well as
protecting the function.

>
>> @@ -5265,6 +5283,11 @@ static char *create_unique_id(struct kmem_cache *s)
>>   if (p != name + 1)
>>   *p++ = '-';
>>   p += sprintf(p, "%07d", s->size);
>> +
>> +#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
>> + if (s->memcg_params.memcg)
>> + p += sprintf(p, "-%08d", memcg_css_id(s->memcg_params.memcg));
>> +#endif
>>   BUG_ON(p> name + ID_STR_LENGTH - 1);
>>   return name;
>>   }
>
>
> So, you use 'id' in user interface. Should we provide 'id' as memory.id file ?

We could.
But that is not the cache name, this is for alias files.

The cache name has css_id:dcache_name, so we'll see something like
2:container1

The css_id plays the role of avoiding name duplicates, since all we use
is the last dentry to derive the name.

So I guess if need arises to go search in sysfs for the slub stuff, it
gets easy enough to correlate so we don't need to export the id.

Subject: Re: [PATCH 09/23] kmem slab accounting basic infrastructure

Posted by Glauber Costa on Wed, 25 Apr 2012 14:38:57 GMT
View Forum Message <> Reply to Message

On 04/24/2012 10:32 PM, KAMEZAWA Hiroyuki wrote:
> (2012/04/21 6:57), 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.
>>
>> The code is not enabled until a limit is set. This can be tested by the flag
>> "kmem_accounted". This means that after the patch is applied, no behavioral
>> changes exists for whoever is still using memcg to control their memory usage.
>>
>
> Hmm, res_counter never goes naeative ?

Why would it?

This one has more or less the same logic as the sock buffers.

If we are not accounted, the caches don't get created. If the caches
don't get created, we don't release them. (this is modulo bugs, of course)

Subject: Re: [PATCH 17/23] kmem controller charge/uncharge infrastructure
Posted by Glauber Costa on Wed, 25 Apr 2012 14:43:28 GMT
View Forum Message <> Reply to Message

On 04/24/2012 07:54 PM, David Rientjes wrote:
> On Tue, 24 Apr 2012, Glauber Costa wrote:
>
>>> Yes, for user memory, I see charging to p->mm->owner as allowing that
>>> process to eventually move and be charged to a different memcg and there's
>>> no way to do proper accounting if the charge is split amongst different
>>> memcgs because of thread membership to a set of memcgs. This is
>>> consistent with charges for shared memory being moved when a thread
>>> mapping it moves to a new memcg, as well.
>>
>> But that's the problem.
>>

>> When we are dealing with kernel memory, we are allocating a whole slab page.
>> It is essentially impossible to track, given a page, which task allocated
>> which object.
>>
>
> Right, so you have to make the distinction that slab charges cannot be
> migrated by memory.move_charge_at_immigrate (and it's not even specified
> to do anything beyond user pages in Documentation/cgroups/memory.txt),

Never intended to.

> but
> it would be consistent to charge the same memcg for a process's slab
> allocations as the process's user allocations.
>
> My response was why we shouldn't be charging user pages to
> mem_cgroup_from_task(current) rather than
> mem_cgroup_from_task(current->mm->owner) which is what is currently
> implemented.

Ah, all right. Well, for user memory I agree with you. My point was
exactly that user memory can always be pinpointed to a specific address
space, while kernel memory can't.

>
> If that can't be changed so that we can still migrate user memory amongst
> memcgs for memory.move_charge_at_immigrate, then it seems consistent to
> have all allocations done by a task to be charged to the same memcg.
> Hence, I suggested current->mm->owner for slab charging as well.

All right. This can be done. Although I don't see this as a must for
slab as already explained, I certainly don't oppose doing so as well.

Subject: Re: [PATCH 17/23] kmem controller charge/uncharge infrastructure
Posted by Glauber Costa on Wed, 25 Apr 2012 14:44:22 GMT
View Forum Message <> Reply to Message

>
> About kmem, if we count task_struct, page tables, etc...which can be freed by
> OOM-Killer i.e. it's allocated for 'process', should be aware of OOM problem.
> Using mm->owner makes sense to me until someone finds a great idea to handle
> OOM situation rather than task killing.
>

noted, will update.

Subject: Re: [PATCH 09/23] kmem slab accounting basic infrastructure
Posted by KAMEZAWA Hiroyuki on Thu, 26 Apr 2012 00:08:39 GMT
View Forum Message <> Reply to Message

(2012/04/25 23:38), Glauber Costa wrote:

> On 04/24/2012 10:32 PM, KAMEZAWA Hiroyuki wrote:
>> (2012/04/21 6:57), 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.
>>>
>>> The code is not enabled until a limit is set. This can be tested by the flag
>>> "kmem_accounted". This means that after the patch is applied, no behavioral
>>> changes exists for whoever is still using memcg to control their memory usage.
>>>
>>
>> Hmm, res_counter never goes naeative ?
>
> Why would it?
>
> This one has more or less the same logic as the sock buffers.
>
> If we are not accounted, the caches don't get created. If the caches
> don't get created, we don't release them. (this is modulo bugs, of course)

Okay. Please note how the logic works in description or Doc.
It's a bit complicated part.

Thanks,
-Kame

Subject: Re: [PATCH 16/23] slab: provide kmalloc_no_account
Posted by KAMEZAWA Hiroyuki on Thu, 26 Apr 2012 00:13:51 GMT
View Forum Message <> Reply to Message

(2012/04/25 23:29), Glauber Costa wrote:

> On 04/24/2012 10:44 PM, KAMEZAWA Hiroyuki wrote:

>> (2012/04/23 8:53), Glauber Costa wrote:
>>
>>> Some allocations need to be accounted to the root memcg regardless
>>> of their context. One trivial example, is the allocations we do
>>> during the memcg slab cache creation themselves. Strictly speaking,
>>> they could go to the parent, but it is way easier to bill them to
>>> the root cgroup.
>>>
>>> Only generic kmalloc allocations are allowed to be bypassed.
>>>
>>> The function is not exported, because drivers code should always
>>> be accounted.
>>>
>>> This code is mosly written by Suleiman Souhlal.
>>>
>>> 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>
>>> CC: Suleiman Souhlal<suleiman@google.com>
>>
>>
>> Seems reasonable.
>> Reviewed-by: KAMEZAWA Hiroyuki<kamezawa.hiroyu@jp.fujitsu.com>
>>
>> Hmm...but can't we find the 'context' in automatic way ?
>>
>
> Not that I can think of. Well, actually, not without adding some tests
> to the allocation path I'd rather not (like testing for the return
> address and then doing a table lookup, etc)
>
> An option would be to store it in the task_struct. So we would allocate
> as following:
>
> memcg_skip_account_start(p);
> do_a_bunch_of_allocations();
> memcg_skip_account_stop(p);
>
> The problem with that, is that it is quite easy to abuse.
> but if we don't export that to modules, it would be acceptable.
>
> Question is, given the fact that the number of kmalloc_no_account() is
> expected to be really small, is it worth it?
>

ok, but.... There was an idea __GFP_NOACCOUNT, which is better ?
Are you afraid that__GFP_NOACCOUNT can be spread too much rather than
kmalloc_no_account() ?

Thanks,
-Kame

Subject: Re: [PATCH 13/23] slub: create duplicate cache
Posted by Frederic Weisbecker on Thu, 26 Apr 2012 13:10:55 GMT
View Forum Message <> Reply to Message

On Tue, Apr 24, 2012 at 11:37:59AM -0300, Glauber Costa wrote:
> On 04/24/2012 11:18 AM, Frederic Weisbecker wrote:
> >On Sun, Apr 22, 2012 at 08:53:30PM -0300, Glauber Costa wrote:
> >>This patch provides kmem_cache_dup(), that duplicates
> >>a cache for a memcg, preserving its creation properties.
> >>Object size, alignment and flags are all respected.
> >>
> >>When a duplicate cache is created, the parent cache cannot
> >>be destructed during the child lifetime. To assure this,
> >>its reference count is increased if the cache creation
> >>succeeds.
> >>
> >>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>
> >>CC: Suleiman Souhlal<suleiman@google.com>
> >>---
> >> include/linux/memcontrol.h | 3 +++
> >> include/linux/slab.h | 3 +++
> >> mm/memcontrol.c | 44 ++
> >> mm/slub.c | 37 +++++++++++++++++++++++++++++++++++++
> >> 4 files changed, 87 insertions(+), 0 deletions(-)
> >>
> >>diff --git a/include/linux/memcontrol.h b/include/linux/memcontrol.h
> >>index 99e14b9..493ecdd 100644
> >>--- a/include/linux/memcontrol.h
> >>+++ b/include/linux/memcontrol.h
> >>@@ -445,6 +445,9 @@ int memcg_css_id(struct mem_cgroup *memcg);
> >> void mem_cgroup_register_cache(struct mem_cgroup *memcg,
> >>   struct kmem_cache *s);
> >> void mem_cgroup_release_cache(struct kmem_cache *cachep);
> >>+extern char *mem_cgroup_cache_name(struct mem_cgroup *memcg,

> >>+ struct kmem_cache *cachep);
> >>+
> >> #else
> >> static inline void mem_cgroup_register_cache(struct mem_cgroup *memcg,
> >> struct kmem_cache *s)
> >>diff --git a/include/linux/slab.h b/include/linux/slab.h
> >>index c7a7e05..909b508 100644
> >>--- a/include/linux/slab.h
> >>+++ b/include/linux/slab.h
> >>@@ -323,6 +323,9 @@ extern void *__kmalloc_track_caller(size_t, gfp_t, unsigned long);
> >>
> >> #ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
> >> #define MAX_KMEM_CACHE_TYPES 400
> >>+extern struct kmem_cache *kmem_cache_dup(struct mem_cgroup *memcg,
> >>+ struct kmem_cache *cachep);
> >>+void kmem_cache_drop_ref(struct kmem_cache *cachep);
> >> #else
> >> #define MAX_KMEM_CACHE_TYPES 0
> >> #endif /* CONFIG_CGROUP_MEM_RES_CTLR_KMEM */
> >>diff --git a/mm/memcontrol.c b/mm/memcontrol.c
> >>index 0015ed0..e881d83 100644
> >>--- a/mm/memcontrol.c
> >>+++ b/mm/memcontrol.c
> >>@@ -467,6 +467,50 @@ struct cg_proto *tcp_proto_cgroup(struct mem_cgroup *memcg)
> >> EXPORT_SYMBOL(tcp_proto_cgroup);
> >> #endif /* CONFIG_INET */
> >>
> >>+/*
> >>+ * This is to prevent races againt the kmalloc cache creations.
> >>+ * Should never be used outside the core memcg code. Therefore,
> >>+ * copy it here, instead of letting it in lib/
> >>+ */
> >>+static char *kasprintf_no_account(gfp_t gfp, const char *fmt, ...)
> >>+{
> >>+ unsigned int len;
> >>+ char *p = NULL;
> >>+ va_list ap, aq;
> >>+
> >>+ va_start(ap, fmt);
> >>+ va_copy(aq, ap);
> >>+ len = vsnprintf(NULL, 0, fmt, aq);
> >>+ va_end(aq);
> >>+
> >>+ p = kmalloc_no_account(len+1, gfp);
> >
> >I can't seem to find kmalloc_no_account() in this patch or may be
> >I missed it in a previous one?
>

> It is in a previous one (actually two, one for the slab, one for the
> slub). They are bundled in the cache creation, but I could separate
> it
> for clarity, if you prefer.

They seem to be the 14th and 16th patches. They should probably
be before the current one for review clarity, so we define that function
before it gets used. This is also good to not break bisection.

>
>
> >>+ if (!p)
> >>+ goto out;
> >>+
> >>+ vsnprintf(p, len+1, fmt, ap);
> >>+
> >>+out:
> >>+ va_end(ap);
> >>+ return p;
> >>+}
> >>+
> >>+char *mem_cgroup_cache_name(struct mem_cgroup *memcg, struct kmem_cache
*cachep)
> >>+{
> >>+ char *name;
> >>+ struct dentry *dentry = memcg->css.cgroup->dentry;
> >>+
> >>+ BUG_ON(dentry == NULL);
> >>+
> >>+ /* Preallocate the space for "dead" at the end */
> >>+ name = kasprintf_no_account(GFP_KERNEL, "%s(%d:%s)dead",
> >>+   cachep->name, css_id(&memcg->css), dentry->d_name.name);
> >>+
> >>+ if (name)
> >>+ /* Remove "dead" */
> >>+ name[strlen(name) - 4] = '\0';
> >
> >Why this space for "dead" ?
>
> Ok, sorry. Since I didn't include the destruction part, it got too
> easy for whoever wasn't following the last discussion on this to get
> lost - My bad. So here it is:
>
> When we destroy the memcg, some objects may still hold the cache in
> memory. It is like a reference count, in a sense, which each object
> being a reference.
>
> In typical cases, like non-shrinkable caches that has create -

> destroy patterns, the caches will go away as soon as the tasks using
> them.
>
> But in cache-like structure like the dentry cache, the objects may
> hang around until a shrinker pass takes them out. And even then,
> some of them will live on.
>
> In this case, we will display them with "dead" in the name.

Ok.

>
> We could hide them, but then it gets weirder because it would be
> hard to understand where is your used memory when you need to
> inspect your system.
>
> Creating another file, slabinfo_deadcaches, and keeping the names,
> is also a possibility, if people think that the string append is way
> too ugly.

Ok, thanks for the explanation.

Subject: Re: [PATCH 17/23] kmem controller charge/uncharge infrastructure
Posted by Frederic Weisbecker on Fri, 27 Apr 2012 11:38:45 GMT
View Forum Message <> Reply to Message

On Tue, Apr 24, 2012 at 01:21:43PM -0700, David Rientjes wrote:
> On Tue, 24 Apr 2012, Frederic Weisbecker wrote:
>
> > > This seems horribly inconsistent with memcg charging of user memory since
> > > it charges to p->mm->owner and you're charging to p. So a thread attached
> > > to a memcg can charge user memory to one memcg while charging slab to
> > > another memcg?
> >
> > Charging to the thread rather than the process seem to me the right behaviour:
> > you can have two threads of a same process attached to different cgroups.
> >
> > Perhaps it is the user memory memcg that needs to be fixed?
> >
>
> No, because memory is represented by mm_struct, not task_struct, so you
> must charge to p->mm->owner to allow for moving threads amongst memcgs
> later for memory.move_charge_at_immigrate. You shouldn't be able to
> charge two different memcgs for memory represented by a single mm.

The idea I had was more that only the memcg of the thread that does the allocation
is charged. But the problem is that this allocation can be later deallocated

from another thread. So probably charging the owner is indeed the only sane
way to go with user memory.

Subject: Re: [PATCH 17/23] kmem controller charge/uncharge infrastructure
Posted by Frederic Weisbecker on Fri, 27 Apr 2012 12:22:22 GMT
View Forum Message <> Reply to Message

On Wed, Apr 25, 2012 at 10:56:16AM +0900, KAMEZAWA Hiroyuki wrote:
> (2012/04/24 23:22), Frederic Weisbecker wrote:
>
> > On Mon, Apr 23, 2012 at 03:25:59PM -0700, David Rientjes wrote:
> >> On Sun, 22 Apr 2012, Glauber Costa wrote:
> >>
> >>> +/*
> >>> + * Return the kmem_cache we're supposed to use for a slab allocation.
> >>> + * If we are in interrupt context or otherwise have an allocation that
> >>> + * can't fail, we return the original cache.
> >>> + * Otherwise, we will try to use the current memcg's version of the cache.
> >>> + *
> >>> + * If the cache does not exist yet, if we are the first user of it,
> >>> + * we either create it immediately, if possible, or create it asynchronously
> >>> + * in a workqueue.
> >>> + * In the latter case, we will let the current allocation go through with
> >>> + * the original cache.
> >>> + *
> >>> + * This function returns with rcu_read_lock() held.
> >>> + */
> >>> +struct kmem_cache *__mem_cgroup_get_kmem_cache(struct kmem_cache *cachep,
> >>> +   gfp_t gfp)
> >>> +{
> >>> +   struct mem_cgroup *memcg;
> >>> +   int idx;
> >>> +
> >>> +   gfp |= cachep->allocflags;
> >>> +
> >>> +   if ((current->mm == NULL))
> >>> +   return cachep;
> >>> +
> >>> +   if (cachep->memcg_params.memcg)
> >>> +   return cachep;
> >>> +
> >>> +   idx = cachep->memcg_params.id;
> >>> +   VM_BUG_ON(idx == -1);
> >>> +
> >>> +   memcg = mem_cgroup_from_task(current);
> >>> +   if (!mem_cgroup_kmem_enabled(memcg))
> >>> +   return cachep;

> >>> +
> >>> + if (rcu_access_pointer(memcg->slabs[idx]) == NULL) {
> >>> + memcg_create_cache_enqueue(memcg, cachep);
> >>> + return cachep;
> >>> + }
> >>> +
> >>> + return rcu_dereference(memcg->slabs[idx]);
> >>> +}
> >>> +EXPORT_SYMBOL(__mem_cgroup_get_kmem_cache);
> >>> +
> >>> +void mem_cgroup_remove_child_kmem_cache(struct kmem_cache *cachep, int id)
> >>> +{
> >>> + rcu_assign_pointer(cachep->memcg_params.memcg->slabs[id], NULL);
> >>> +}
> >>> +
> >>> +bool __mem_cgroup_charge_kmem(gfp_t gfp, size_t size)
> >>> +{
> >>> + struct mem_cgroup *memcg;
> >>> + bool ret = true;
> >>> +
> >>> + rcu_read_lock();
> >>> + memcg = mem_cgroup_from_task(current);
> >>
> >> This seems horribly inconsistent with memcg charging of user memory since
> >> it charges to p->mm->owner and you're charging to p. So a thread attached
> >> to a memcg can charge user memory to one memcg while charging slab to
> >> another memcg?
> >
> > Charging to the thread rather than the process seem to me the right behaviour:
> > you can have two threads of a same process attached to different cgroups.
> >
> > Perhaps it is the user memory memcg that needs to be fixed?
> >
>
> There is a problem of OOM-Kill.
> To free memory by killing process, 'mm' should be released by kill.
> So, oom-killer just finds a leader of process.
>
> Assume A process X consists of thread A, B and A is thread-group-leader.
>
> Put thread A into cgroup/Gold
>   thread B into cgroup/Silver.
>
> If we do accounting based on threads, we can't do anything at OOM in cgroup/Silver.
> An idea 'Killing thread-A to kill thread-B'..... breaks isolation.

Right. But then if one wanted true isolation without worrying about such
side effect, he would avoid to scatter a thread group across more than one

memcg.

>
> As far as resources used by process, I think accounting should be done per process.
> It's not tied to thread.

Yep, makes sense. Especially as thread B might free memory allocated by thread A.
Maintaining a per thread granularity would create too much mess.

> About kmem, if we count task_struct, page tables, etc...which can be freed by
> OOM-Killer i.e. it's allocated for 'process', should be aware of OOM problem.
> Using mm->owner makes sense to me until someone finds a great idea to handle
> OOM situation rather than task killing.

kmem is different because the memory allocated is in essence available to every
threads. Because this becomes a global resource, I don't find the accounting to p->mm->owner
more relevant than to p.

Subject: Re: [PATCH 17/23] kmem controller charge/uncharge infrastructure
Posted by David Rientjes on Fri, 27 Apr 2012 18:13:31 GMT
View Forum Message <> Reply to Message

On Fri, 27 Apr 2012, Frederic Weisbecker wrote:

> > No, because memory is represented by mm_struct, not task_struct, so you
> > must charge to p->mm->owner to allow for moving threads amongst memcgs
> > later for memory.move_charge_at_immigrate. You shouldn't be able to
> > charge two different memcgs for memory represented by a single mm.
>
> The idea I had was more that only the memcg of the thread that does the allocation
> is charged. But the problem is that this allocation can be later deallocated
> from another thread. So probably charging the owner is indeed the only sane
> way to go with user memory.
>

It's all really the same concept: if we want to move memory of a process,
willingly free memory in the process itself, or free memory of a process
by way of the oom killer, we need a way to do that for the entire process
so the accounting makes sense afterwards. And since we have that
requirement for user memory, it makes sense that its consistent with slab
as well. I don't think a thread of a process should be able to charge
slab to one memcg while its user memory is charged to another memcg.

Subject: [PATCH 0/3] A few fixes for '[PATCH 00/23] slab+slub accounting for memcg' series

Hello Glauber,

On Fri, Apr 20, 2012 at 06:57:08PM -0300, Glauber Costa wrote:
> This is my current attempt at getting the kmem controller
> into a mergeable state. IMHO, all the important bits are there, and it should't
> change *that* much from now on. I am, however, expecting at least a couple more
> interactions before we sort all the edges out.
>
> This series works for both the slub and the slab. One of my main goals was to
> make sure that the interfaces we are creating actually makes sense for both
> allocators.
>
> I did some adaptations to the slab-specific patches, but the bulk of it
> comes from Suleiman's patches. I did the best to use his patches
> as-is where possible so to keep authorship information. When not possible,
> I tried to be fair and quote it in the commit message.
>
> In this series, all existing caches are created per-memcg after its first hit.
> The main reason is, during discussions in the memory summit we came into
> agreement that the fragmentation problems that could arise from creating all
> of them are mitigated by the typically small quantity of caches in the system
> (order of a few megabytes total for sparsely used caches).
> The lazy creation from Suleiman is kept, although a bit modified. For instance,
> I now use a locked scheme instead of cmpxcgh to make sure cache creation won't
> fail due to duplicates, which simplifies things by quite a bit.
>
> The slub is a bit more complex than what I came up with in my slub-only
> series. The reason is we did not need to use the cache-selection logic
> in the allocator itself - it was done by the cache users. But since now
> we are lazy creating all caches, this is simply no longer doable.
>
> I am leaving destruction of caches out of the series, although most
> of the infrastructure for that is here, since we did it in earlier
> series. This is basically because right now Kame is reworking it for
> user memcg, and I like the new proposed behavior a lot more. We all seemed
> to have agreed that reclaim is an interesting problem by itself, and
> is not included in this already too complicated series. Please note
> that this is still marked as experimental, so we have so room. A proper
> shrinker implementation is a hard requirement to take the kmem controller
> out of the experimental state.
>
> I am also not including documentation, but it should only be a matter
> of merging what we already wrote in earlier series plus some additions.

The patches look great, thanks a lot for your work!

I finally tried them, and after a few fixes the kmem accounting
seems to work fine with slab. The fixes will follow this email,
and if they're fine, feel free to fold them into your patches.

However, with slub I'm getting kernel hangs and various traces[1].
It seems that kernel memcg recurses when trying to call
memcg_create_cache_enqueue() -- it calls kmalloc_no_account()
which was introduced to not recurse into memcg, but looking
into 'slub: provide kmalloc_no_account' patch, I don't see
any difference between _no_account and ordinary kmalloc. Hm.

OK, slub apart... the accounting works with slab, which is great.

There's another, more generic question: is there any particular
reason why you don't want to account slab memory for root cgroup?

Personally I'm interested in kmem accounting because I use
memcg for lowmemory notifications. I'm installing events
on the root's memory.usage_in_bytes, and the thresholds values
are calculated like this:

  total_ram - wanted_threshold

So, if we want to get a notification when there's 64 MB memory
left on a 256 MB machine, we'd install an event on the 194 MB
mark (the good thing about usage_in_bytes, is that it does
account file caches, so the formula is simple).

Obviously, without kmem accounting the formula can be very
imprecise when kernel (e.g. hw drivers) itself start using a
lot of memory. With root's slab accounting the problem
would be solved, but for some reason you deliberately do not
want to account it for root cgroup. I suspect that there are
some performance concerns?..

Thanks,

[1]

BUG: unable to handle kernel paging request at ffffffffb2e80900
IP: [<ffffffff8105940c>] check_preempt_wakeup+0x3c/0x210
PGD 160d067 PUD 1611063 PMD 0
Thread overran stack, or stack corrupted
Oops: 0000 [#1] SMP
CPU 0
Pid: 943, comm: bash Not tainted 3.4.0-rc4+ #34 Bochs Bochs
RIP: 0010:[<ffffffff8105940c>] [<ffffffff8105940c>] check_preempt_wakeup+0x3c/0x210
RSP: 0018:ffff880006305ee8 EFLAGS: 00010006

RAX: 00000000000109c0 RBX: ffff8800071b4e20 RCX: ffff880006306000
RDX: 0000000000000000 RSI: 0000000006306028 RDI: ffff880007c109c0
RBP: ffff880006305f28 R08: 0000000000000000 R09: 0000000000000001
R10: 0000000000000000 R11: 0000000000000000 R12: ffff880007c109c0
R13: ffff88000644ddc0 R14: ffff8800071b4e68 R15: 0000000000000000
FS: 00007fad1244c700(0000) GS:ffff880007c00000(0000) knlGS:0000000000000000
CS: 0010 DS: 0000 ES: 0000 CR0: 0000000080050033
CR2: ffffffffb2e80900 CR3: 00000000063b8000 CR4: 00000000000006b0
DR0: 0000000000000000 DR1: 0000000000000000 DR2: 0000000000000000
DR3: 0000000000000000 DR6: 00000000ffff0ff0 DR7: 0000000000000400
Process bash (pid: 943, threadinfo ffff880006306000, task ffff88000644ddc0)
Stack:
 0000000000000000 ffff88000644de08 ffff880007c109c0 ffff880007c109c0
 ffff8800071b4e20 0000000000000000 0000000000000000 0000000000000000
 ffff880006305f48 ffffffff81053304 ffff880007c109c0 ffff880007c109c0
Call Trace:
Code: 76 48 41 55 41 54 49 89 fc 53 48 89 f3 48 83 ec 18 4c 8b af e0 07 00 00 49 8d 4d 48 48 89 4d c8 49 8b 4d 08 4c 3b 75 c8 8b 71 18 <48> 8b 34 f5 c0 07 65 81 48 8b bc 30 a8 00 00 00 8b 35 3a 3f 5c
RIP [<ffffffff8105940c>] check_preempt_wakeup+0x3c/0x210
 RSP <ffff880006305ee8>
CR2: ffffffffb2e80900
---[end trace 78fa9c86bebb1214]---

--
Anton Vorontsov
Email: cbouatmailru@gmail.com

Subject: [PATCH 1/3] slab: Proper off-slabs handling when duplicating caches
Posted by Anton Vorontsov on Mon, 30 Apr 2012 10:01:21 GMT
View Forum Message <> Reply to Message

OFF_SLAB is not CREATE_MASK bit, so we should clear it before
calling __kmem_cache_create(), otherwise kernel gets very upset,
see below.

As a side effect, now we let slab to reevaluate off-slab
decision, but the decision will be the same, because whether
we do off-slabs only depend on the size and create_mask
bits.

------------[cut here]------------
kernel BUG at mm/slab.c:2376!
invalid opcode: 0000 [#1] SMP
CPU 0
Pid: 14, comm: kworker/0:1 Not tainted 3.4.0-rc4+ #32 Bochs Bochs
RIP: 0010:[<ffffffff810c1839>] [<ffffffff810c1839>] __kmem_cache_create+0x609/0x650

```
RSP: 0018:ffff8800072c9c90  EFLAGS: 00010286
RAX: 0000000000000800 RBX: ffffffff81f26bf8 RCX: 000000000000000b
RDX: 000000000000000c RSI: 000000000000000b RDI: ffff8800065c66f8
RBP: ffff8800072c9d40 R08: ffffffff80002800 R09: 0000000000000000
R10: 0000000000000000 R11: 0000000000000001 R12: ffff8800072c8000
R13: ffff8800072c9fd8 R14: ffffffffffffffff R15: ffff8800072c9d0c
FS:  00007f45eb0f2700(0000) GS:ffff880007c00000(0000) knlGS:0000000000000000
CS:  0010 DS: 0000 ES: 0000 CR0: 000000008005003b
CR2: ffffffffff600400 CR3: 000000000650e000 CR4: 00000000000006b0
DR0: 0000000000000000 DR1: 0000000000000000 DR2: 0000000000000000
DR3: 0000000000000000 DR6: 00000000ffff0ff0 DR7: 0000000000000400
Process kworker/0:1 (pid: 14, threadinfo ffff8800072c8000, task ffff88000725d100)
Stack:
 ffff8800072c9cb0 0000000000000000 ffffc9000000c000 ffffffff81621e80
 ffff8800072c9cc0 ffffffff81621e80 ffff8800072c9d40 ffffffff81355cbf
 ffffffff810c1944 0000000000000000 ffffffff81621ec0 ffffffff80002800
Call Trace:
 [<ffffffff81355cbf>] ? mutex_lock_nested+0x26f/0x340
 [<ffffffff810c1944>] ? kmem_cache_dup+0x44/0x110
 [<ffffffff810c2aa0>] ? memcg_create_kmem_cache+0xd0/0xd0
 [<ffffffff810c196b>] kmem_cache_dup+0x6b/0x110
 [<ffffffff810c2a73>] memcg_create_kmem_cache+0xa3/0xd0
 [<ffffffff810c2b1a>] memcg_create_cache_work_func+0x7a/0xe0
 [<ffffffff810405d4>] process_one_work+0x174/0x450
 [<ffffffff81040576>] ? process_one_work+0x116/0x450
 [<ffffffff81040e53>] worker_thread+0x123/0x2d0
 [<ffffffff81040d30>] ? manage_workers.isra.27+0x120/0x120
 [<ffffffff8104639e>] kthread+0x8e/0xa0

Signed-off-by: Anton Vorontsov <anton.vorontsov@linaro.org>
---
 mm/slab.c |    7 +++++++
 1 file changed, 7 insertions(+)

diff --git a/mm/slab.c b/mm/slab.c
index eed72ac..dff87ef 100644
--- a/mm/slab.c
+++ b/mm/slab.c
@@ -2619,6 +2619,13 @@ kmem_cache_dup(struct mem_cgroup *memcg, struct kmem_cache
*cachep)
   return NULL;

  flags = cachep->flags & ~SLAB_PANIC;
+ /*
+  * OFF_SLAB is not CREATE_MASK bit, so we should clear it before
+  * calling __kmem_cache_create(). As a side effect, we let slab
+  * to reevaluate off-slab decision; but that is OK, as the bit
+  * is automatically set depending on the size and other flags.
```

```
+  */
+ flags &= ~CFLGS_OFF_SLAB;
  mutex_lock(&cache_chain_mutex);
  new = __kmem_cache_create(memcg, name, obj_size(cachep),
      cachep->memcg_params.orig_align, flags, cachep->ctor);
--
1.7.9.2
```

Subject: [PATCH 2/3] slab: Fix imbalanced rcu locking
Posted by Anton Vorontsov on Mon, 30 Apr 2012 10:01:39 GMT
View Forum Message <> Reply to Message

Not sure why the code tries to unlock the rcu. The
only case where slab grabs the lock is around
mem_cgroup_get_kmem_cache() call, which won't result
into calling cache_grow() (that tries to unlock the rcu).

```
=====================================
[ BUG: bad unlock balance detected! ]
3.4.0-rc4+ #33 Not tainted
-------------------------------------
swapper/0/0 is trying to release lock (rcu_read_lock) at:
[<ffffffff8134f0b4>] cache_grow.constprop.63+0xe8/0x371
but there are no more locks to release!

other info that might help us debug this:
no locks held by swapper/0/0.

stack backtrace:
Pid: 0, comm: swapper/0 Not tainted 3.4.0-rc4+ #33
Call Trace:
 [<ffffffff8134f0b4>] ? cache_grow.constprop.63+0xe8/0x371
 [<ffffffff8134cf09>] print_unlock_inbalance_bug.part.26+0xd1/0xd9
 [<ffffffff8134f0b4>] ? cache_grow.constprop.63+0xe8/0x371
 [<ffffffff8106865e>] print_unlock_inbalance_bug+0x4e/0x50
 [<ffffffff8134f0b4>] ? cache_grow.constprop.63+0xe8/0x371
 [<ffffffff8106cb26>] __lock_release+0xd6/0xe0
 [<ffffffff8106cb8c>] lock_release+0x5c/0x80
 [<ffffffff8134f0cc>] cache_grow.constprop.63+0x100/0x371
 [<ffffffff8134f5c6>] cache_alloc_refill+0x289/0x2dc
 [<ffffffff810bf682>] ? kmem_cache_alloc+0x92/0x260
 [<ffffffff81676a0f>] ? pidmap_init+0x79/0xb2
 [<ffffffff810bf842>] kmem_cache_alloc+0x252/0x260
 [<ffffffff810bf5f0>] ? kmem_freepages+0x180/0x180
 [<ffffffff81676a0f>] pidmap_init+0x79/0xb2
 [<ffffffff81667aa3>] start_kernel+0x297/0x2f8
 [<ffffffff8166769e>] ? repair_env_string+0x5a/0x5a
```

```
 [<ffffffff816672fd>] x86_64_start_reservations+0x101/0x105
 [<ffffffff816673f1>] x86_64_start_kernel+0xf0/0xf7

Signed-off-by: Anton Vorontsov <anton.vorontsov@linaro.org>
---
 include/linux/slab_def.h |    2 --
 1 file changed, 2 deletions(-)

diff --git a/include/linux/slab_def.h b/include/linux/slab_def.h
index c4f7e45..2d371ae 100644
--- a/include/linux/slab_def.h
+++ b/include/linux/slab_def.h
@@ -245,13 +245,11 @@ mem_cgroup_kmem_cache_prepare_sleep(struct kmem_cache
*cachep)
   * enabled.
   */
  kmem_cache_get_ref(cachep);
- rcu_read_unlock();
 }

 static inline void
 mem_cgroup_kmem_cache_finish_sleep(struct kmem_cache *cachep)
 {
- rcu_read_lock();
  kmem_cache_drop_ref(cachep);
 }

--
1.7.9.2
```

Subject: [PATCH 3/3] slab: Get rid of mem_cgroup_put_kmem_cache()
Posted by Anton Vorontsov on Mon, 30 Apr 2012 10:02:02 GMT
View Forum Message <> Reply to Message

The function is no longer used, so can be safely removed.

```
Signed-off-by: Anton Vorontsov <anton.vorontsov@linaro.org>
---
 include/linux/slab_def.h |   11 -----------
 1 file changed, 11 deletions(-)

diff --git a/include/linux/slab_def.h b/include/linux/slab_def.h
index 2d371ae..72ea626 100644
--- a/include/linux/slab_def.h
+++ b/include/linux/slab_def.h
@@ -232,12 +232,6 @@ kmem_cache_get_ref(struct kmem_cache *cachep)
 }
```

```
 static inline void
-mem_cgroup_put_kmem_cache(struct kmem_cache *cachep)
-{
- rcu_read_unlock();
-}
-
-static inline void
 mem_cgroup_kmem_cache_prepare_sleep(struct kmem_cache *cachep)
 {
  /*
@@ -266,11 +260,6 @@ kmem_cache_drop_ref(struct kmem_cache *cachep)
 }

 static inline void
-mem_cgroup_put_kmem_cache(struct kmem_cache *cachep)
-{
-}
-
-static inline void
 mem_cgroup_kmem_cache_prepare_sleep(struct kmem_cache *cachep)
 {
 }
--
1.7.9.2
```

Subject: Re: [PATCH 09/23] kmem slab accounting basic infrastructure
Posted by Suleiman Souhlal on Mon, 30 Apr 2012 19:33:53 GMT
View Forum Message <> Reply to Message

On Fri, Apr 20, 2012 at 2:57 PM, Glauber Costa <glommer@parallels.com> 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.
>
> The code is not enabled until a limit is set. This can be tested by the flag
> "kmem_accounted". This means that after the patch is applied, no behavioral
> changes exists for whoever is still using memcg to control their memory usage.
>
> 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)
>
> Signed-off-by: Glauber Costa <glommer@parallels.com>
> CC: Michal Hocko <mhocko@suse.cz>
> CC: Kamezawa Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com>
> CC: Johannes Weiner <hannes@cmpxchg.org>
> ---
> mm/memcontrol.c | 80
++-
> 1 files changed, 79 insertions(+), 1 deletions(-)
>
> diff --git a/mm/memcontrol.c b/mm/memcontrol.c
> index 2810228..36f1e6b 100644
> --- a/mm/memcontrol.c
> +++ b/mm/memcontrol.c
> @@ -252,6 +252,10 @@ 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.
>   */
> @@ -266,6 +270,7 @@ struct mem_cgroup {
>   * Should the accounting and control be hierarchical, per subtree?
>   */
>   bool use_hierarchy;
> +   bool kmem_accounted;
>
>   bool   oom_lock;
>   atomic_t   under_oom;
> @@ -378,6 +383,7 @@ enum res_type {
>   _MEM,
>   _MEMSWAP,
>   _OOM_TYPE,
> +   _KMEM,
> };
>
> #define MEMFILE_PRIVATE(x, val)   (((x) << 16) | (val))

```
> @@ -1470,6 +1476,10 @@ 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));
>  }
>
>  /*
> @@ -3914,6 +3924,11 @@ static ssize_t mem_cgroup_read(struct cgroup *cont, struct cftype
*cft,
>               else
>                       val = res_counter_read_u64(&memcg->memsw, name);
>               break;
> +#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
> +       case _KMEM:
> +              val = res_counter_read_u64(&memcg->kmem, name);
> +              break;
> +#endif
>       default:
>               BUG();
>       }
> @@ -3951,8 +3966,26 @@ 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);
> +#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
> +             else if (type == _KMEM) {
> +                     ret = res_counter_set_limit(&memcg->kmem, val);
> +                     if (ret)
> +                             break;
> +                     /*
> +                      * Once enabled, can't be disabled. We could in theory
> +                      * disable it if we haven't yet created any caches, or
> +                      * if we can shrink them all to death.
> +                      *
> +                      * But it is not worth the trouble
> +                      */
> +                     if (!memcg->kmem_accounted && val != RESOURCE_MAX)
> +                             memcg->kmem_accounted = true;
> +             }
> +#endif
> +             else
```

> +   return -EINVAL;
>   break;
>   case RES_SOFT_LIMIT:
>   ret = res_counter_memparse_write_strategy(buffer, &val);

Why is RESOURCE_MAX special?

-- Suleiman

Subject: Re: [PATCH 10/23] slab/slub: struct memcg_params
Posted by Suleiman Souhlal on Mon, 30 Apr 2012 19:42:28 GMT
View Forum Message <> Reply to Message

On Fri, Apr 20, 2012 at 2:57 PM, Glauber Costa <glommer@parallels.com> wrote:
> For the kmem slab controller, we need to record some extra
> information in the kmem_cache structure.
>
> 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>
> CC: Suleiman Souhlal <suleiman@google.com>
> ---
> include/linux/slab.h | 15 +++++++++++++++
> include/linux/slab_def.h | 4 ++++
> include/linux/slub_def.h | 3 +++
> 3 files changed, 22 insertions(+), 0 deletions(-)
>
> diff --git a/include/linux/slab.h b/include/linux/slab.h
> index a595dce..a5127e1 100644
> --- a/include/linux/slab.h
> +++ b/include/linux/slab.h
> @@ -153,6 +153,21 @@ unsigned int kmem_cache_size(struct kmem_cache *);
> #define ARCH_SLAB_MINALIGN __alignof__(unsigned long long)
> #endif
>
> +#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
> +struct mem_cgroup_cache_params {
> +   struct mem_cgroup *memcg;
> +   int id;
> +
> +#ifdef CONFIG_SLAB
> +   /* Original cache parameters, used when creating a memcg cache */
> +   size_t orig_align;
> +   atomic_t refcnt;

> +
> +#endif
> +	struct list_head destroyed_list; /* Used when deleting cpuset cache */

s,cpuset,memcg,

Sorry about that.

-- Suleiman

Subject: Re: [PATCH 11/23] slub: consider a memcg parameter in kmem_create_cache
Posted by Suleiman Souhlal on Mon, 30 Apr 2012 19:51:18 GMT
View Forum Message <> Reply to Message

On Fri, Apr 20, 2012 at 2:57 PM, Glauber Costa <glommer@parallels.com> wrote:
> Allow a memcg parameter to be passed during cache creation.
> The slub allocator will only merge caches that belong to
> the same memcg.
>
> Default function is created as a wrapper, passing NULL
> to the memcg version. We only merge caches that belong
> to the same memcg.
>
> From the memcontrol.c side, 3 helper functions are created:
>
> 1) memcg_css_id: because slub needs a unique cache name
>   for sysfs. Since this is visible, but not the canonical
>   location for slab data, the cache name is not used, the
>   css_id should suffice.
>
> 2) mem_cgroup_register_cache: is responsible for assigning
>   a unique index to each cache, and other general purpose
>   setup. The index is only assigned for the root caches. All
>   others are assigned index == -1.
>
> 3) mem_cgroup_release_cache: can be called from the root cache
>   destruction, and will release the index for other caches.
>
> This index mechanism was developed by Suleiman Souhlal.
>
> 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>

> CC: Suleiman Souhlal <suleiman@google.com>
> ---
> include/linux/memcontrol.h | 14 +++++++++++++
> include/linux/slab.h | 6 ++++++
> mm/memcontrol.c | 29 +++++++++++++++++++++++++++++
> mm/slub.c | 31 +++++++++++++++++++++++++++++++----
> 4 files changed, 76 insertions(+), 4 deletions(-)
>
> diff --git a/include/linux/memcontrol.h b/include/linux/memcontrol.h
> index f94efd2..99e14b9 100644
> --- a/include/linux/memcontrol.h
> +++ b/include/linux/memcontrol.h
> @@ -26,6 +26,7 @@ struct mem_cgroup;
> struct page_cgroup;
> struct page;
> struct mm_struct;
> +struct kmem_cache;
>
> /* Stats that can be updated by kernel. */
> enum mem_cgroup_page_stat_item {
> @@ -440,7 +441,20 @@ struct sock;
> #ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
> void sock_update_memcg(struct sock *sk);
> void sock_release_memcg(struct sock *sk);
> +int memcg_css_id(struct mem_cgroup *memcg);
> +void mem_cgroup_register_cache(struct mem_cgroup *memcg,
> +   struct kmem_cache *s);
> +void mem_cgroup_release_cache(struct kmem_cache *cachep);
> #else
> +static inline void mem_cgroup_register_cache(struct mem_cgroup *memcg,
> +   struct kmem_cache *s)
> +{
> +}
> +
> +static inline void mem_cgroup_release_cache(struct kmem_cache *cachep)
> +{
> +}
> +
> static inline void sock_update_memcg(struct sock *sk)
> {
> }
> diff --git a/include/linux/slab.h b/include/linux/slab.h
> index a5127e1..c7a7e05 100644
> --- a/include/linux/slab.h
> +++ b/include/linux/slab.h
> @@ -321,6 +321,12 @@ extern void *__kmalloc_track_caller(size_t, gfp_t, unsigned long);
>   __kmalloc(size, flags)
> #endif /* DEBUG_SLAB */

```
>
> +#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
> +#define MAX_KMEM_CACHE_TYPES 400
> +#else
> +#define MAX_KMEM_CACHE_TYPES 0
> +#endif /* CONFIG_CGROUP_MEM_RES_CTLR_KMEM */
> +
>  #ifdef CONFIG_NUMA
>  /*
>  * kmalloc_node_track_caller is a special version of kmalloc_node that
> diff --git a/mm/memcontrol.c b/mm/memcontrol.c
> index 36f1e6b..0015ed0 100644
> --- a/mm/memcontrol.c
> +++ b/mm/memcontrol.c
> @@ -323,6 +323,11 @@ struct mem_cgroup {
>  #endif
>  };
>
> +int memcg_css_id(struct mem_cgroup *memcg)
> +{
> +       return css_id(&memcg->css);
> +}
> +
>  /* Stuffs for move charges at task migration. */
>  /*
>  * Types of charges to be moved. "move_charge_at_immitgrate" is treated as a
> @@ -461,6 +466,30 @@ struct cg_proto *tcp_proto_cgroup(struct mem_cgroup *memcg)
>  }
>  EXPORT_SYMBOL(tcp_proto_cgroup);
>  #endif /* CONFIG_INET */
> +
> +/* Bitmap used for allocating the cache id numbers. */
> +static DECLARE_BITMAP(cache_types, MAX_KMEM_CACHE_TYPES);
> +
> +void mem_cgroup_register_cache(struct mem_cgroup *memcg,
> +                               struct kmem_cache *cachep)
> +{
> +       int id = -1;
> +
> +       cachep->memcg_params.memcg = memcg;
> +
> +       if (!memcg) {
> +               id = find_first_zero_bit(cache_types, MAX_KMEM_CACHE_TYPES);
> +               BUG_ON(id < 0 || id >= MAX_KMEM_CACHE_TYPES);
> +               __set_bit(id, cache_types);
> +       } else
> +               INIT_LIST_HEAD(&cachep->memcg_params.destroyed_list);
> +       cachep->memcg_params.id = id;
```

```
> +}
> +
> +void mem_cgroup_release_cache(struct kmem_cache *cachep)
> +{
> +       __clear_bit(cachep->memcg_params.id, cache_types);
> +}
>  #endif /* CONFIG_CGROUP_MEM_RES_CTLR_KMEM */
>
>  static void drain_all_stock_async(struct mem_cgroup *memcg);
> diff --git a/mm/slub.c b/mm/slub.c
> index 2652e7c..86e40cc 100644
> --- a/mm/slub.c
> +++ b/mm/slub.c
> @@ -32,6 +32,7 @@
>  #include <linux/prefetch.h>
>
>  #include <trace/events/kmem.h>
> +#include <linux/memcontrol.h>
>
>  /*
>  * Lock order:
> @@ -3880,7 +3881,7 @@ static int slab_unmergeable(struct kmem_cache *s)
>         return 0;
>  }
>
> -static struct kmem_cache *find_mergeable(size_t size,
> +static struct kmem_cache *find_mergeable(struct mem_cgroup *memcg, size_t size,
>                 size_t align, unsigned long flags, const char *name,
>                 void (*ctor)(void *))
>  {
> @@ -3916,21 +3917,29 @@ static struct kmem_cache *find_mergeable(size_t size,
>                 if (s->size - size >= sizeof(void *))
>                         continue;
>
> +               if (memcg && s->memcg_params.memcg != memcg)
> +                       continue;
> +
>                 return s;
>         }
>         return NULL;
>  }
>
> -struct kmem_cache *kmem_cache_create(const char *name, size_t size,
> -               size_t align, unsigned long flags, void (*ctor)(void *))
> +struct kmem_cache *
> +kmem_cache_create_memcg(struct mem_cgroup *memcg, const char *name, size_t size,
> +                       size_t align, unsigned long flags, void (*ctor)(void *))
>  {
```

```
>       struct kmem_cache *s;
>
>       if (WARN_ON(!name))
>               return NULL;
>
> +#ifndef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
> +       WARN_ON(memcg != NULL);
> +#endif
> +
>       down_write(&slub_lock);
> -      s = find_mergeable(size, align, flags, name, ctor);
> +      s = find_mergeable(memcg, size, align, flags, name, ctor);
>       if (s) {
>               s->refcount++;
>               /*
> @@ -3954,12 +3963,15 @@ struct kmem_cache *kmem_cache_create(const char *name,
size_t size,
>                               size, align, flags, ctor)) {
>                       list_add(&s->list, &slab_caches);
>                       up_write(&slub_lock);
> +                     mem_cgroup_register_cache(memcg, s);
```

Do the kmalloc caches get their id registered correctly?

-- Suleiman

Subject: Re: [PATCH 12/23] slab: pass memcg parameter to kmem_cache_create
Posted by Suleiman Souhlal on Mon, 30 Apr 2012 19:54:36 GMT
View Forum Message <> Reply to Message

On Sun, Apr 22, 2012 at 4:53 PM, Glauber Costa <glommer@parallels.com> wrote:
> Allow a memcg parameter to be passed during cache creation.
>
> Default function is created as a wrapper, passing NULL
> to the memcg version. We only merge caches that belong
> to the same memcg.
>
> This code was mostly written by Suleiman Souhlal and
> only adapted to my patchset, plus a couple of simplifications
>
> 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>
> CC: Suleiman Souhlal <suleiman@google.com>

```
> ---
>  mm/slab.c |   38 ++++++++++++++++++++++++++++++++++++++---------
>  1 files changed, 29 insertions(+), 9 deletions(-)
>
> diff --git a/mm/slab.c b/mm/slab.c
> index a0d51dd..362bb6e 100644
> --- a/mm/slab.c
> +++ b/mm/slab.c
> @@ -2287,14 +2287,15 @@ static int __init_refok setup_cpu_cache(struct kmem_cache
*cachep, gfp_t gfp)
>  * cacheline.  This can be beneficial if you're counting cycles as closely
>  * as davem.
>  */
> -struct kmem_cache *
> -kmem_cache_create (const char *name, size_t size, size_t align,
> -       unsigned long flags, void (*ctor)(void *))
> +static struct kmem_cache *
> +__kmem_cache_create(struct mem_cgroup *memcg, const char *name, size_t size,
> +                   size_t align, unsigned long flags, void (*ctor)(void *))
>  {
> -       size_t left_over, slab_size, ralign;
> +       size_t left_over, orig_align, ralign, slab_size;
>        struct kmem_cache *cachep = NULL, *pc;
>        gfp_t gfp;
>
> +       orig_align = align;
>        /*
>         * Sanity checks... these are all serious usage bugs.
>         */
> @@ -2311,7 +2312,6 @@ kmem_cache_create (const char *name, size_t size, size_t align,
>         */
>        if (slab_is_available()) {
>                get_online_cpus();
> -               mutex_lock(&cache_chain_mutex);
>        }
>
>        list_for_each_entry(pc, &cache_chain, next) {
> @@ -2331,9 +2331,9 @@ kmem_cache_create (const char *name, size_t size, size_t align,
>                        continue;
>                }
>
> -               if (!strcmp(pc->name, name)) {
> +               if (!strcmp(pc->name, name) && !memcg) {
>                        printk(KERN_ERR
> -                              "kmem_cache_create: duplicate cache %s\n", name);
> +                                "kmem_cache_create: duplicate cache %s\n", name);
>                        dump_stack();
>                        goto oops;
```

```
>             }
> @@ -2434,6 +2434,9 @@ kmem_cache_create (const char *name, size_t size, size_t align,
>       cachep->nodelists = (struct kmem_list3 **)&cachep->array[nr_cpu_ids];
>
>       set_obj_size(cachep, size);
> +#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
> +       cachep->memcg_params.orig_align = orig_align;
> +#endif
>  #if DEBUG
>
>       /*
> @@ -2541,7 +2544,12 @@ kmem_cache_create (const char *name, size_t size, size_t align,
>               BUG_ON(ZERO_OR_NULL_PTR(cachep->slabp_cache));
>       }
>       cachep->ctor = ctor;
> -       cachep->name = name;
> +       cachep->name = (char *)name;
> +
> +#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
> +       mem_cgroup_register_cache(memcg, cachep);
> +       atomic_set(&cachep->memcg_params.refcnt, 1);
> +#endif
```

cache_cache probably doesn't get its id registered correctly. :-(
We might need to add a mem_cgroup_register_cache() call to kmem_cache_init().

-- Suleiman

Subject: Re: [PATCH 13/23] slub: create duplicate cache
Posted by Suleiman Souhlal on Mon, 30 Apr 2012 20:15:23 GMT
View Forum Message <> Reply to Message

On Sun, Apr 22, 2012 at 4:53 PM, Glauber Costa <glommer@parallels.com> wrote:
> This patch provides kmem_cache_dup(), that duplicates
> a cache for a memcg, preserving its creation properties.
> Object size, alignment and flags are all respected.
>
> When a duplicate cache is created, the parent cache cannot
> be destructed during the child lifetime. To assure this,
> its reference count is increased if the cache creation
> succeeds.
>
> 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>
> CC: Suleiman Souhlal <suleiman@google.com>
> ---
> include/linux/memcontrol.h | 3 +++
> include/linux/slab.h | 3 +++
> mm/memcontrol.c | 44
++
> mm/slub.c | 37 ++++++++++++++++++++++++++++++++++++
> 4 files changed, 87 insertions(+), 0 deletions(-)
>
> diff --git a/include/linux/memcontrol.h b/include/linux/memcontrol.h
> index 99e14b9..493ecdd 100644
> --- a/include/linux/memcontrol.h
> +++ b/include/linux/memcontrol.h
> @@ -445,6 +445,9 @@ int memcg_css_id(struct mem_cgroup *memcg);
> void mem_cgroup_register_cache(struct mem_cgroup *memcg,
>   struct kmem_cache *s);
> void mem_cgroup_release_cache(struct kmem_cache *cachep);
> +extern char *mem_cgroup_cache_name(struct mem_cgroup *memcg,
> +   struct kmem_cache *cachep);
> +
> #else
> static inline void mem_cgroup_register_cache(struct mem_cgroup *memcg,
>   struct kmem_cache *s)
> diff --git a/include/linux/slab.h b/include/linux/slab.h
> index c7a7e05..909b508 100644
> --- a/include/linux/slab.h
> +++ b/include/linux/slab.h
> @@ -323,6 +323,9 @@ extern void *__kmalloc_track_caller(size_t, gfp_t, unsigned long);
>
> #ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
> #define MAX_KMEM_CACHE_TYPES 400
> +extern struct kmem_cache *kmem_cache_dup(struct mem_cgroup *memcg,
> +   struct kmem_cache *cachep);
> +void kmem_cache_drop_ref(struct kmem_cache *cachep);
> #else
> #define MAX_KMEM_CACHE_TYPES 0
> #endif /* CONFIG_CGROUP_MEM_RES_CTLR_KMEM */
> diff --git a/mm/memcontrol.c b/mm/memcontrol.c
> index 0015ed0..e881d83 100644
> --- a/mm/memcontrol.c
> +++ b/mm/memcontrol.c
> @@ -467,6 +467,50 @@ struct cg_proto *tcp_proto_cgroup(struct mem_cgroup *memcg)
> EXPORT_SYMBOL(tcp_proto_cgroup);
> #endif /* CONFIG_INET */
>
> +/*
> + * This is to prevent races againt the kmalloc cache creations.

```
> + * Should never be used outside the core memcg code. Therefore,
> + * copy it here, instead of letting it in lib/
> + */
> +static char *kasprintf_no_account(gfp_t gfp, const char *fmt, ...)
> +{
> +       unsigned int len;
> +       char *p = NULL;
> +       va_list ap, aq;
> +
> +       va_start(ap, fmt);
> +       va_copy(aq, ap);
> +       len = vsnprintf(NULL, 0, fmt, aq);
> +       va_end(aq);
> +
> +       p = kmalloc_no_account(len+1, gfp);
> +       if (!p)
> +               goto out;
> +
> +       vsnprintf(p, len+1, fmt, ap);
> +
> +out:
> +       va_end(ap);
> +       return p;
> +}
> +
> +char *mem_cgroup_cache_name(struct mem_cgroup *memcg, struct kmem_cache *cachep)
> +{
> +       char *name;
> +       struct dentry *dentry = memcg->css.cgroup->dentry;
```

Do we need rcu_dereference() here (and make sure we have rcu_read_lock())?

This might need to be done in all the other places in the patchset
that get the memcg's dentry.

-- Suleiman

Subject: Re: [PATCH 17/23] kmem controller charge/uncharge infrastructure
Posted by Suleiman Souhlal on Mon, 30 Apr 2012 20:56:46 GMT
View Forum Message <> Reply to Message

On Sun, Apr 22, 2012 at 4:53 PM, Glauber Costa <glommer@parallels.com> wrote:
> With all the dependencies already in place, this patch introduces
> the charge/uncharge functions for the slab cache accounting in memcg.
>
> Before we can charge a cache, we need to select the right cache.
> This is done by using the function __mem_cgroup_get_kmem_cache().

>
> If we should use the root kmem cache, this function tries to detect
> that and return as early as possible.
>
> The charge and uncharge functions comes in two flavours:
> * __mem_cgroup_(un)charge_slab(), that assumes the allocation is
>   a slab page, and
> * __mem_cgroup_(un)charge_kmem(), that does not. This later exists
>   because the slub allocator draws the larger kmalloc allocations
>   from the page allocator.
>
> In memcontrol.h those functions are wrapped in inline acessors.
> The idea is to later on, patch those with jump labels, so we don't
> incur any overhead when no mem cgroups are being used.
>
> Because the slub allocator tends to inline the allocations whenever
> it can, those functions need to be exported so modules can make use
> of it properly.
>
> I apologize in advance to the reviewers. This patch is quite big, but
> I was not able to split it any further due to all the dependencies
> between the code.
>
> This code is inspired by the code written by Suleiman Souhlal,
> but heavily changed.
>
> 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>
> CC: Suleiman Souhlal <suleiman@google.com>
> ---
> include/linux/memcontrol.h | 68 ++++++++
> init/Kconfig | 2 +-
> mm/memcontrol.c | 373
++-
> 3 files changed, 441 insertions(+), 2 deletions(-)
>

> +
> +static struct kmem_cache *memcg_create_kmem_cache(struct mem_cgroup *memcg,
> +   struct kmem_cache
*cachep)
> +{
> +   struct kmem_cache *new_cachep;
> +   int idx;

```
> +
> +       BUG_ON(!mem_cgroup_kmem_enabled(memcg));
> +
> +       idx = cachep->memcg_params.id;
> +
> +       mutex_lock(&memcg_cache_mutex);
> +       new_cachep = memcg->slabs[idx];
> +       if (new_cachep)
> +               goto out;
> +
> +       new_cachep = kmem_cache_dup(memcg, cachep);
> +
> +       if (new_cachep == NULL) {
> +               new_cachep = cachep;
> +               goto out;
> +       }
> +
> +       mem_cgroup_get(memcg);
> +       memcg->slabs[idx] = new_cachep;
> +       new_cachep->memcg_params.memcg = memcg;
> +out:
> +       mutex_unlock(&memcg_cache_mutex);
> +       return new_cachep;
> +}
> +
> +struct create_work {
> +       struct mem_cgroup *memcg;
> +       struct kmem_cache *cachep;
> +       struct list_head list;
> +};
> +
> +/* Use a single spinlock for destruction and creation, not a frequent op */
> +static DEFINE_SPINLOCK(cache_queue_lock);
> +static LIST_HEAD(create_queue);
> +static LIST_HEAD(destroyed_caches);
> +
> +static void kmem_cache_destroy_work_func(struct work_struct *w)
> +{
> +       struct kmem_cache *cachep;
> +       char *name;
> +
> +       spin_lock_irq(&cache_queue_lock);
> +       while (!list_empty(&destroyed_caches)) {
> +               cachep = container_of(list_first_entry(&destroyed_caches,
> +                   struct mem_cgroup_cache_params, destroyed_list), struct
> +                   kmem_cache, memcg_params);
> +               name = (char *)cachep->name;
> +               list_del(&cachep->memcg_params.destroyed_list);
```

```
> +               spin_unlock_irq(&cache_queue_lock);
> +               synchronize_rcu();
```

Is this synchronize_rcu() still needed, now that we don't use RCU to
protect memcgs from disappearing during allocation anymore?

Also, should we drop the memcg reference we got in
memcg_create_kmem_cache() here?

```
> +               kmem_cache_destroy(cachep);
> +               kfree(name);
> +               spin_lock_irq(&cache_queue_lock);
> +       }
> +       spin_unlock_irq(&cache_queue_lock);
> +}
> +static DECLARE_WORK(kmem_cache_destroy_work, kmem_cache_destroy_work_func);
> +
> +void mem_cgroup_destroy_cache(struct kmem_cache *cachep)
> +{
> +       unsigned long flags;
> +
> +       BUG_ON(cachep->memcg_params.id != -1);
> +
> +       /*
> +        * We have to defer the actual destroying to a workqueue, because
> +        * we might currently be in a context that cannot sleep.
> +        */
> +       spin_lock_irqsave(&cache_queue_lock, flags);
> +       list_add(&cachep->memcg_params.destroyed_list, &destroyed_caches);
> +       spin_unlock_irqrestore(&cache_queue_lock, flags);
> +
> +       schedule_work(&kmem_cache_destroy_work);
> +}
> +
> +
> +/*
> + * Flush the queue of kmem_caches to create, because we're creating a cgroup.
> + *
> + * We might end up flushing other cgroups' creation requests as well, but
> + * they will just get queued again next time someone tries to make a slab
> + * allocation for them.
> + */
> +void mem_cgroup_flush_cache_create_queue(void)
> +{
> +       struct create_work *cw, *tmp;
> +       unsigned long flags;
> +
> +       spin_lock_irqsave(&cache_queue_lock, flags);
```

```
> +	list_for_each_entry_safe(cw, tmp, &create_queue, list) {
> +		list_del(&cw->list);
> +		kfree(cw);
> +	}
> +	spin_unlock_irqrestore(&cache_queue_lock, flags);
> +}
> +
> +static void memcg_create_cache_work_func(struct work_struct *w)
> +{
> +	struct kmem_cache *cachep;
> +	struct create_work *cw;
> +
> +	spin_lock_irq(&cache_queue_lock);
> +	while (!list_empty(&create_queue)) {
> +		cw = list_first_entry(&create_queue, struct create_work, list);
> +		list_del(&cw->list);
> +		spin_unlock_irq(&cache_queue_lock);
> +		cachep = memcg_create_kmem_cache(cw->memcg, cw->cachep);
> +		if (cachep == NULL)
> +			printk(KERN_ALERT
> +			"%s: Couldn't create memcg-cache for %s memcg %s\n",
> +			__func__, cw->cachep->name,
> +			cw->memcg->css.cgroup->dentry->d_name.name);
```

We might need rcu_dereference() here (and hold rcu_read_lock()).
Or we could just remove this message.

```
> +		/* Drop the reference gotten when we enqueued. */
> +		css_put(&cw->memcg->css);
> +		kfree(cw);
> +		spin_lock_irq(&cache_queue_lock);
> +	}
> +	spin_unlock_irq(&cache_queue_lock);
> +}
> +
> +static DECLARE_WORK(memcg_create_cache_work, memcg_create_cache_work_func);
> +
> +/*
> + * Enqueue the creation of a per-memcg kmem_cache.
> + * Called with rcu_read_lock.
> + */
> +static void memcg_create_cache_enqueue(struct mem_cgroup *memcg,
> +				       struct kmem_cache *cachep)
> +{
> +	struct create_work *cw;
> +	unsigned long flags;
> +
> +	spin_lock_irqsave(&cache_queue_lock, flags);
```

```
> +	list_for_each_entry(cw, &create_queue, list) {
> +		if (cw->memcg == memcg && cw->cachep == cachep) {
> +			spin_unlock_irqrestore(&cache_queue_lock, flags);
> +			return;
> +		}
> +	}
> +	spin_unlock_irqrestore(&cache_queue_lock, flags);
> +
> +	/* The corresponding put will be done in the workqueue. */
> +	if (!css_tryget(&memcg->css))
> +		return;
> +
> +	cw = kmalloc_no_account(sizeof(struct create_work), GFP_NOWAIT);
> +	if (cw == NULL) {
> +		css_put(&memcg->css);
> +		return;
> +	}
> +
> +	cw->memcg = memcg;
> +	cw->cachep = cachep;
> +	spin_lock_irqsave(&cache_queue_lock, flags);
> +	list_add_tail(&cw->list, &create_queue);
> +	spin_unlock_irqrestore(&cache_queue_lock, flags);
> +
> +	schedule_work(&memcg_create_cache_work);
> +}
> +
> +/*
> + * Return the kmem_cache we're supposed to use for a slab allocation.
> + * If we are in interrupt context or otherwise have an allocation that
> + * can't fail, we return the original cache.
> + * Otherwise, we will try to use the current memcg's version of the cache.
> + *
> + * If the cache does not exist yet, if we are the first user of it,
> + * we either create it immediately, if possible, or create it asynchronously
> + * in a workqueue.
> + * In the latter case, we will let the current allocation go through with
> + * the original cache.
> + *
> + * This function returns with rcu_read_lock() held.
> + */
> +struct kmem_cache *__mem_cgroup_get_kmem_cache(struct kmem_cache *cachep,
> +					       gfp_t gfp)
> +{
> +	struct mem_cgroup *memcg;
> +	int idx;
> +
> +	gfp |=  cachep->allocflags;
```

```
> +
> +       if ((current->mm == NULL))
> +               return cachep;
> +
> +       if (cachep->memcg_params.memcg)
> +               return cachep;
> +
> +       idx = cachep->memcg_params.id;
> +       VM_BUG_ON(idx == -1);
> +
> +       memcg = mem_cgroup_from_task(current);
> +       if (!mem_cgroup_kmem_enabled(memcg))
> +               return cachep;
> +
> +       if (rcu_access_pointer(memcg->slabs[idx]) == NULL) {
> +               memcg_create_cache_enqueue(memcg, cachep);
> +               return cachep;
> +       }
> +
> +       return rcu_dereference(memcg->slabs[idx]);
```

Is it ok to call rcu_access_pointer() and rcu_dereference() without
holding rcu_read_lock()?

```
> +}
> +EXPORT_SYMBOL(__mem_cgroup_get_kmem_cache);
> +
> +void mem_cgroup_remove_child_kmem_cache(struct kmem_cache *cachep, int id)
> +{
> +       rcu_assign_pointer(cachep->memcg_params.memcg->slabs[id], NULL);
> +}
> +
> +bool __mem_cgroup_charge_kmem(gfp_t gfp, size_t size)
> +{
> +       struct mem_cgroup *memcg;
> +       bool ret = true;
> +
> +       rcu_read_lock();
> +       memcg = mem_cgroup_from_task(current);
> +
> +       if (!mem_cgroup_kmem_enabled(memcg))
> +               goto out;
> +
> +       mem_cgroup_get(memcg);
```

Why do we need to get a reference to the memcg for every charge?
How will this work when deleting a memcg?

```
> +	ret = memcg_charge_kmem(memcg, gfp, size) == 0;
> +	if (ret)
> +		mem_cgroup_put(memcg);
> +out:
> +	rcu_read_unlock();
> +	return ret;
> +}
> +EXPORT_SYMBOL(__mem_cgroup_charge_kmem);
> +
> +void __mem_cgroup_uncharge_kmem(size_t size)
> +{
> +	struct mem_cgroup *memcg;
> +
> +	rcu_read_lock();
> +	memcg = mem_cgroup_from_task(current);
> +
> +	if (!mem_cgroup_kmem_enabled(memcg))
> +		goto out;
> +
> +	mem_cgroup_put(memcg);
> +	memcg_uncharge_kmem(memcg, size);
> +out:
> +	rcu_read_unlock();
> +}
> +EXPORT_SYMBOL(__mem_cgroup_uncharge_kmem);
> +
> +bool __mem_cgroup_charge_slab(struct kmem_cache *cachep, gfp_t gfp, size_t size)
> +{
> +	struct mem_cgroup *memcg;
> +	bool ret = true;
> +
> +	rcu_read_lock();
> +	memcg = cachep->memcg_params.memcg;
> +	if (!mem_cgroup_kmem_enabled(memcg))
> +		goto out;
> +
> +	ret = memcg_charge_kmem(memcg, gfp, size) == 0;
> +out:
> +	rcu_read_unlock();
> +	return ret;
> +}
> +EXPORT_SYMBOL(__mem_cgroup_charge_slab);
> +
> +void __mem_cgroup_uncharge_slab(struct kmem_cache *cachep, size_t size)
> +{
> +	struct mem_cgroup *memcg;
> +
> +	rcu_read_lock();
```

```
> +       memcg = cachep->memcg_params.memcg;
> +
> +       if (!mem_cgroup_kmem_enabled(memcg)) {
> +               rcu_read_unlock();
> +               return;
> +       }
> +       rcu_read_unlock();
> +
> +       memcg_uncharge_kmem(memcg, size);
> +}
> +EXPORT_SYMBOL(__mem_cgroup_uncharge_slab);
> +
> +static void memcg_slab_init(struct mem_cgroup *memcg)
> +{
> +       int i;
> +
> +       for (i = 0; i < MAX_KMEM_CACHE_TYPES; i++)
> +               rcu_assign_pointer(memcg->slabs[i], NULL);
> +}
>  #endif /* CONFIG_CGROUP_MEM_RES_CTLR_KMEM */
>
>  static void drain_all_stock_async(struct mem_cgroup *memcg);
> @@ -4790,7 +5103,11 @@ static struct cftype kmem_cgroup_files[] = {
>
>  static int memcg_init_kmem(struct mem_cgroup *memcg, struct cgroup_subsys *ss)
>  {
> -       return mem_cgroup_sockets_init(memcg, ss);
> +       int ret = mem_cgroup_sockets_init(memcg, ss);
> +
> +       if (!ret)
> +               memcg_slab_init(memcg);
> +       return ret;
>  };
>
>  static void kmem_cgroup_destroy(struct mem_cgroup *memcg)
> @@ -5805,3 +6122,57 @@ static int __init enable_swap_account(char *s)
>  __setup("swapaccount=", enable_swap_account);
>
>  #endif
> +
> +#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
> +int memcg_charge_kmem(struct mem_cgroup *memcg, gfp_t gfp, s64 delta)
> +{
> +       struct res_counter *fail_res;
> +       struct mem_cgroup *_memcg;
> +       int may_oom, ret;
> +       bool nofail = false;
> +
```

```
> +       may_oom = (gfp & __GFP_WAIT) && (gfp & __GFP_FS) &&
> +           !(gfp & __GFP_NORETRY);
> +
> +       ret = 0;
> +
> +       if (!memcg)
> +               return ret;
> +
> +       _memcg = memcg;
> +       ret = __mem_cgroup_try_charge(NULL, gfp, delta / PAGE_SIZE,
> +           &_memcg, may_oom);
> +       if (ret == -ENOMEM)
> +               return ret;
> +       else if ((ret == -EINTR) || (ret && (gfp & __GFP_NOFAIL)))  {
> +               nofail = true;
> +               /*
> +                * __mem_cgroup_try_charge() chose 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, force the
> +                * change, going above the limit if needed.
> +                */
> +               res_counter_charge_nofail(&memcg->res, delta, &fail_res);
```

We might need to charge memsw here too.

```
> +       }
> +
> +       if (nofail)
> +               res_counter_charge_nofail(&memcg->kmem, delta, &fail_res);
> +       else
> +               ret = res_counter_charge(&memcg->kmem, delta, &fail_res);
> +
> +       if (ret)
> +               res_counter_uncharge(&memcg->res, delta);
> +
> +       return ret;
> +}
> +
> +void memcg_uncharge_kmem(struct mem_cgroup *memcg, s64 delta)
> +{
> +       if (!memcg)
> +               return;
> +
> +       res_counter_uncharge(&memcg->kmem, delta);
> +       res_counter_uncharge(&memcg->res, delta);
```

Might need to uncharge memsw.

> +}
> +#endif /* CONFIG_CGROUP_MEM_RES_CTLR_KMEM */
> --
> 1.7.7.6
>

Subject: Re: [PATCH 19/23] slab: per-memcg accounting of slab caches
Posted by Suleiman Souhlal on Mon, 30 Apr 2012 21:25:47 GMT
View Forum Message <> Reply to Message

On Sun, Apr 22, 2012 at 4:53 PM, Glauber Costa <glommer@parallels.com> wrote:
> This patch charges allocation of a slab object to a particular
> memcg.
>
> The cache is selected with mem_cgroup_get_kmem_cache(),
> which is the biggest overhead we pay here, because
> it happens at all allocations. However, other than forcing
> a function call, this function is not very expensive, and
> try to return as soon as we realize we are not a memcg cache.
>
> The charge/uncharge functions are heavier, but are only called
> for new page allocations.
>
> Code is heavily inspired by Suleiman's, with adaptations to
> the patchset and minor simplifications by me.
>
> 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>
> CC: Suleiman Souhlal <suleiman@google.com>
> ---
> include/linux/slab_def.h | 66 ++++++++++++++++++++++++++++-
> mm/slab.c | 105
++----
> 2 files changed, 162 insertions(+), 9 deletions(-)
>
> diff --git a/include/linux/slab_def.h b/include/linux/slab_def.h
> index 54d25d7..c4f7e45 100644
> --- a/include/linux/slab_def.h
> +++ b/include/linux/slab_def.h
> @@ -51,7 +51,7 @@ struct kmem_cache {
> void (*ctor)(void *obj);
>

```
>  /* 4) cache creation/removal */
> -       const char *name;
> +       char *name;
>         struct list_head next;
>
>  /* 5) statistics */
> @@ -219,4 +219,68 @@ found:
>
>  #endif /* CONFIG_NUMA */
>
> +#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
> +
> +void kmem_cache_drop_ref(struct kmem_cache *cachep);
> +
> +static inline void
> +kmem_cache_get_ref(struct kmem_cache *cachep)
> +{
> +       if (cachep->memcg_params.id == -1 &&
> +           unlikely(!atomic_add_unless(&cachep->memcg_params.refcnt, 1, 0)))
> +               BUG();
> +}
> +
> +static inline void
> +mem_cgroup_put_kmem_cache(struct kmem_cache *cachep)
> +{
> +       rcu_read_unlock();
> +}
> +
> +static inline void
> +mem_cgroup_kmem_cache_prepare_sleep(struct kmem_cache *cachep)
> +{
> +       /*
> +        * Make sure the cache doesn't get freed while we have interrupts
> +        * enabled.
> +        */
> +       kmem_cache_get_ref(cachep);
> +       rcu_read_unlock();
> +}
> +
> +static inline void
> +mem_cgroup_kmem_cache_finish_sleep(struct kmem_cache *cachep)
> +{
> +       rcu_read_lock();
> +       kmem_cache_drop_ref(cachep);
> +}
> +
> +#else /* CONFIG_CGROUP_MEM_RES_CTLR_KMEM */
> +
```

```
> +static inline void
> +kmem_cache_get_ref(struct kmem_cache *cachep)
> +{
> +}
> +
> +static inline void
> +kmem_cache_drop_ref(struct kmem_cache *cachep)
> +{
> +}
> +
> +static inline void
> +mem_cgroup_put_kmem_cache(struct kmem_cache *cachep)
> +{
> +}
> +
> +static inline void
> +mem_cgroup_kmem_cache_prepare_sleep(struct kmem_cache *cachep)
> +{
> +}
> +
> +static inline void
> +mem_cgroup_kmem_cache_finish_sleep(struct kmem_cache *cachep)
> +{
> +}
> +#endif /* CONFIG_CGROUP_MEM_RES_CTLR_KMEM */
> +
>  #endif /* _LINUX_SLAB_DEF_H */
> diff --git a/mm/slab.c b/mm/slab.c
> index 13948c3..ac0916b 100644
> --- a/mm/slab.c
> +++ b/mm/slab.c
> @@ -1818,20 +1818,28 @@ static void *kmem_getpages(struct kmem_cache *cachep, gfp_t
flags, int nodeid)
>       if (cachep->flags & SLAB_RECLAIM_ACCOUNT)
>               flags |= __GFP_RECLAIMABLE;
>
> +     nr_pages = (1 << cachep->gfporder);
> +     if (!mem_cgroup_charge_slab(cachep, flags, nr_pages * PAGE_SIZE))
> +             return NULL;
> +
>       page = alloc_pages_exact_node(nodeid, flags | __GFP_NOTRACK,
cachep->gfporder);
>       if (!page) {
>               if (!(flags & __GFP_NOWARN) && printk_ratelimit())
>                       slab_out_of_memory(cachep, flags, nodeid);
> +
> +             mem_cgroup_uncharge_slab(cachep, nr_pages * PAGE_SIZE);
>               return NULL;
```

```
>       }
>
> -      nr_pages = (1 << cachep->gfporder);
>       if (cachep->flags & SLAB_RECLAIM_ACCOUNT)
>             add_zone_page_state(page_zone(page),
>                   NR_SLAB_RECLAIMABLE, nr_pages);
>       else
>             add_zone_page_state(page_zone(page),
>                   NR_SLAB_UNRECLAIMABLE, nr_pages);
> +
> +       kmem_cache_get_ref(cachep);
> +
>       for (i = 0; i < nr_pages; i++)
>             __SetPageSlab(page + i);
>
> @@ -1864,6 +1872,8 @@ static void kmem_freepages(struct kmem_cache *cachep, void
*addr)
>       else
>             sub_zone_page_state(page_zone(page),
>                         NR_SLAB_UNRECLAIMABLE, nr_freed);
> +       mem_cgroup_uncharge_slab(cachep, i * PAGE_SIZE);
> +       kmem_cache_drop_ref(cachep);
>       while (i--) {
>             BUG_ON(!PageSlab(page));
>             __ClearPageSlab(page);
> @@ -2823,12 +2833,28 @@ void kmem_cache_destroy(struct kmem_cache *cachep)
>       if (unlikely(cachep->flags & SLAB_DESTROY_BY_RCU))
>             rcu_barrier();
>
> +#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
> +       /* Not a memcg cache */
> +       if (cachep->memcg_params.id != -1) {
> +             mem_cgroup_release_cache(cachep);
> +             mem_cgroup_flush_cache_create_queue();
> +       }
> +#endif
>       __kmem_cache_destroy(cachep);
>       mutex_unlock(&cache_chain_mutex);
>       put_online_cpus();
>  }
>  EXPORT_SYMBOL(kmem_cache_destroy);
>
> +#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
> +void kmem_cache_drop_ref(struct kmem_cache *cachep)
> +{
> +       if (cachep->memcg_params.id == -1 &&
> +           unlikely(atomic_dec_and_test(&cachep->memcg_params.refcnt)))
> +             mem_cgroup_destroy_cache(cachep);
```

```
> +}
> +#endif /* CONFIG_CGROUP_MEM_RES_CTLR_KMEM */
> +
>  /*
>   * Get the memory for a slab management obj.
>   * For a slab cache when the slab descriptor is off-slab, slab descriptors
> @@ -3028,8 +3054,10 @@ static int cache_grow(struct kmem_cache *cachep,
>
>       offset *= cachep->colour_off;
>
> -     if (local_flags & __GFP_WAIT)
> +     if (local_flags & __GFP_WAIT) {
>               local_irq_enable();
> +             mem_cgroup_kmem_cache_prepare_sleep(cachep);
> +     }
>
>       /*
>        * The test for missing atomic flag is performed here, rather than
> @@ -3058,8 +3086,10 @@ static int cache_grow(struct kmem_cache *cachep,
>
>       cache_init_objs(cachep, slabp);
>
> -     if (local_flags & __GFP_WAIT)
> +     if (local_flags & __GFP_WAIT) {
>               local_irq_disable();
> +             mem_cgroup_kmem_cache_finish_sleep(cachep);
> +     }
>       check_irq_off();
>       spin_lock(&l3->list_lock);
>
> @@ -3072,8 +3102,10 @@ static int cache_grow(struct kmem_cache *cachep,
>  opps1:
>       kmem_freepages(cachep, objp);
>  failed:
> -     if (local_flags & __GFP_WAIT)
> +     if (local_flags & __GFP_WAIT) {
>               local_irq_disable();
> +             mem_cgroup_kmem_cache_finish_sleep(cachep);
> +     }
>       return 0;
>  }
>
> @@ -3834,11 +3866,15 @@ static inline void __cache_free(struct kmem_cache *cachep, void
*objp,
>  */
>  void *kmem_cache_alloc(struct kmem_cache *cachep, gfp_t flags)
>  {
> -     void *ret = __cache_alloc(cachep, flags, __builtin_return_address(0));
```

```
> +       void *ret;
> +
> +       rcu_read_lock();
> +       cachep = mem_cgroup_get_kmem_cache(cachep, flags);
> +       rcu_read_unlock();
```

Don't we need to check in_interrupt(), current, __GFP_NOFAIL every time we call mem_cgroup_cgroup_get_kmem_cache()?

I would personally prefer if those checks were put inside mem_cgroup_get_kmem_cache() instead of having to check for every caller.

-- Suleiman

Subject: Re: [PATCH 09/23] kmem slab accounting basic infrastructure
Posted by Glauber Costa on Wed, 02 May 2012 15:15:43 GMT
View Forum Message <> Reply to Message

```
>> @@ -3951,8 +3966,26 @@ 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);
>> +#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
>> +               else if (type == _KMEM) {
>> +                       ret = res_counter_set_limit(&memcg->kmem, val);
>> +                       if (ret)
>> +                               break;
>> +                       /*
>> +                        * Once enabled, can't be disabled. We could in theory
>> +                        * disable it if we haven't yet created any caches, or
>> +                        * if we can shrink them all to death.
>> +                        *
>> +                        * But it is not worth the trouble
>> +                        */
>> +                       if (!memcg->kmem_accounted&&  val != RESOURCE_MAX)
>> +                               memcg->kmem_accounted = true;
>> +               }
>> +#endif
>> +               else
>> +                       return -EINVAL;
>>                 break;
>>         case RES_SOFT_LIMIT:
>>                 ret = res_counter_memparse_write_strategy(buffer,&val);
```

>
> Why is RESOURCE_MAX special?

Because I am using the convention that setting it to any value different
than that will enable accounting.

Subject: Re: [PATCH 11/23] slub: consider a memcg parameter in kmem_create_cache
Posted by Glauber Costa on Wed, 02 May 2012 15:18:04 GMT
View Forum Message <> Reply to Message

On 04/30/2012 04:51 PM, Suleiman Souhlal wrote:
> On Fri, Apr 20, 2012 at 2:57 PM, Glauber Costa<glommer@parallels.com> wrote:
>> Allow a memcg parameter to be passed during cache creation.
>> The slub allocator will only merge caches that belong to
>> the same memcg.
>>
>> Default function is created as a wrapper, passing NULL
>> to the memcg version. We only merge caches that belong
>> to the same memcg.
>>
>> From the memcontrol.c side, 3 helper functions are created:
>>
>> 1) memcg_css_id: because slub needs a unique cache name
>> for sysfs. Since this is visible, but not the canonical
>> location for slab data, the cache name is not used, the
>> css_id should suffice.
>>
>> 2) mem_cgroup_register_cache: is responsible for assigning
>> a unique index to each cache, and other general purpose
>> setup. The index is only assigned for the root caches. All
>> others are assigned index == -1.
>>
>> 3) mem_cgroup_release_cache: can be called from the root cache
>> destruction, and will release the index for other caches.
>>
>> This index mechanism was developed by Suleiman Souhlal.
>>
>> 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>
>> CC: Suleiman Souhlal<suleiman@google.com>
>> ---
>> include/linux/memcontrol.h | 14 ++++++++++++++

>> include/linux/slab.h | 6 ++++++
>> mm/memcontrol.c | 29 ++++++++++++++++++++++++++++
>> mm/slub.c | 31 ++++++++++++++++++++++++++++++----
>> 4 files changed, 76 insertions(+), 4 deletions(-)
>>
>> diff --git a/include/linux/memcontrol.h b/include/linux/memcontrol.h
>> index f94efd2..99e14b9 100644
>> --- a/include/linux/memcontrol.h
>> +++ b/include/linux/memcontrol.h
>> @@ -26,6 +26,7 @@ struct mem_cgroup;
>> struct page_cgroup;
>> struct page;
>> struct mm_struct;
>> +struct kmem_cache;
>>
>> /* Stats that can be updated by kernel. */
>> enum mem_cgroup_page_stat_item {
>> @@ -440,7 +441,20 @@ struct sock;
>> #ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
>> void sock_update_memcg(struct sock *sk);
>> void sock_release_memcg(struct sock *sk);
>> +int memcg_css_id(struct mem_cgroup *memcg);
>> +void mem_cgroup_register_cache(struct mem_cgroup *memcg,
>> +   struct kmem_cache *s);
>> +void mem_cgroup_release_cache(struct kmem_cache *cachep);
>> #else
>> +static inline void mem_cgroup_register_cache(struct mem_cgroup *memcg,
>> +   struct kmem_cache *s)
>> +{
>> +}
>> +
>> +static inline void mem_cgroup_release_cache(struct kmem_cache *cachep)
>> +{
>> +}
>> +
>> static inline void sock_update_memcg(struct sock *sk)
>> {
>> }
>> diff --git a/include/linux/slab.h b/include/linux/slab.h
>> index a5127e1..c7a7e05 100644
>> --- a/include/linux/slab.h
>> +++ b/include/linux/slab.h
>> @@ -321,6 +321,12 @@ extern void *__kmalloc_track_caller(size_t, gfp_t, unsigned long);
>>   __kmalloc(size, flags)
>> #endif /* DEBUG_SLAB */
>>
>> +#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
>> +#define MAX_KMEM_CACHE_TYPES 400

```
>> +#else
>> +#define MAX_KMEM_CACHE_TYPES 0
>> +#endif /* CONFIG_CGROUP_MEM_RES_CTLR_KMEM */
>> +
>>  #ifdef CONFIG_NUMA
>>  /*
>>  * kmalloc_node_track_caller is a special version of kmalloc_node that
>> diff --git a/mm/memcontrol.c b/mm/memcontrol.c
>> index 36f1e6b..0015ed0 100644
>> --- a/mm/memcontrol.c
>> +++ b/mm/memcontrol.c
>> @@ -323,6 +323,11 @@ struct mem_cgroup {
>>  #endif
>>  };
>>
>> +int memcg_css_id(struct mem_cgroup *memcg)
>> +{
>> +       return css_id(&memcg->css);
>> +}
>> +
>>  /* Stuffs for move charges at task migration. */
>>  /*
>>  * Types of charges to be moved. "move_charge_at_immitgrate" is treated as a
>> @@ -461,6 +466,30 @@ struct cg_proto *tcp_proto_cgroup(struct mem_cgroup *memcg)
>>  }
>>  EXPORT_SYMBOL(tcp_proto_cgroup);
>>  #endif /* CONFIG_INET */
>> +
>> +/* Bitmap used for allocating the cache id numbers. */
>> +static DECLARE_BITMAP(cache_types, MAX_KMEM_CACHE_TYPES);
>> +
>> +void mem_cgroup_register_cache(struct mem_cgroup *memcg,
>> +                               struct kmem_cache *cachep)
>> +{
>> +       int id = -1;
>> +
>> +       cachep->memcg_params.memcg = memcg;
>> +
>> +       if (!memcg) {
>> +               id = find_first_zero_bit(cache_types, MAX_KMEM_CACHE_TYPES);
>> +               BUG_ON(id<  0 || id>= MAX_KMEM_CACHE_TYPES);
>> +               __set_bit(id, cache_types);
>> +       } else
>> +               INIT_LIST_HEAD(&cachep->memcg_params.destroyed_list);
>> +       cachep->memcg_params.id = id;
>> +}
>> +
>> +void mem_cgroup_release_cache(struct kmem_cache *cachep)
```

```
>> +{
>> +       __clear_bit(cachep->memcg_params.id, cache_types);
>> +}
>>   #endif /* CONFIG_CGROUP_MEM_RES_CTLR_KMEM */
>>
>>   static void drain_all_stock_async(struct mem_cgroup *memcg);
>> diff --git a/mm/slub.c b/mm/slub.c
>> index 2652e7c..86e40cc 100644
>> --- a/mm/slub.c
>> +++ b/mm/slub.c
>> @@ -32,6 +32,7 @@
>>   #include<linux/prefetch.h>
>>
>>   #include<trace/events/kmem.h>
>> +#include<linux/memcontrol.h>
>>
>>   /*
>>   * Lock order:
>> @@ -3880,7 +3881,7 @@ static int slab_unmergeable(struct kmem_cache *s)
>>         return 0;
>>   }
>>
>> -static struct kmem_cache *find_mergeable(size_t size,
>> +static struct kmem_cache *find_mergeable(struct mem_cgroup *memcg, size_t size,
>>               size_t align, unsigned long flags, const char *name,
>>               void (*ctor)(void *))
>>   {
>> @@ -3916,21 +3917,29 @@ static struct kmem_cache *find_mergeable(size_t size,
>>               if (s->size - size>= sizeof(void *))
>>                     continue;
>>
>> +             if (memcg&&  s->memcg_params.memcg != memcg)
>> +                   continue;
>> +
>>               return s;
>>         }
>>         return NULL;
>>   }
>>
>> -struct kmem_cache *kmem_cache_create(const char *name, size_t size,
>> -             size_t align, unsigned long flags, void (*ctor)(void *))
>> +struct kmem_cache *
>> +kmem_cache_create_memcg(struct mem_cgroup *memcg, const char *name, size_t size,
>> +                   size_t align, unsigned long flags, void (*ctor)(void *))
>>   {
>>         struct kmem_cache *s;
>>
>>         if (WARN_ON(!name))
```

```
>>                 return NULL;
>>
>> +#ifndef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
>> +       WARN_ON(memcg != NULL);
>> +#endif
>> +
>>         down_write(&slub_lock);
>> -       s = find_mergeable(size, align, flags, name, ctor);
>> +       s = find_mergeable(memcg, size, align, flags, name, ctor);
>>         if (s) {
>>                 s->refcount++;
>>                 /*
>> @@ -3954,12 +3963,15 @@ struct kmem_cache *kmem_cache_create(const char *name,
size_t size,
>>                                 size, align, flags, ctor)) {
>>                         list_add(&s->list,&slab_caches);
>>                         up_write(&slub_lock);
>> +                       mem_cgroup_register_cache(memcg, s);
>
> Do the kmalloc caches get their id registered correctly?
>
```

For the slub, it seems to work okay. But I had to use the trick that for the memcg-specific kmalloc caches, they come from the normal caches rather than the special kmalloc pool. Since we are already paying the penalty of dealing with the memcg finding, I hope this is okay.

For the slab, my investigation wasn't that deep. But basic functionality works okay.

Subject: Re: [PATCH 17/23] kmem controller charge/uncharge infrastructure
Posted by Glauber Costa on Wed, 02 May 2012 15:34:44 GMT
View Forum Message <> Reply to Message

On 04/30/2012 05:56 PM, Suleiman Souhlal wrote:

```
>> +
>> +static void kmem_cache_destroy_work_func(struct work_struct *w)
>> +{
>> +       struct kmem_cache *cachep;
>> +       char *name;
>> +
>> +       spin_lock_irq(&cache_queue_lock);
>> +       while (!list_empty(&destroyed_caches)) {
>> +               cachep = container_of(list_first_entry(&destroyed_caches,
>> +                   struct mem_cgroup_cache_params, destroyed_list), struct
>> +                   kmem_cache, memcg_params);
>> +               name = (char *)cachep->name;
```

>> +		list_del(&cachep->memcg_params.destroyed_list);
>> +		spin_unlock_irq(&cache_queue_lock);
>> +		synchronize_rcu();
>
> Is this synchronize_rcu() still needed, now that we don't use RCU to
> protect memcgs from disappearing during allocation anymore?
>
> Also, should we drop the memcg reference we got in
> memcg_create_kmem_cache() here?

I have a reworked code I like better for this part.

It reads as follows:

```
static void kmem_cache_destroy_work_func(struct work_struct *w)
{
	struct kmem_cache *cachep;
	const char *name;
	struct mem_cgroup_cache_params *params, *tmp;
	unsigned long flags;
	LIST_HEAD(delete_unlocked);

	synchronize_rcu();

	spin_lock_irqsave(&cache_queue_lock, flags);
	list_for_each_entry_safe(params, tmp, &destroyed_caches,
destroyed_list) {
		cachep = container_of(params, struct kmem_cache,
memcg_params);
		list_move(&cachep->memcg_params.destroyed_list,
&delete_unlocked);
	}
	spin_unlock_irqrestore(&cache_queue_lock, flags);

	list_for_each_entry_safe(params, tmp, &delete_unlocked,
destroyed_list) {
		cachep = container_of(params, struct kmem_cache,
memcg_params);
		list_del(&cachep->memcg_params.destroyed_list);
		name = cachep->name;
		mem_cgroup_put(cachep->memcg_params.memcg);
		kmem_cache_destroy(cachep);
		kfree(name);
	}
}
```

I think having a list in stack is better because we don't need to hold &
drop the spinlock, and can achieve more parallelism if multiple cpus are

scheduling the destroy worker.

As you see, this version does a put() - so the answer to your question
is yes.

synchronize_rcu() also gets a new meaning, in the sense that it only
waits until everybody that is destroying a cache can have the chance to
get their stuff into the list.

But to be honest, one of the things we need to do for the next version,
is audit all the locking rules and write them down...

```
>> +static void memcg_create_cache_work_func(struct work_struct *w)
>> +{
>> +	struct kmem_cache *cachep;
>> +	struct create_work *cw;
>> +
>> +	spin_lock_irq(&cache_queue_lock);
>> +	while (!list_empty(&create_queue)) {
>> +		cw = list_first_entry(&create_queue, struct create_work, list);
>> +		list_del(&cw->list);
>> +		spin_unlock_irq(&cache_queue_lock);
>> +		cachep = memcg_create_kmem_cache(cw->memcg, cw->cachep);
>> +		if (cachep == NULL)
>> +			printk(KERN_ALERT
>> +			"%s: Couldn't create memcg-cache for %s memcg %s\n",
>> +			__func__, cw->cachep->name,
>> +			cw->memcg->css.cgroup->dentry->d_name.name);
>
> We might need rcu_dereference() here (and hold rcu_read_lock()).
> Or we could just remove this message.
```

Don't understand this "or". Again, cache creation can still fail. This
is specially true in constrained memory situations.

```
>> +/*
>> + * Return the kmem_cache we're supposed to use for a slab allocation.
>> + * If we are in interrupt context or otherwise have an allocation that
>> + * can't fail, we return the original cache.
>> + * Otherwise, we will try to use the current memcg's version of the cache.
>> + *
>> + * If the cache does not exist yet, if we are the first user of it,
>> + * we either create it immediately, if possible, or create it asynchronously
>> + * in a workqueue.
>> + * In the latter case, we will let the current allocation go through with
>> + * the original cache.
>> + *
>> + * This function returns with rcu_read_lock() held.
```

```
>> + */
>> +struct kmem_cache *__mem_cgroup_get_kmem_cache(struct kmem_cache *cachep,
>> +                                               gfp_t gfp)
>> +{
>> +       struct mem_cgroup *memcg;
>> +       int idx;
>> +
>> +       gfp |=  cachep->allocflags;
>> +
>> +       if ((current->mm == NULL))
>> +               return cachep;
>> +
>> +       if (cachep->memcg_params.memcg)
>> +               return cachep;
>> +
>> +       idx = cachep->memcg_params.id;
>> +       VM_BUG_ON(idx == -1);
>> +
>> +       memcg = mem_cgroup_from_task(current);
>> +       if (!mem_cgroup_kmem_enabled(memcg))
>> +               return cachep;
>> +
>> +       if (rcu_access_pointer(memcg->slabs[idx]) == NULL) {
>> +               memcg_create_cache_enqueue(memcg, cachep);
>> +               return cachep;
>> +       }
>> +
>> +       return rcu_dereference(memcg->slabs[idx]);
>
> Is it ok to call rcu_access_pointer() and rcu_dereference() without
> holding rcu_read_lock()?
```

No, but mem_cgroup_from_task should be called with rcu_read_lock() held
as well.

I forgot to change it in the comments, but this function should be
called with the rcu_read_lock() held. I was careful enough to check the
callers, and they do. But being only human, some of them might have
escaped...

```
>> +
>> +bool __mem_cgroup_charge_kmem(gfp_t gfp, size_t size)
>> +{
>> +       struct mem_cgroup *memcg;
>> +       bool ret = true;
>> +
>> +       rcu_read_lock();
>> +       memcg = mem_cgroup_from_task(current);
```

```
>> +
>> +       if (!mem_cgroup_kmem_enabled(memcg))
>> +               goto out;
>> +
>> +       mem_cgroup_get(memcg);
>
> Why do we need to get a reference to the memcg for every charge?
> How will this work when deleting a memcg?
```

There are two charging functions here:

mem_cgroup_charge_slab() and mem_cgroup_charge_kmem() (the later had its
name changed in my private branch, for the next submission)

The slub allocator will draw large kmalloc allocations directly from the
page allocator, which is the case this function is designed to handle.

Since we have no cache to bill this against, we need to hold the
reference here.

```
>> +       _memcg = memcg;
>> +       ret = __mem_cgroup_try_charge(NULL, gfp, delta / PAGE_SIZE,
>> +&_memcg, may_oom);
>> +       if (ret == -ENOMEM)
>> +               return ret;
>> +       else if ((ret == -EINTR) || (ret&&  (gfp&  __GFP_NOFAIL)))  {
>> +               nofail = true;
>> +               /*
>> +                * __mem_cgroup_try_charge() chose 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, force the
>> +                * change, going above the limit if needed.
>> +                */
>> +               res_counter_charge_nofail(&memcg->res, delta,&fail_res);
>
> We might need to charge memsw here too.
```

hummm, isn't there a more automated way to do that ?

I'll take a look.

> Might need to uncharge memsw.

Here too.

Subject: Re: [PATCH 19/23] slab: per-memcg accounting of slab caches
Posted by Glauber Costa on Wed, 02 May 2012 15:40:37 GMT
View Forum Message <> Reply to Message

>> @@ -3834,11 +3866,15 @@ static inline void __cache_free(struct kmem_cache *cachep,
void *objp,
>> */
>> void *kmem_cache_alloc(struct kmem_cache *cachep, gfp_t flags)
>> {
>> -	void *ret = __cache_alloc(cachep, flags, __builtin_return_address(0));
>> +	void *ret;
>> +
>> +	rcu_read_lock();
>> +	cachep = mem_cgroup_get_kmem_cache(cachep, flags);
>> +	rcu_read_unlock();
>
> Don't we need to check in_interrupt(), current, __GFP_NOFAIL every
> time we call mem_cgroup_cgroup_get_kmem_cache()?
>
> I would personally prefer if those checks were put inside
> mem_cgroup_get_kmem_cache() instead of having to check for every
> caller.
>

in_interrupt() yes, __GFP_NOFAIL I don't think so.

__GFP_NOFAIL should lead to a res_counter_charge_nofail() in the end.
The name similarity is no coincidence...

 From a code style PoV, it makes sense to bundle an in_interrupt() check
here, but from a performance PoV, putting it in the callers can help us
avoid the price of a function call.

But well, looking at the code, I see it is not there as well... =(

I plan to change memcontrol.h to look like this:

```
static __always_inline struct kmem_cache *
mem_cgroup_get_kmem_cache(struct kmem_cache *cachep, gfp_t gfp)
{
	if (mem_cgroup_kmem_on && current->mm && !in_interrupt())
		return __mem_cgroup_get_kmem_cache(cachep, gfp);
	return cachep;
}
```