Subject: [PATCH v2 00/29] kmem limitation for memcg
Posted by Glauber Costa on Fri, 11 May 2012 17:44:02 GMT
View Forum Message <> Reply to Message

Hello All,

This is my new take for the memcg kmem accounting.
At this point, I consider the series pretty mature - although of course,
bugs are always there...

As a disclaimer, however, I must say that the slub code is much more stressed
by me, since I know it better. If you have no more objections to the concepts
presented, the remaining edges can probably be polished in a rc cycle,
at the maintainers discretion, of course.

Otherwise, I'll be happy to address any concerns of yours.

Since last submission:

 * memcgs can be properly removed.
 * We are not charging based on current->mm->owner instead of current
 * kmem_large allocations for slub got some fixes, specially for the free case
 * A cache that is registered can be properly removed (common module case)
   even if it spans memcg children. Slab had some code for that, now it works
   well with both
 * A new mechanism for skipping allocations is proposed (patch posted
   separately already). Now instead of having kmalloc_no_account, we mark
   a region as non-accountable for memcg.

I should point out again that most, if not all, of the code in the caches
are wrapped in static_key areas, meaning they will be completely patched out
until the first limit is set.

I also put a lot of effort, as you will all see, in the proper separation
of the patches, so the review process is made as easy as the complexity of
the work allows to.

Frederic Weisbecker (1):
  cgroups: ability to stop res charge propagation on bounded ancestor

Glauber Costa (24):
  slab: dup name string
  slub: fix slab_state for slub
  memcg: Always free struct memcg through schedule_work()
  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
  memcg: change defines to an enum

```
  res_counter: 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
  slab: create duplicate cache
  memcg: kmem controller charge/uncharge infrastructure
  skip memcg kmem allocations in specified code regions
  slub: charge allocation to a memcg
  slab: per-memcg accounting of slab caches
  memcg: disable kmem code when not in use.
  memcg: destroy memcg caches
  memcg/slub: shrink dead caches
  slub: create slabinfo file for memcg
  slub: track all children of a kmem cache
  Documentation: add documentation for slab tracker 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.

 Documentation/cgroups/memory.txt           |   33 ++
 Documentation/cgroups/resource_counter.txt |   18 +-
 include/linux/memcontrol.h                 |   88 ++++
 include/linux/res_counter.h                |   23 +-
 include/linux/sched.h                      |    1 +
 include/linux/slab.h                       |   29 +
 include/linux/slab_def.h                   |   72 +++-
 include/linux/slub_def.h                   |   51 ++-
 init/Kconfig                               |    2 +-
 kernel/res_counter.c                       |   13 +-
 mm/memcontrol.c                            |  773 ++++++++++++++++++++++++++--
 mm/slab.c                                  |  394 +++++++++++++---
 mm/slub.c                                  |  298 ++++++++++-
 13 files changed, 1658 insertions(+), 137 deletions(-)

--
1.7.7.6
```

Subject: [PATCH v2 01/29] slab: dup name string
Posted by Glauber Costa on Fri, 11 May 2012 17:44:03 GMT

View Forum Message <> Reply to Message

The slub allocator creates a copy of the name string, and
frees it later. I would like them both to behave the same,
whether it is the slab starting to create a copy of it itself,
or the slub ceasing to.

This is because when I create memcg copies of it, I have to
kmalloc strings for the new names, and having the allocators to
behave differently here, would make it a lot uglier.

My first submission removed the duplication for the slub. But
the code started to get a bit complicated when dealing with
deletion of chained caches. Also, Christoph voiced his opinion
that patching the slab to keep copies would be better.

So here it is.

Signed-off-by: Glauber Costa <glommer@parallels.com>
CC: Christoph Lameter <cl@linux.com>
CC: Pekka Enberg <penberg@cs.helsinki.fi>

```
---
 mm/slab.c |    3 ++-
 1 files changed, 2 insertions(+), 1 deletions(-)

diff --git a/mm/slab.c b/mm/slab.c
index e901a36..91b9c13 100644
--- a/mm/slab.c
+++ b/mm/slab.c
@@ -2118,6 +2118,7 @@ static void __kmem_cache_destroy(struct kmem_cache *cachep)
    kfree(l3);
   }
  }
+ kfree(cachep->name);
  kmem_cache_free(&cache_cache, cachep);
 }

@@ -2526,7 +2527,7 @@ 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 = kstrdup(name, GFP_KERNEL);

  if (setup_cpu_cache(cachep, gfp)) {
   __kmem_cache_destroy(cachep);
--
1.7.7.6
```

Subject: [PATCH v2 02/29] slub: fix slab_state for slub
Posted by Glauber Costa on Fri, 11 May 2012 17:44:04 GMT

View Forum Message <> Reply to Message

When the slub code wants to know if the sysfs state has already been
initialized, it tests for slab_state == SYSFS. This is quite fragile,
since new state can be added in the future (it is, in fact, for
memcg caches). This patch fixes this behavior so the test matches
>= SYSFS, as all other state does.

Signed-off-by: Glauber Costa <glommer@parallels.com>
CC: Christoph Lameter <cl@linux.com>
CC: Pekka Enberg <penberg@cs.helsinki.fi>

```
---
 mm/slub.c |    2 +-
 1 files changed, 1 insertions(+), 1 deletions(-)

diff --git a/mm/slub.c b/mm/slub.c
index ffe13fd..226e053 100644
--- a/mm/slub.c
+++ b/mm/slub.c
@@ -5356,7 +5356,7 @@ static int sysfs_slab_alias(struct kmem_cache *s, const char *name)
 {
 	struct saved_alias *al;

-	if (slab_state == SYSFS) {
+	if (slab_state >= SYSFS) {
 		/*
 		 * If we have a leftover link then remove it.
 		 */
-- 
1.7.7.6
```

Subject: [PATCH v2 03/29] memcg: Always free struct memcg through schedule_work()
Posted by Glauber Costa on Fri, 11 May 2012 17:44:05 GMT

View Forum Message <> Reply to Message

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

This patch unifies this behavior, by ensuring that even kfree will
happen in a separate thread. The goal is to have a stable place to
call the upcoming jump label destruction function outside the realm
of the complicated and quite far-reaching cgroup lock (that can't be

held when calling neither the cpu_hotplug.lock nor the jump_label_mutex)

Signed-off-by: Glauber Costa <glommer@parallels.com>
CC: Tejun Heo <tj@kernel.org>
CC: Li Zefan <lizefan@huawei.com>
CC: Kamezawa Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com>
CC: Johannes Weiner <hannes@cmpxchg.org>
CC: Michal Hocko <mhocko@suse.cz>

```
---
 mm/memcontrol.c |   24 +++++++++++++-----------
 1 files changed, 13 insertions(+), 11 deletions(-)

diff --git a/mm/memcontrol.c b/mm/memcontrol.c
index 932a734..0b4b4c8 100644
--- a/mm/memcontrol.c
+++ b/mm/memcontrol.c
@@ -245,8 +245,8 @@ struct mem_cgroup {
    */
   struct rcu_head rcu_freeing;
   /*
-   * But when using vfree(), that cannot be done at
-   * interrupt time, so we must then queue the work.
+   * We also need some space for a worker in deferred freeing.
+   * By the time we call it, rcu_freeing is not longer in use.
    */
   struct work_struct work_freeing;
  };
@@ -4826,23 +4826,28 @@ out_free:
 }

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

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

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

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

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

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

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

Subject: [PATCH v2 04/29] slub: always get the cache from its page in kfree
Posted by Glauber Costa on Fri, 11 May 2012 17:44:06 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>
CC: Christoph Lameter <cl@linux.com>
CC: Pekka Enberg <penberg@cs.helsinki.fi>

```
---
 mm/slub.c |    2 +-
 1 files changed, 1 insertions(+), 1 deletions(-)

diff --git a/mm/slub.c b/mm/slub.c
index 226e053..e606f1b 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 v2 05/29] slab: rename gfpflags to allocflags
Posted by Glauber Costa on Fri, 11 May 2012 17:44:07 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>
CC: Christoph Lameter <cl@linux.com>
CC: Pekka Enberg <penberg@cs.helsinki.fi>

```
---
 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 91b9c13..8a851ed 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;

@@ -2509,9 +2509,9 @@ 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);

@@ -2858,9 +2858,9 @@ 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 v2 06/29] memcg: Make it possible to use the stock for more than one page.

Posted by Glauber Costa on Fri, 11 May 2012 17:44:08 GMT

View Forum Message <> Reply to Message

From: Suleiman Souhlal <ssouhlal@FreeBSD.org>

Signed-off-by: Suleiman Souhlal <suleiman@google.com>
Signed-off-by: Glauber Costa <glommer@parallels.com>
Acked-by: Kamezawa Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com>

```
---
 mm/memcontrol.c |   18 +++++++++---------
 1 files changed, 9 insertions(+), 9 deletions(-)

diff --git a/mm/memcontrol.c b/mm/memcontrol.c
index 0b4b4c8..248d80b 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 v2 07/29] memcg: Reclaim when more than one page needed.
Posted by Glauber Costa on Fri, 11 May 2012 17:44:09 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>
Signed-off-by: Glauber Costa <glommer@parallels.com>
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 248d80b..47d3979 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 v2 08/29] slab: use obj_size field of struct kmem_cache when not debugging

Posted by Glauber Costa on Fri, 11 May 2012 17:44:10 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 +++++++++++++++++++++++++++-----------
 2 files changed, 29 insertions(+), 12 deletions(-)

diff --git a/include/linux/slab_def.h b/include/linux/slab_def.h
index d41effe..cba3139 100644
--- a/include/linux/slab_def.h
+++ b/include/linux/slab_def.h
@@ -78,8 +78,10 @@ struct kmem_cache {
    * variables contain the offset to the user object and its size.
    */
   int obj_offset;
- int obj_size;
 #endif /* CONFIG_DEBUG_SLAB */
+#if defined(CONFIG_DEBUG_SLAB) || defined(CONFIG_CGROUP_MEM_RES_CTLR_KMEM)
+ int obj_size;
+#endif

 /* 6) per-cpu/per-node data, touched during every alloc/free */
  /*
diff --git a/mm/slab.c b/mm/slab.c
index 8a851ed..56f2ba8 100644
--- 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 =
@@ -2419,8 +2433,9 @@ 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 v2 09/29] memcg: change defines to an enum
Posted by Glauber Costa on Fri, 11 May 2012 17:44:11 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: Johannes Weiner <hannes@cmpxchg.org>
Acked-by: Kamezawa Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com>

```
---
 mm/memcontrol.c |    9 ++++++---
 1 files changed, 6 insertions(+), 3 deletions(-)

diff --git a/mm/memcontrol.c b/mm/memcontrol.c
index 47d3979..789ca5a 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 v2 10/29] res_counter: don't force return value checking in res_counter_charge_nofail

Posted by Glauber Costa on Fri, 11 May 2012 17:44:12 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>
Acked-by: 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 v2 11/29] cgroups: ability to stop res charge propagation on bounded ancestor

Posted by Glauber Costa on Fri, 11 May 2012 17:44:13 GMT

View Forum Message <> Reply to Message

From: Frederic Weisbecker <fweisbec@gmail.com>

Moving a task from a cgroup to another may require to substract its
resource charge from the old cgroup and add it to the new one.

For this to happen, the uncharge/charge propagation can just stop when we
reach the common ancestor for the two cgroups. Further the performance
reasons, we also want to avoid to temporarily overload the common
ancestors with a non-accurate resource counter usage if we charge first
the new cgroup and uncharge the old one thereafter. This is going to be a
requirement for the coming max number of task subsystem.

To solve this, provide a pair of new API that can charge/uncharge a
resource counter until we reach a given ancestor.

Signed-off-by: Frederic Weisbecker <fweisbec@gmail.com>
Acked-by: Paul Menage <paul@paulmenage.org>
Acked-by: Glauber Costa <glommer@parallels.com>
Cc: Li Zefan <lizf@cn.fujitsu.com>
Cc: Johannes Weiner <hannes@cmpxchg.org>
Cc: Aditya Kali <adityakali@google.com>
Cc: Oleg Nesterov <oleg@redhat.com>
Cc: Kay Sievers <kay.sievers@vrfy.org>
Cc: Tim Hockin <thockin@hockin.org>
Cc: Tejun Heo <htejun@gmail.com>
Acked-by: Kirill A. Shutemov <kirill@shutemov.name>
Signed-off-by: Andrew Morton <akpm@linux-foundation.org>

```
---
 Documentation/cgroups/resource_counter.txt |   18 +++++++++++++++++-
 include/linux/res_counter.h                |   21 ++++++++++++++++++---
 kernel/res_counter.c                       |   13 ++++++++-----
 3 files changed, 43 insertions(+), 9 deletions(-)

diff --git a/Documentation/cgroups/resource_counter.txt
b/Documentation/cgroups/resource_counter.txt
index 95b24d7..a2cd05b 100644
```

```
--- a/Documentation/cgroups/resource_counter.txt
+++ b/Documentation/cgroups/resource_counter.txt
@@ -83,7 +83,15 @@ to work with it.
  res_counter->lock internally (it must be called with res_counter->lock
  held).

- e. void res_counter_uncharge[_locked]
+ e. int res_counter_charge_until(struct res_counter *counter,
+       struct res_counter *limit, unsigned long val,
+       struct res_counter **limit_fail_at)
+
+ The same as res_counter_charge(), but the charge propagation to
+ the hierarchy stops at the limit given in the "limit" parameter.
+
+
+ f. void res_counter_uncharge[_locked]
    (struct res_counter *rc, unsigned long val)

  When a resource is released (freed) it should be de-accounted
@@ -92,6 +100,14 @@ to work with it.

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

+
+ g. void res_counter_uncharge_until(struct res_counter *counter,
+    struct res_counter *limit,
+    unsigned long val)
+
+ The same as res_counter_charge, but the uncharge propagation to
+ the hierarchy stops at the limit given in the "limit" parameter.
+
  2.1 Other accounting routines

     There are more routines that may help you with common needs, like
diff --git a/include/linux/res_counter.h b/include/linux/res_counter.h
index f7621cf..c12143e 100644
--- a/include/linux/res_counter.h
+++ b/include/linux/res_counter.h
@@ -117,11 +117,20 @@ void res_counter_init(struct res_counter *counter, struct res_counter
*parent);

 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_until(struct res_counter *counter,
+      struct res_counter *limit,
+      unsigned long val,
+       struct res_counter **limit_fail_at);
 int res_counter_charge_nofail(struct res_counter *counter,
   unsigned long val, struct res_counter **limit_fail_at);

+static inline int __must_check
+res_counter_charge(struct res_counter *counter, unsigned long val,
+     struct res_counter **limit_fail_at)
+{
+ return res_counter_charge_until(counter, NULL, val, limit_fail_at);
+}
+
 /*
  * uncharge - tell that some portion of the resource is released
  *
@@ -133,7 +142,13 @@ int res_counter_charge_nofail(struct res_counter *counter,
  */

 void res_counter_uncharge_locked(struct res_counter *counter, unsigned long val);
-void res_counter_uncharge(struct res_counter *counter, unsigned long val);
+void res_counter_uncharge_until(struct res_counter *counter,
+    struct res_counter *limit, unsigned long val);
+static inline void res_counter_uncharge(struct res_counter *counter,
+     unsigned long val)
+{
+ res_counter_uncharge_until(counter, NULL, val);
+}

 /**
  * res_counter_margin - calculate chargeable space of a counter
diff --git a/kernel/res_counter.c b/kernel/res_counter.c
index d508363..7aebf96 100644
--- a/kernel/res_counter.c
+++ b/kernel/res_counter.c
@@ -35,8 +35,9 @@ int res_counter_charge_locked(struct res_counter *counter, unsigned long val)
  return 0;
 }

-int res_counter_charge(struct res_counter *counter, unsigned long val,
-   struct res_counter **limit_fail_at)
+int res_counter_charge_until(struct res_counter *counter,
+       struct res_counter *limit, unsigned long val,
+       struct res_counter **limit_fail_at)
 {
  int ret;
  unsigned long flags;
@@ -44,7 +45,7 @@ int res_counter_charge(struct res_counter *counter, unsigned long val,
  *limit_fail_at = NULL;
  local_irq_save(flags);
- for (c = counter; c != NULL; c = c->parent) {
+ for (c = counter; c != limit; c = c->parent) {
   spin_lock(&c->lock);
   ret = res_counter_charge_locked(c, val);
   spin_unlock(&c->lock);
@@ -99,13 +100,15 @@ void res_counter_uncharge_locked(struct res_counter *counter,
unsigned long val)
  counter->usage -= val;
 }

-void res_counter_uncharge(struct res_counter *counter, unsigned long val)
+void res_counter_uncharge_until(struct res_counter *counter,
+    struct res_counter *limit,
+    unsigned long val)
 {
  unsigned long flags;
  struct res_counter *c;

  local_irq_save(flags);
- for (c = counter; c != NULL; c = c->parent) {
+ for (c = counter; c != limit; c = c->parent) {
   spin_lock(&c->lock);
   res_counter_uncharge_locked(c, val);
   spin_unlock(&c->lock);
--
1.7.7.6
```

Subject: [PATCH v2 12/29] kmem slab accounting basic infrastructure
Posted by Glauber Costa on Fri, 11 May 2012 17:44:14 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: Johannes Weiner <hannes@cmpxchg.org>
Reviewed-by: Kamezawa Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com>

```
 mm/memcontrol.c |   79 +++++++++++++++++++++++++++++++++++++++++++++++++++++++++-
 1 files changed, 78 insertions(+), 1 deletions(-)
```

```
diff --git a/mm/memcontrol.c b/mm/memcontrol.c
index 789ca5a..49b1129 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);
@@ -4981,6 +5049,12 @@ 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;
@@ -4999,6 +5073,7 @@ 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.
@@ -5009,6 +5084,7 @@ 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);
@@ -5016,6 +5092,7 @@ 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 v2 13/29] slab/slub: struct memcg_params
Posted by Glauber Costa on Fri, 11 May 2012 17:44:15 GMT

View Forum Message <> Reply to Message

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     |   14 ++++++++++++++
 include/linux/slab_def.h |    4 ++++
 include/linux/slub_def.h |    3 +++
 3 files changed, 21 insertions(+), 0 deletions(-)

diff --git a/include/linux/slab.h b/include/linux/slab.h
index a595dce..dbf36b5 100644
--- a/include/linux/slab.h
+++ b/include/linux/slab.h
@@ -153,6 +153,20 @@ 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;
+ atomic_t refcnt;
+
+#ifdef CONFIG_SLAB
+ /* Original cache parameters, used when creating a memcg cache */
+ size_t orig_align;
+
+#endif
+};
+#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 v2 14/29] slub: consider a memcg parameter in kmem_create_cache

Posted by Glauber Costa on Fri, 11 May 2012 17:44:16 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.

We can't assign indexes until the basic slab is up and running
this is because the ida subsystem will itself call slab functions
such as kmalloc a couple of times. Because of that, we have
a late_initcall that scan all caches and register them after the
kernel is booted up. Only caches registered after that receive
their index right away.

This index mechanism was developed by Suleiman Souhlal.
Changed to a idr/ida based approach based on suggestion
from Kamezawa.

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            |   27 ++++++++++++++++
 mm/slub.c                  |   67 ++++++++++++++++++++++++++++++++++++---
 4 files changed, 109 insertions(+), 5 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 dbf36b5..1386650 100644
--- a/include/linux/slab.h
+++ b/include/linux/slab.h
@@ -320,6 +320,12 @@ 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 49b1129..9327996 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 +466,27 @@ struct cg_proto *tcp_proto_cgroup(struct mem_cgroup *memcg)
 }
 EXPORT_SYMBOL(tcp_proto_cgroup);
 #endif /* CONFIG_INET */
+
+struct ida 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 = ida_simple_get(&cache_types, 0, MAX_KMEM_CACHE_TYPES,
+        GFP_KERNEL);
+ cachep->memcg_params.id = id;
+}
+
+void mem_cgroup_release_cache(struct kmem_cache *cachep)
+{
+ if (cachep->memcg_params.id != -1)
+  ida_simple_remove(&cache_types, cachep->memcg_params.id);
+}
 #endif /* CONFIG_CGROUP_MEM_RES_CTLR_KMEM */

 static void drain_all_stock_async(struct mem_cgroup *memcg);
@@ -5053,6 +5079,7 @@ mem_cgroup_create(struct cgroup *cont)
 #ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
   WARN_ON(cgroup_add_cftypes(&mem_cgroup_subsys,
         kmem_cgroup_files));
+  ida_init(&cache_types);
 #endif
```

```
   if (mem_cgroup_soft_limit_tree_init())
diff --git a/mm/slub.c b/mm/slub.c
index e606f1b..1698371 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:
@@ -186,7 +187,8 @@ static enum {
  DOWN,  /* No slab functionality available */
  PARTIAL, /* Kmem_cache_node works */
  UP,  /* Everything works but does not show up in sysfs */
- SYSFS  /* Sysfs up */
+ SYSFS,  /* Sysfs up */
+ MEMCG  /* Memcg registered */
 } slab_state = DOWN;

 /* A list of all slab caches on the system */
@@ -3193,6 +3195,11 @@ 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);
+#endif
   up_write(&slub_lock);
   if (kmem_cache_close(s)) {
    printk(KERN_ERR "SLUB %s: %s called for cache that "
@@ -3860,6 +3867,33 @@ void __init kmem_cache_init_late(void)
 {
 }

+static int __init memcg_slab_register_all(void)
+{
+ struct kmem_cache *s;
+ int i;
+
+ for (i = 0; i < SLUB_PAGE_SHIFT; i++) {
+  struct kmem_cache *s;
+  s = kmalloc_caches[i];
+  if (s)
+   mem_cgroup_register_cache(NULL, s);
+  s = kmalloc_dma_caches[i];
+  if (s)
+   mem_cgroup_register_cache(NULL, s);
+ }
+
+
+
+ down_write(&slub_lock);
+ list_for_each_entry(s, &slab_caches, list)
+  mem_cgroup_register_cache(NULL, s);
+
+ slab_state = MEMCG;
+ up_write(&slub_lock);
+ return 0;
+}
+late_initcall(memcg_slab_register_all);
+
 /*
  * Find a mergeable slab cache
  */
@@ -3880,7 +3914,7 @@ 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 *))
 {
@@ -3916,13 +3950,19 @@ static struct kmem_cache *find_mergeable(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;
  }
  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;
  char *n;
@@ -3930,8 +3970,12 @@ struct kmem_cache *kmem_cache_create(const char *name, size_t
size,
  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++;
   /*
@@ -3959,6 +4003,8 @@ 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);
+   if (slab_state >= MEMCG)
+    mem_cgroup_register_cache(memcg, s);
    if (sysfs_slab_add(s)) {
     down_write(&slub_lock);
     list_del(&s->list);
@@ -3980,6 +4026,12 @@ 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
@@ -5273,6 +5325,11 @@ 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 v2 15/29] slab: pass memcg parameter to kmem_cache_create
Posted by Glauber Costa on Fri, 11 May 2012 17:44:17 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 |   74 +++++++++++++++++++++++++++++++++++++++++++++++++++++++++++------
 1 files changed, 66 insertions(+), 8 deletions(-)

diff --git a/mm/slab.c b/mm/slab.c
index 56f2ba8..d05a326 100644
--- a/mm/slab.c
+++ b/mm/slab.c
@@ -502,6 +502,9 @@ EXPORT_SYMBOL(slab_buffer_size);
 #define SLAB_MAX_ORDER_LO 0
 static int slab_max_order = SLAB_MAX_ORDER_LO;
 static bool slab_max_order_set __initdata;
+#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
+static int slab_memcg_init __read_mostly;
+#endif

 /*
  * Functions for storing/retrieving the cachep and or slab from the page
```

```
@@ -2288,14 +2291,15 @@ 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.
   */
@@ -2312,7 +2316,6 @@ 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) {
@@ -2332,9 +2335,9 @@ 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;
   }
@@ -2435,6 +2438,9 @@ 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

  /*
@@ -2544,6 +2550,11 @@ kmem_cache_create (const char *name, size_t size, size_t align,
  cachep->ctor = ctor;
  cachep->name = kstrdup(name, GFP_KERNEL);

+#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
+ if (slab_memcg_init)
+  mem_cgroup_register_cache(memcg, cachep);
+#endif
+
  if (setup_cpu_cache(cachep, gfp)) {
   __kmem_cache_destroy(cachep);
   cachep = NULL;
@@ -2567,13 +2578,54 @@ 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);

+#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
+static int __init memcg_slab_register_all(void)
+{
+ struct kmem_cache *cachep;
+ struct cache_sizes *sizes;
+
+ mem_cgroup_register_cache(NULL, &cache_cache);
+
+ sizes = malloc_sizes;
```

```
+
+ while (sizes->cs_size != ULONG_MAX) {
+  if (sizes->cs_cachep)
+   mem_cgroup_register_cache(NULL, sizes->cs_cachep);
+  if (sizes->cs_dmacachep)
+   mem_cgroup_register_cache(NULL, sizes->cs_dmacachep);
+  sizes++;
+ }
+
+ mutex_lock(&cache_chain_mutex);
+ list_for_each_entry(cachep, &cache_chain, next)
+  mem_cgroup_register_cache(NULL, cachep);
+
+ slab_memcg_init = 1;
+ mutex_unlock(&cache_chain_mutex);
+ return 0;
+}
+late_initcall(memcg_slab_register_all);
+#endif /* CONFIG_CGROUP_MEM_RES_CTLR_KMEM */
+
 #if DEBUG
 static void check_irq_off(void)
 {
@@ -2768,6 +2820,12 @@ 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);
+#endif
+
  __kmem_cache_destroy(cachep);
  mutex_unlock(&cache_chain_mutex);
  put_online_cpus();
--
1.7.7.6
```

Subject: [PATCH v2 16/29] slub: create duplicate cache
Posted by Glauber Costa on Fri, 11 May 2012 17:44:18 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 |    2 ++
 include/linux/slab.h       |    2 ++
 mm/memcontrol.c            |   17 +++++++++++++++++
 mm/slub.c                  |   31 +++++++++++++++++++++++++++++++
 4 files changed, 52 insertions(+), 0 deletions(-)

diff --git a/include/linux/memcontrol.h b/include/linux/memcontrol.h
index 99e14b9..f93021a 100644
--- a/include/linux/memcontrol.h
+++ b/include/linux/memcontrol.h
@@ -445,6 +445,8 @@ 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 1386650..e73ef71 100644
--- a/include/linux/slab.h
+++ b/include/linux/slab.h
@@ -322,6 +322,8 @@ 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);
 #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 9327996..a8171cb 100644
--- a/mm/memcontrol.c
```

```
+++ b/mm/memcontrol.c
@@ -467,6 +467,23 @@ struct cg_proto *tcp_proto_cgroup(struct mem_cgroup *memcg)
 EXPORT_SYMBOL(tcp_proto_cgroup);
 #endif /* CONFIG_INET */

+char *mem_cgroup_cache_name(struct mem_cgroup *memcg, struct kmem_cache *cachep)
+{
+	char *name;
+	struct dentry *dentry;
+
+	rcu_read_lock();
+	dentry = rcu_dereference(memcg->css.cgroup->dentry);
+	rcu_read_unlock();
+
+	BUG_ON(dentry == NULL);
+
+	name = kasprintf(GFP_KERNEL, "%s(%d:%s)",
+	    cachep->name, css_id(&memcg->css), dentry->d_name.name);
+
+	return name;
+}
+
 struct ida cache_types;

 void mem_cgroup_register_cache(struct mem_cgroup *memcg,
diff --git a/mm/slub.c b/mm/slub.c
index 1698371..9b21b38 100644
--- a/mm/slub.c
+++ b/mm/slub.c
@@ -4034,6 +4034,37 @@ 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 & ~SLAB_PANIC), 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);
+ }
+ kfree(name);
+ return new;
+}
+#endif
+
 #ifdef CONFIG_SMP
 /*
  * Use the cpu notifier to insure that the cpu slabs are flushed when
--
1.7.7.6
```

Subject: [PATCH v2 17/29] slab: create duplicate cache

Posted by Glauber Costa on Fri, 11 May 2012 17:44:19 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 |   32 ++++++++++++++++++++++++++++++++
 1 files changed, 32 insertions(+), 0 deletions(-)
```

```
diff --git a/mm/slab.c b/mm/slab.c
index d05a326..985714a 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
@@ -2598,6 +2600,36 @@ 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;
+ unsigned long flags;
+ char *name;
+
+ name = mem_cgroup_cache_name(memcg, cachep);
+ if (!name)
+  return NULL;
+
+ flags = cachep->flags & ~(SLAB_PANIC|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);
+
+ if (new == NULL)
+  goto out;
+
+ 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);
+out:
+ mutex_unlock(&cache_chain_mutex);
+ kfree(name);
+ return new;
+}
+
 static int __init memcg_slab_register_all(void)
 {
  struct kmem_cache *cachep;
--
1.7.7.6
```

Subject: [PATCH v2 18/29] memcg: kmem controller charge/uncharge infrastructure
Posted by Glauber Costa on Fri, 11 May 2012 17:44:20 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 |   67 ++++++++
 init/Kconfig               |    2 +-
 mm/memcontrol.c            |  379 ++++++++++++++++++++++++++++++++++++++++++++-
 3 files changed, 446 insertions(+), 2 deletions(-)

diff --git a/include/linux/memcontrol.h b/include/linux/memcontrol.h
index f93021a..c555799 100644
--- a/include/linux/memcontrol.h
+++ b/include/linux/memcontrol.h
@@ -21,6 +21,7 @@
 #define _LINUX_MEMCONTROL_H
 #include <linux/cgroup.h>
 #include <linux/vm_event_item.h>
+#include <linux/hardirq.h>

 struct mem_cgroup;
 struct page_cgroup;
@@ -447,6 +448,19 @@ void mem_cgroup_register_cache(struct mem_cgroup *memcg,
 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);
+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_new_kmem_page(struct page *page, gfp_t gfp);
+void __mem_cgroup_free_kmem_page(struct page *page);
+
+struct kmem_cache *
+__mem_cgroup_get_kmem_cache(struct kmem_cache *cachep, gfp_t gfp);
+
+#define mem_cgroup_kmem_on 1
 #else
 static inline void mem_cgroup_register_cache(struct mem_cgroup *memcg,
          struct kmem_cache *s)
@@ -463,6 +477,59 @@ 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_new_kmem_page(a, gfp) false
+#define __mem_cgroup_uncharge_slab(a, b)
+#define __mem_cgroup_free_kmem_page(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 && current->mm && !in_interrupt())
+  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_new_kmem_page(struct page *page, gfp_t gfp)
+{
+ if (mem_cgroup_kmem_on && current->mm && !in_interrupt())
+  return __mem_cgroup_new_kmem_page(page, gfp);
+ return true;
+}
+
+static __always_inline
+void mem_cgroup_free_kmem_page(struct page *page)
+{
+ if (mem_cgroup_kmem_on)
+  __mem_cgroup_free_kmem_page(page);
+}
```

```
 #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 a8171cb..5a7416b 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) {
@@ -484,7 +496,14 @@ char *mem_cgroup_cache_name(struct mem_cgroup *memcg, struct
kmem_cache *cachep)
  return name;
 }

+static inline bool mem_cgroup_kmem_enabled(struct mem_cgroup *memcg)
+{
+ return !mem_cgroup_disabled() && memcg &&
+        !mem_cgroup_is_root(memcg) && memcg->kmem_accounted;
+}
+
 struct ida cache_types;
+static DEFINE_MUTEX(memcg_cache_mutex);

 void mem_cgroup_register_cache(struct mem_cgroup *memcg,
          struct kmem_cache *cachep)
@@ -504,6 +523,298 @@ void mem_cgroup_release_cache(struct kmem_cache *cachep)
  if (cachep->memcg_params.id != -1)
   ida_simple_remove(&cache_types, cachep->memcg_params.id);
 }
+
+
+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;
+ atomic_set(&new_cachep->memcg_params.refcnt, 1);
+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);
+
+/*
+ * 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 create_work *cw, *tmp;
+ unsigned long flags;
+ LIST_HEAD(create_unlocked);
+
+ spin_lock_irqsave(&cache_queue_lock, flags);
+ list_for_each_entry_safe(cw, tmp, &create_queue, list)
+  list_move(&cw->list, &create_unlocked);
+ spin_unlock_irqrestore(&cache_queue_lock, flags);
+
+ list_for_each_entry_safe(cw, tmp, &create_unlocked, list) {
+  list_del(&cw->list);
+  memcg_create_kmem_cache(cw->memcg, cw->cachep);
+  /* Drop the reference gotten when we enqueued. */
+  css_put(&cw->memcg->css);
+  kfree(cw);
+ }
+}
+
+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(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.
+ * We 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.
+ *
+ * Can't be called in interrupt context or from kernel threads.
+ * This function needs to be called 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;
+ struct task_struct *p;
+
+ gfp |=  cachep->allocflags;
+
+ if (cachep->memcg_params.memcg)
+  return cachep;
+
+ idx = cachep->memcg_params.id;
+ VM_BUG_ON(idx == -1);
+
+ p = rcu_dereference(current->mm->owner);
+ memcg = mem_cgroup_from_task(p);
+
+ if (!mem_cgroup_kmem_enabled(memcg))
+  return cachep;
+
+ if (memcg->slabs[idx] == NULL) {
+  memcg_create_cache_enqueue(memcg, cachep);
+  return cachep;
+ }
+
+ return memcg->slabs[idx];
+}
+EXPORT_SYMBOL(__mem_cgroup_get_kmem_cache);
+
+bool __mem_cgroup_new_kmem_page(struct page *page, gfp_t gfp)
+{
```

```
+ struct mem_cgroup *memcg;
+ struct page_cgroup *pc;
+ bool ret = true;
+ size_t size;
+ struct task_struct *p;
+
+ if (!current->mm || in_interrupt())
+  return true;
+
+ rcu_read_lock();
+ p = rcu_dereference(current->mm->owner);
+ memcg = mem_cgroup_from_task(p);
+
+ if (!mem_cgroup_kmem_enabled(memcg))
+  goto out;
+
+ mem_cgroup_get(memcg);
+
+ size = (1 << compound_order(page)) << PAGE_SHIFT;
+
+ ret = memcg_charge_kmem(memcg, gfp, size) == 0;
+ if (!ret) {
+  mem_cgroup_put(memcg);
+  goto out;
+ }
+
+ pc = lookup_page_cgroup(page);
+ lock_page_cgroup(pc);
+ pc->mem_cgroup = memcg;
+ SetPageCgroupUsed(pc);
+ unlock_page_cgroup(pc);
+
+out:
+ rcu_read_unlock();
+ return ret;
+}
+EXPORT_SYMBOL(__mem_cgroup_new_kmem_page);
+
+void __mem_cgroup_free_kmem_page(struct page *page)
+{
+ struct mem_cgroup *memcg;
+ size_t size;
+ struct page_cgroup *pc;
+
+ if (mem_cgroup_disabled())
+  return;
+
+ pc = lookup_page_cgroup(page);
+ lock_page_cgroup(pc);
+ memcg = pc->mem_cgroup;
+ pc->mem_cgroup = NULL;
+ if (!PageCgroupUsed(pc)) {
+  unlock_page_cgroup(pc);
+  return;
+ }
+ ClearPageCgroupUsed(pc);
+ unlock_page_cgroup(pc);
+
+ /*
+  * The classical disabled check won't work
+  * for uncharge, since it is possible that the user enabled
+  * kmem tracking, allocated, and then disabled.
+  *
+  * We trust if there is a memcg associated with the page,
+  * it is a valid allocation
+  */
+
+ if (!memcg)
+  return;
+
+ WARN_ON(mem_cgroup_is_root(memcg));
+ size = (1 << compound_order(page)) << PAGE_SHIFT;
+ memcg_uncharge_kmem(memcg, size);
+ mem_cgroup_put(memcg);
+}
+EXPORT_SYMBOL(__mem_cgroup_free_kmem_page);
+
+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;
+ rcu_read_unlock();
+
+ /*
+  * The classical disabled check won't work
+  * for uncharge, since it is possible that the user enabled
+  * kmem tracking, allocated, and then disabled.
+  *
+  * We trust if there is a memcg associated with the slab,
+  * it is a valid allocation
+  */
+ if (!memcg)
+  return;
+
+ 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++)
+  memcg->slabs[i] = NULL;
+}
 #endif /* CONFIG_CGROUP_MEM_RES_CTLR_KMEM */

 static void drain_all_stock_async(struct mem_cgroup *memcg);
@@ -4760,7 +5071,11 @@ 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)
@@ -5777,3 +6092,65 @@ 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 == -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 (do_swap_account)
+   res_counter_charge_nofail(&memcg->memsw, delta,
+        &fail_res);
+ } else if (ret == -ENOMEM)
+  return ret;
+
+ 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);
+  if (do_swap_account)
+   res_counter_uncharge(&memcg->memsw, 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);
+ if (do_swap_account)
+  res_counter_uncharge(&memcg->memsw, delta);
+}
+#endif /* CONFIG_CGROUP_MEM_RES_CTLR_KMEM */
--
1.7.7.6
```

Subject: [PATCH v2 19/29] skip memcg kmem allocations in specified code regions

Posted by Glauber Costa on Fri, 11 May 2012 17:44:21 GMT

View Forum Message <> Reply to Message

This patch creates a mechanism that skip memcg allocations during
certain pieces of our core code. It basically works in the same way
as preempt_disable()/preempt_enable(): By marking a region under
which all allocations will be accounted to the root memcg.

We need this to prevent races in early cache creation, when we
allocate data using caches that are not necessarily created already.

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/sched.h |    1 +
 mm/memcontrol.c       |   25 +++++++++++++++++++++++++
 2 files changed, 26 insertions(+), 0 deletions(-)

diff --git a/include/linux/sched.h b/include/linux/sched.h
index 81a173c..0501114 100644
--- a/include/linux/sched.h
+++ b/include/linux/sched.h
@@ -1613,6 +1613,7 @@ struct task_struct {
   unsigned long nr_pages; /* uncharged usage */
   unsigned long memsw_nr_pages; /* uncharged mem+swap usage */
  } memcg_batch;
+ atomic_t memcg_kmem_skip_account;
 #endif
 #ifdef CONFIG_HAVE_HW_BREAKPOINT
  atomic_t ptrace_bp_refcnt;
diff --git a/mm/memcontrol.c b/mm/memcontrol.c
index 5a7416b..c4ecf9c 100644
--- a/mm/memcontrol.c
+++ b/mm/memcontrol.c
@@ -479,6 +479,21 @@ struct cg_proto *tcp_proto_cgroup(struct mem_cgroup *memcg)
 EXPORT_SYMBOL(tcp_proto_cgroup);
 #endif /* CONFIG_INET */

+static void memcg_stop_kmem_account(void)
+{
+ if (!current->mm)
+  return;
+
+ atomic_inc(&current->memcg_kmem_skip_account);
+}
+
+static void memcg_resume_kmem_account(void)
+{
+ if (!current->mm)
+  return;
+
+ atomic_dec(&current->memcg_kmem_skip_account);
+}
 char *mem_cgroup_cache_name(struct mem_cgroup *memcg, struct kmem_cache *cachep)
 {
  char *name;
@@ -540,7 +555,9 @@ static struct kmem_cache *memcg_create_kmem_cache(struct
mem_cgroup *memcg,
  if (new_cachep)
   goto out;

+ memcg_stop_kmem_account();
  new_cachep = kmem_cache_dup(memcg, cachep);
+ memcg_resume_kmem_account();

  if (new_cachep == NULL) {
   new_cachep = cachep;
@@ -631,7 +648,9 @@ static void memcg_create_cache_enqueue(struct mem_cgroup *memcg,
  if (!css_tryget(&memcg->css))
   return;

+ memcg_stop_kmem_account();
```

```
  cw = kmalloc(sizeof(struct create_work), GFP_NOWAIT);
+ memcg_resume_kmem_account();
  if (cw == NULL) {
   css_put(&memcg->css);
   return;
@@ -666,6 +685,9 @@ struct kmem_cache *__mem_cgroup_get_kmem_cache(struct
kmem_cache *cachep,
  int idx;
  struct task_struct *p;

+ if (!current->mm || atomic_read(&current->memcg_kmem_skip_account))
+  return cachep;
+
  gfp |=  cachep->allocflags;

  if (cachep->memcg_params.memcg)
@@ -700,6 +722,9 @@ bool __mem_cgroup_new_kmem_page(struct page *page, gfp_t gfp)
  if (!current->mm || in_interrupt())
   return true;

+ if (!current->mm || atomic_read(&current->memcg_kmem_skip_account))
+  return true;
+
  rcu_read_lock();
  p = rcu_dereference(current->mm->owner);
  memcg = mem_cgroup_from_task(p);
--
1.7.7.6
```

Subject: [PATCH v2 20/29] slub: charge allocation to a memcg
Posted by Glauber Costa on Fri, 11 May 2012 17:44:22 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 |   39 ++++++++++++++--
 mm/slub.c                |  113 +++++++++++++++++++++++++++++++++++++++++-----
 2 files changed, 135 insertions(+), 17 deletions(-)

diff --git a/include/linux/slub_def.h b/include/linux/slub_def.h
index 5f5e942..56b6fb4 100644
--- a/include/linux/slub_def.h
+++ b/include/linux/slub_def.h
@@ -13,6 +13,8 @@
 #include <linux/kobject.h>

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

 enum stat_item {
  ALLOC_FASTPATH,  /* Allocation from cpu slab */
@@ -210,27 +212,54 @@ 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);
 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;
+ struct page *page;
+
+ ret = kmalloc_order_base(size, flags, order);
+ if (!ret)
+  return ret;
+
+ page = virt_to_head_page(ret);
+
+ if (!mem_cgroup_new_kmem_page(page, flags)) {
+  put_page(page);
+  return NULL;
+ }
+
+ return ret;
+}
+
 /**
  * Calling this on allocated memory will check that the memory
  * is expected to be in use, and print warnings if not.
@@ -275,7 +304,7 @@ 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;
@@ -308,7 +337,7 @@ 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 9b21b38..c70db56 100644
--- a/mm/slub.c
+++ b/mm/slub.c
@@ -1284,11 +1284,39 @@ 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;
+}
+
+#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
+static void kmem_cache_inc_ref(struct kmem_cache *s)
+{
+ if (s->memcg_params.memcg)
+  atomic_inc(&s->memcg_params.refcnt);
+}
+static void kmem_cache_drop_ref(struct kmem_cache *s)
+{
+ if (s->memcg_params.memcg)
+  atomic_dec(&s->memcg_params.refcnt);
+}
+#else
+static inline void kmem_cache_inc_ref(struct kmem_cache *s)
+{
+}
+static inline void kmem_cache_drop_ref(struct kmem_cache *s)
+{
+}
+#endif
+
+
+
 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;

@@ -1297,13 +1325,29 @@ 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;
   /*
@@ -1314,13 +1358,25 @@ 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;
+ }
+
+ kmem_cache_inc_ref(s);

  if (kmemcheck_enabled
   && !(s->flags & (SLAB_NOTRACK | DEBUG_DEFAULT_FLAGS))) {
@@ -1420,6 +1476,9 @@ 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);
+ kmem_cache_drop_ref(s);
 }

 #define need_reserve_slab_rcu      \
@@ -2301,8 +2360,9 @@ 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;
@@ -2370,6 +2430,24 @@ 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_);
@@ -3197,8 +3275,10 @@ void kmem_cache_destroy(struct kmem_cache *s)
   list_del(&s->list);
 #ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
   /* Not a memcg cache */
-  if (s->memcg_params.id != -1)
+  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)) {
@@ -3372,10 +3452,18 @@ static void *kmalloc_large_node(size_t size, gfp_t flags, int node)
  void *ptr = NULL;

  flags |= __GFP_COMP | __GFP_NOTRACK;
+
+
  page = alloc_pages_node(node, flags, get_order(size));
- if (page)
+ if (!page)
+  goto out;
+
+ if (!mem_cgroup_new_kmem_page(page, flags))
+  put_page(page);
+ else
   ptr = page_address(page);
+out:
  kmemleak_alloc(ptr, size, 1, flags);
  return ptr;
 }
@@ -3477,6 +3565,7 @@ void kfree(const void *x)
  if (unlikely(!PageSlab(page))) {
   BUG_ON(!PageCompound(page));
   kmemleak_free(x);
+  mem_cgroup_free_kmem_page(page);
   put_page(page);
   return;
  }
--
1.7.7.6
```

Subject: [PATCH v2 21/29] slab: per-memcg accounting of slab caches
Posted by Glauber Costa on Fri, 11 May 2012 17:44:23 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 |   62 ++++++++++++++++++++++++++++
 mm/slab.c                |  102 +++++++++++++++++++++++++++++++++++++++++++++-----
 2 files changed, 154 insertions(+), 10 deletions(-)
```

```
diff --git a/include/linux/slab_def.h b/include/linux/slab_def.h
index 06e4a3e..ed8c43c 100644
--- a/include/linux/slab_def.h
+++ b/include/linux/slab_def.h
@@ -218,4 +218,66 @@ 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);
+}
+
+static inline void
+mem_cgroup_kmem_cache_finish_sleep(struct kmem_cache *cachep)
+{
+ 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 985714a..7022f86 100644
--- a/mm/slab.c
+++ b/mm/slab.c
@@ -1821,20 +1821,28 @@ 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);

@@ -1850,6 +1858,14 @@ static void *kmem_getpages(struct kmem_cache *cachep, gfp_t flags, int nodeid)
  return page_address(page);
 }

+#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
+void kmem_cache_drop_ref(struct kmem_cache *cachep)
+{
+ if (cachep->memcg_params.id == -1)
+  atomic_dec(&cachep->memcg_params.refcnt);
+}
+#endif /* CONFIG_CGROUP_MEM_RES_CTLR_KMEM */
+
 /*
  * Interface to system's page release.
  */
@@ -1867,6 +1883,8 @@ 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);
@@ -2854,10 +2872,11 @@ void kmem_cache_destroy(struct kmem_cache *cachep)

 #ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
  /* Not a memcg cache */
- if (cachep->memcg_params.id != -1)
+ 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();
@@ -3063,8 +3082,10 @@ 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
@@ -3093,8 +3114,10 @@ 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);

@@ -3107,8 +3130,10 @@ 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;
 }

@@ -3869,11 +3894,15 @@ 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);
@@ -3884,6 +3913,10 @@ 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,
@@ -3896,13 +3929,17 @@ 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);
@@ -3915,6 +3952,9 @@ 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,
@@ -4023,9 +4063,33 @@ 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);
@@ -4053,9 +4117,19 @@ 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);
@@ -4324,6 +4398,13 @@ 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
@@ -4356,6 +4437,7 @@ 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 v2 22/29] memcg: disable kmem code when not in use.
Posted by Glauber Costa on Fri, 11 May 2012 17:44:24 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            |   19 ++++++++++++++++++-
 2 files changed, 21 insertions(+), 2 deletions(-)

diff --git a/include/linux/memcontrol.h b/include/linux/memcontrol.h
index c555799..4000798 100644
--- a/include/linux/memcontrol.h
+++ b/include/linux/memcontrol.h
@@ -22,6 +22,7 @@
 #include <linux/cgroup.h>
 #include <linux/vm_event_item.h>
 #include <linux/hardirq.h>
+#include <linux/jump_label.h>

 struct mem_cgroup;
 struct page_cgroup;
@@ -460,7 +461,8 @@ void __mem_cgroup_free_kmem_page(struct page *page);
 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)
 #else
 static inline void mem_cgroup_register_cache(struct mem_cgroup *memcg,
 	     struct kmem_cache *s)
diff --git a/mm/memcontrol.c b/mm/memcontrol.c
index c4ecf9c..ad60648 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)
 {
@@ -840,6 +850,10 @@ static void memcg_slab_init(struct mem_cgroup *memcg)
  for (i = 0; i < MAX_KMEM_CACHE_TYPES; i++)
   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);
@@ -4359,8 +4373,10 @@ 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)
+    if (!memcg->kmem_accounted && val != RESOURCE_MAX) {
+     static_key_slow_inc(&mem_cgroup_kmem_enabled_key);
     memcg->kmem_accounted = true;
+    }
   }
 #endif
   else
@@ -5294,6 +5310,7 @@ static void free_work(struct work_struct *work)
  int size = sizeof(struct mem_cgroup);

  memcg = container_of(work, struct mem_cgroup, work_freeing);
+ disarm_static_keys(memcg);
  if (size < PAGE_SIZE)
   kfree(memcg);
  else
```

```
--
1.7.7.6
```

Subject: [PATCH v2 23/29] memcg: destroy memcg caches
Posted by Glauber Costa on Fri, 11 May 2012 17:44:25 GMT

View Forum Message <> Reply to Message

This patch implements destruction of memcg caches. Right now,
only caches where our reference counter is the last remaining are
deleted. If there are any other reference counters around, we just
leave the caches lying around until they go away.

When that happen, a destruction function is called from the cache
code. Caches are only destroyed in process context, so we queue them
up for later processing in the general case.

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 |    2 +
 include/linux/slab.h       |    1 +
 mm/memcontrol.c            |   91 ++++++++++++++++++++++++++++++++++++++++++++-
 mm/slab.c                  |    5 +-
 mm/slub.c                  |    7 ++-
 5 files changed, 101 insertions(+), 5 deletions(-)

diff --git a/include/linux/memcontrol.h b/include/linux/memcontrol.h
index 4000798..3e03f26 100644
--- a/include/linux/memcontrol.h
+++ b/include/linux/memcontrol.h
@@ -463,6 +463,8 @@ __mem_cgroup_get_kmem_cache(struct kmem_cache *cachep, gfp_t gfp);

 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
 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 e73ef71..a03a4f2 100644
```

```
--- a/include/linux/slab.h
+++ b/include/linux/slab.h
@@ -164,6 +164,7 @@ struct mem_cgroup_cache_params {
  size_t orig_align;

 #endif
+ struct list_head destroyed_list; /* Used when deleting memcg cache */
 };
 #endif

diff --git a/mm/memcontrol.c b/mm/memcontrol.c
index ad60648..1d1a307 100644
--- a/mm/memcontrol.c
+++ b/mm/memcontrol.c
@@ -476,6 +476,11 @@ static void disarm_static_keys(struct mem_cgroup *memcg)
 {
  if (memcg->kmem_accounted)
   static_key_slow_dec(&mem_cgroup_kmem_enabled_key);
+ /*
+  * This check can't live in kmem destruction function,
+  * since the charges will outlive the cgroup
+  */
+ BUG_ON(res_counter_read_u64(&memcg->kmem, RES_USAGE) != 0);
 }

 #ifdef CONFIG_INET
@@ -540,6 +545,8 @@ void mem_cgroup_register_cache(struct mem_cgroup *memcg,
  if (!memcg)
   id = ida_simple_get(&cache_types, 0, MAX_KMEM_CACHE_TYPES,
         GFP_KERNEL);
+ else
+  INIT_LIST_HEAD(&cachep->memcg_params.destroyed_list);
  cachep->memcg_params.id = id;
 }

@@ -592,6 +599,53 @@ struct create_work {
 /* 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;
+ struct mem_cgroup_cache_params *p, *tmp;
+ unsigned long flags;
+ LIST_HEAD(del_unlocked);
+
+ spin_lock_irqsave(&cache_queue_lock, flags);
+ list_for_each_entry_safe(p, tmp, &destroyed_caches, destroyed_list) {
+  cachep = container_of(p, struct kmem_cache, memcg_params);
+  list_move(&cachep->memcg_params.destroyed_list, &del_unlocked);
+ }
+ spin_unlock_irqrestore(&cache_queue_lock, flags);
+
+ list_for_each_entry_safe(p, tmp, &del_unlocked, destroyed_list) {
+  cachep = container_of(p, struct kmem_cache, memcg_params);
+  list_del(&cachep->memcg_params.destroyed_list);
+  if (!atomic_read(&cachep->memcg_params.refcnt)) {
+   mem_cgroup_put(cachep->memcg_params.memcg);
+   kmem_cache_destroy(cachep);
+  }
+ }
+}
+static DECLARE_WORK(kmem_cache_destroy_work, kmem_cache_destroy_work_func);
+
+static void __mem_cgroup_destroy_cache(struct kmem_cache *cachep)
+{
+ BUG_ON(cachep->memcg_params.id != -1);
+ list_add(&cachep->memcg_params.destroyed_list, &destroyed_caches);
+}
+
+void mem_cgroup_destroy_cache(struct kmem_cache *cachep)
+{
+ unsigned long flags;
+
+ /*
+  * 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);
+ __mem_cgroup_destroy_cache(cachep);
+ 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.
@@ -613,6 +667,33 @@ void mem_cgroup_flush_cache_create_queue(void)
  spin_unlock_irqrestore(&cache_queue_lock, flags);
 }

+static void mem_cgroup_destroy_all_caches(struct mem_cgroup *memcg)
+{
+ struct kmem_cache *cachep;
+ unsigned long flags;
+ int i;
+
+ /*
+  * pre_destroy() gets called with no tasks in the cgroup.
+  * this means that after flushing the create queue, no more caches
+  * will appear
+  */
+ mem_cgroup_flush_cache_create_queue();
+
+ spin_lock_irqsave(&cache_queue_lock, flags);
+ for (i = 0; i < MAX_KMEM_CACHE_TYPES; i++) {
+  cachep = memcg->slabs[i];
+  if (!cachep)
+   continue;
+
+  if (atomic_dec_and_test(&cachep->memcg_params.refcnt))
+   __mem_cgroup_destroy_cache(cachep);
+ }
+ spin_unlock_irqrestore(&cache_queue_lock, flags);
+
+ schedule_work(&kmem_cache_destroy_work);
+}
+
 static void memcg_create_cache_work_func(struct work_struct *w)
 {
  struct create_work *cw, *tmp;
@@ -854,6 +935,10 @@ static void memcg_slab_init(struct mem_cgroup *memcg)
 static inline void disarm_static_keys(struct mem_cgroup *memcg)
 {
 }
+
+static inline void mem_cgroup_destroy_all_caches(struct mem_cgroup *memcg)
+{
+}
 #endif /* CONFIG_CGROUP_MEM_RES_CTLR_KMEM */

 static void drain_all_stock_async(struct mem_cgroup *memcg);
@@ -4133,6 +4218,7 @@ static int mem_cgroup_force_empty(struct mem_cgroup *memcg, bool free_all)
  int node, zid, shrink;
  int nr_retries = MEM_CGROUP_RECLAIM_RETRIES;
  struct cgroup *cgrp = memcg->css.cgroup;
+ u64 usage;

  css_get(&memcg->css);

@@ -4172,8 +4258,10 @@ move_account:
   if (ret == -ENOMEM)
    goto try_to_free;
   cond_resched();
+  usage = res_counter_read_u64(&memcg->res, RES_USAGE) -
+   res_counter_read_u64(&memcg->kmem, RES_USAGE);
  /* "ret" should also be checked to ensure all lists are empty. */
- } while (res_counter_read_u64(&memcg->res, RES_USAGE) > 0 || ret);
+ } while (usage > 0 || ret);
 out:
  css_put(&memcg->css);
  return ret;
@@ -5518,6 +5606,7 @@ static int mem_cgroup_pre_destroy(struct cgroup *cont)
 {
  struct mem_cgroup *memcg = mem_cgroup_from_cont(cont);

+ mem_cgroup_destroy_all_caches(memcg);
  return mem_cgroup_force_empty(memcg, false);
 }

diff --git a/mm/slab.c b/mm/slab.c
index 7022f86..a6fd82e 100644
--- a/mm/slab.c
+++ b/mm/slab.c
@@ -1861,8 +1861,9 @@ static void *kmem_getpages(struct kmem_cache *cachep, gfp_t flags,
int nodeid)
 #ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
 void kmem_cache_drop_ref(struct kmem_cache *cachep)
 {
- if (cachep->memcg_params.id == -1)
-  atomic_dec(&cachep->memcg_params.refcnt);
+ 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 */

diff --git a/mm/slub.c b/mm/slub.c
index c70db56..02d8f5e 100644
--- a/mm/slub.c
+++ b/mm/slub.c
@@ -1297,8 +1297,11 @@ static void kmem_cache_inc_ref(struct kmem_cache *s)
 }
 static void kmem_cache_drop_ref(struct kmem_cache *s)
 {
- if (s->memcg_params.memcg)
-  atomic_dec(&s->memcg_params.refcnt);
+ if (!s->memcg_params.memcg)
+  return;
```

```
+
+ if (unlikely(atomic_dec_and_test(&s->memcg_params.refcnt)))
+  mem_cgroup_destroy_cache(s);
 }
 #else
 static inline void kmem_cache_inc_ref(struct kmem_cache *s)
--
1.7.7.6
```

Subject: [PATCH v2 24/29] memcg/slub: shrink dead caches
Posted by Glauber Costa on Fri, 11 May 2012 17:44:26 GMT

View Forum Message <> Reply to Message

In the slub allocator, when the last object of a page goes away, we
don't necessarily free it - there is not necessarily a test for empty
page in any slab_free path.

This means that when we destroy a memcg cache that happened to be empty,
those caches may take a lot of time to go away: removing the memcg
reference won't destroy them - because there are pending references,
and the empty pages will stay there, until a shrinker is called upon
for any reason.

This patch marks all memcg caches as dead. kmem_cache_shrink is called
for the ones who are not yet dead - this will force internal cache
reorganization, and then all references to empty pages will be removed.

An unlikely branch is used to make sure this case does not affect
performance in the usual slab_free path.

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     |    4 +++
 include/linux/slub_def.h |    9 ++++++++
 mm/memcontrol.c          |   49 ++++++++++++++++++++++++++++++++++++++++++++++++--
 mm/slub.c                |    1 +
 4 files changed, 60 insertions(+), 3 deletions(-)

diff --git a/include/linux/slab.h b/include/linux/slab.h
index a03a4f2..d03637e 100644
--- a/include/linux/slab.h
```

```
+++ b/include/linux/slab.h
@@ -154,10 +154,14 @@ unsigned int kmem_cache_size(struct kmem_cache *);
 #endif

 #ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
+#include <linux/workqueue.h>
+
 struct mem_cgroup_cache_params {
  struct mem_cgroup *memcg;
  int id;
  atomic_t refcnt;
+ bool dead;
+ struct work_struct cache_shrinker;

 #ifdef CONFIG_SLAB
  /* Original cache parameters, used when creating a memcg cache */
diff --git a/include/linux/slub_def.h b/include/linux/slub_def.h
index 56b6fb4..7462b2e 100644
--- a/include/linux/slub_def.h
+++ b/include/linux/slub_def.h
@@ -117,6 +117,15 @@ struct kmem_cache {
  struct kmem_cache_node *node[MAX_NUMNODES];
 };

+
+static inline void kmem_cache_verify_dead(struct kmem_cache *cachep)
+{
+#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
+ if (unlikely(cachep->memcg_params.dead))
+  schedule_work(&cachep->memcg_params.cache_shrinker);
+#endif
+}
+
 /*
  * Kmalloc subsystem.
  */
diff --git a/mm/memcontrol.c b/mm/memcontrol.c
index 1d1a307..c3772dc 100644
--- a/mm/memcontrol.c
+++ b/mm/memcontrol.c
@@ -520,7 +520,7 @@ char *mem_cgroup_cache_name(struct mem_cgroup *memcg, struct kmem_cache *cachep)

  BUG_ON(dentry == NULL);

- name = kasprintf(GFP_KERNEL, "%s(%d:%s)",
+ name = kasprintf(GFP_KERNEL, "%s(%d:%s)dead",
      cachep->name, css_id(&memcg->css), dentry->d_name.name);
  return name;
@@ -557,11 +557,24 @@ void mem_cgroup_release_cache(struct kmem_cache *cachep)
 }


+static void cache_shrinker_work_func(struct work_struct *work)
+{
+ struct mem_cgroup_cache_params *params;
+ struct kmem_cache *cachep;
+
+ params = container_of(work, struct mem_cgroup_cache_params,
+        cache_shrinker);
+ cachep = container_of(params, struct kmem_cache, memcg_params);
+
+ kmem_cache_shrink(cachep);
+}
+
 static struct kmem_cache *memcg_create_kmem_cache(struct mem_cgroup *memcg,
         struct kmem_cache *cachep)
 {
  struct kmem_cache *new_cachep;
  int idx;
+ char *name;

  BUG_ON(!mem_cgroup_kmem_enabled(memcg));

@@ -581,10 +594,21 @@ static struct kmem_cache *memcg_create_kmem_cache(struct
mem_cgroup *memcg,
   goto out;
  }

+ /*
+  * Because the cache is expected to duplicate the string,
+  * we must make sure it has opportunity to copy its full
+  * name. Only now we can remove the dead part from it
+  */
+ name = (char *)new_cachep->name;
+ if (name)
+  name[strlen(name) - 4] = '\0';
+
  mem_cgroup_get(memcg);
  memcg->slabs[idx] = new_cachep;
  new_cachep->memcg_params.memcg = memcg;
  atomic_set(&new_cachep->memcg_params.refcnt, 1);
+ INIT_WORK(&new_cachep->memcg_params.cache_shrinker,
+    cache_shrinker_work_func);
 out:
  mutex_unlock(&memcg_cache_mutex);
  return new_cachep;
@@ -607,6 +631,21 @@ static void kmem_cache_destroy_work_func(struct work_struct *w)
  struct mem_cgroup_cache_params *p, *tmp;
  unsigned long flags;
  LIST_HEAD(del_unlocked);
+ LIST_HEAD(shrinkers);
+
+ spin_lock_irqsave(&cache_queue_lock, flags);
+ list_for_each_entry_safe(p, tmp, &destroyed_caches, destroyed_list) {
+  cachep = container_of(p, struct kmem_cache, memcg_params);
+  if (atomic_read(&cachep->memcg_params.refcnt) != 0)
+   list_move(&cachep->memcg_params.destroyed_list, &shrinkers);
+ }
+ spin_unlock_irqrestore(&cache_queue_lock, flags);
+
+ list_for_each_entry_safe(p, tmp, &shrinkers, destroyed_list) {
+  cachep = container_of(p, struct kmem_cache, memcg_params);
+  list_del(&cachep->memcg_params.destroyed_list);
+  kmem_cache_shrink(cachep);
+ }

  spin_lock_irqsave(&cache_queue_lock, flags);
  list_for_each_entry_safe(p, tmp, &destroyed_caches, destroyed_list) {
@@ -682,12 +721,16 @@ static void mem_cgroup_destroy_all_caches(struct mem_cgroup
*memcg)

  spin_lock_irqsave(&cache_queue_lock, flags);
  for (i = 0; i < MAX_KMEM_CACHE_TYPES; i++) {
+  char *name;
   cachep = memcg->slabs[i];
   if (!cachep)
    continue;

-  if (atomic_dec_and_test(&cachep->memcg_params.refcnt))
-   __mem_cgroup_destroy_cache(cachep);
+  atomic_dec(&cachep->memcg_params.refcnt);
+  cachep->memcg_params.dead = true;
+  name = (char *)cachep->name;
+  name[strlen(name)] = 'd';
+  __mem_cgroup_destroy_cache(cachep);
  }
  spin_unlock_irqrestore(&cache_queue_lock, flags);

diff --git a/mm/slub.c b/mm/slub.c
index 02d8f5e..f077b90 100644
--- a/mm/slub.c
+++ b/mm/slub.c
@@ -2675,6 +2675,7 @@ redo:
  } else
   __slab_free(s, page, x, addr);

+ kmem_cache_verify_dead(s);
 }

 void kmem_cache_free(struct kmem_cache *s, void *x)
--
1.7.7.6
```

Subject: [PATCH v2 25/29] memcg: Track all the memcg children of a kmem_cache.

Posted by Glauber Costa on Fri, 11 May 2012 17:44:27 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/memcontrol.h |    1 +
 include/linux/slab.h       |    1 +
 mm/memcontrol.c            |    9 +++++++
 mm/slab.c                  |   52 +++++++++++++++++++++++++++++++++++++++++++---
 4 files changed, 59 insertions(+), 4 deletions(-)

diff --git a/include/linux/memcontrol.h b/include/linux/memcontrol.h
index 3e03f26..5a5cd42 100644
--- a/include/linux/memcontrol.h
+++ b/include/linux/memcontrol.h
@@ -465,6 +465,7 @@ 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);
+void mem_cgroup_remove_child_kmem_cache(struct kmem_cache *cachep, int id);
 #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 d03637e..876783b 100644
--- a/include/linux/slab.h
+++ b/include/linux/slab.h
@@ -169,6 +169,7 @@ struct mem_cgroup_cache_params {

 #endif
  struct list_head destroyed_list; /* Used when deleting memcg cache */
+ struct list_head sibling_list;
 };
 #endif

diff --git a/mm/memcontrol.c b/mm/memcontrol.c
index c3772dc..933edf1 100644
--- a/mm/memcontrol.c
+++ b/mm/memcontrol.c
@@ -548,6 +548,7 @@ void mem_cgroup_register_cache(struct mem_cgroup *memcg,
  else
   INIT_LIST_HEAD(&cachep->memcg_params.destroyed_list);
  cachep->memcg_params.id = id;
+ INIT_LIST_HEAD(&cachep->memcg_params.sibling_list);
 }

 void mem_cgroup_release_cache(struct kmem_cache *cachep)
@@ -845,6 +846,14 @@ struct kmem_cache *__mem_cgroup_get_kmem_cache(struct
kmem_cache *cachep,
 }
 EXPORT_SYMBOL(__mem_cgroup_get_kmem_cache);

+void mem_cgroup_remove_child_kmem_cache(struct kmem_cache *cachep, int id)
+{
+ mutex_lock(&memcg_cache_mutex);
+ cachep->memcg_params.memcg->slabs[id] = NULL;
+ mutex_unlock(&memcg_cache_mutex);
+ mem_cgroup_put(cachep->memcg_params.memcg);
+}
+
 bool __mem_cgroup_new_kmem_page(struct page *page, gfp_t gfp)
 {
  struct mem_cgroup *memcg;
diff --git a/mm/slab.c b/mm/slab.c
index a6fd82e..cd4600d 100644
--- a/mm/slab.c
+++ b/mm/slab.c
@@ -2638,6 +2638,8 @@ struct kmem_cache *kmem_cache_dup(struct mem_cgroup *memcg,
  if (new == NULL)
   goto out;
```

```
+ 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))
@@ -2853,6 +2855,29 @@ 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;
+  int id = cachep->memcg_params.id;
+
+  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, 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);
@@ -2860,6 +2885,9 @@ 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);
@@ -4650,11 +4678,27 @@ 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 v2 26/29] memcg: Per-memcg memory.kmem.slabinfo file.
Posted by Glauber Costa on Fri, 11 May 2012 17:44:28 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            |   87 ++++++++++++++++++++++++++++++++++++-------------
 mm/slub.c            |    5 +++
 4 files changed, 87 insertions(+), 23 deletions(-)

diff --git a/include/linux/slab.h b/include/linux/slab.h
index 876783b..e250111 100644
--- a/include/linux/slab.h
+++ b/include/linux/slab.h
@@ -330,6 +330,7 @@ extern void *__kmalloc_track_caller(size_t, gfp_t, unsigned long);
 #define MAX_KMEM_CACHE_TYPES 400
 extern struct kmem_cache *kmem_cache_dup(struct mem_cgroup *memcg,
       struct kmem_cache *cachep);
+extern 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 933edf1..6b49b5e 100644
--- a/mm/memcontrol.c
+++ b/mm/memcontrol.c
@@ -5223,6 +5223,19 @@ 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",
@@ -5247,6 +5260,10 @@ 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 cd4600d..afc26df 100644
--- a/mm/slab.c
+++ b/mm/slab.c
@@ -4523,21 +4523,26 @@ 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)
@@ -4549,31 +4554,43 @@ 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;
@@ -4581,12 +4598,12 @@ 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;
@@ -4620,6 +4637,30 @@ 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 f077b90..0efcd77 100644
--- a/mm/slub.c
+++ b/mm/slub.c
@@ -4156,6 +4156,11 @@ struct kmem_cache *kmem_cache_dup(struct mem_cgroup *memcg,
  kfree(name);
  return new;
 }
+
+int mem_cgroup_slabinfo(struct mem_cgroup *memcg, struct seq_file *m)
+{
+ return 0;
+}
 #endif

 #ifdef CONFIG_SMP
--
1.7.7.6
```

Subject: [PATCH v2 27/29] slub: create slabinfo file for memcg
Posted by Glauber Costa on Fri, 11 May 2012 17:44:29 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 0efcd77..afe29ef 100644
--- a/mm/slub.c
+++ b/mm/slub.c
@@ -4159,6 +4159,33 @@ struct kmem_cache *kmem_cache_dup(struct mem_cgroup *memcg,

 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: [PATCH v2 28/29] slub: track all children of a kmem cache
Posted by Glauber Costa on Fri, 11 May 2012 17:44:30 GMT

View Forum Message <> Reply to Message

When we destroy a cache (like for instance, if we're unloading a module)
we need to go through the list of memcg caches and destroy them as well.

The caches are expected to be empty by themselves, so nothing is changed
here. All previous guarantees are kept and no new guarantees are given.

So given all memcg caches are expected to be empty - even though they are
likely to be hanging around in the system, we just need to scan a list of
sibling caches, and destroy each one of them.

This is very similar to the work done by Suleiman for the slab.

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 |   61 +++++++++++++++++++++++++++++++++++++++++++++++------------ -
 1 files changed, 47 insertions(+), 14 deletions(-)

diff --git a/mm/slub.c b/mm/slub.c
index afe29ef..cfa6295 100644
--- a/mm/slub.c
+++ b/mm/slub.c
@@ -3267,6 +3267,20 @@ static inline int kmem_cache_close(struct kmem_cache *s)
  return 0;
 }

+void kmem_cache_destroy_unlocked(struct kmem_cache *s)
+{
+ mem_cgroup_release_cache(s);
+ if (kmem_cache_close(s)) {
+  printk(KERN_ERR "SLUB %s: %s called for cache that "
+   "still has objects.\n", s->name, __func__);
+  dump_stack();
+ }
+
+ if (s->flags & SLAB_DESTROY_BY_RCU)
+  rcu_barrier();
+ sysfs_slab_remove(s);
+}
+
 /*
  * Close a cache and release the kmem_cache structure
  * (must be used for caches created using kmem_cache_create)
@@ -3275,24 +3289,41 @@ void kmem_cache_destroy(struct kmem_cache *s)
 {
  down_write(&slub_lock);
  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();
```

```
+ /* Not a memcg cache */
+ if (s->memcg_params.id != -1) {
+  struct mem_cgroup_cache_params *p, *tmp, *this;
+  struct kmem_cache *c;
+  int id = s->memcg_params.id;
+
+  this = &s->memcg_params;
+  mem_cgroup_flush_cache_create_queue();
+  list_for_each_entry_safe(p, tmp, &this->sibling_list, sibling_list) {
+   c = container_of(p, struct kmem_cache, memcg_params);
+   /* We never added the main cache to the sibling list */
+   if (WARN_ON(c == s))
+    continue;
+
+   c->refcount--;
+   if (c->refcount)
+    continue;
+
+   list_del(&c->list);
+   list_del(&c->memcg_params.sibling_list);
+   s->refcount--; /* parent reference */
+   up_write(&slub_lock);
+   mem_cgroup_remove_child_kmem_cache(c, id);
+   kmem_cache_destroy_unlocked(c);
+   down_write(&slub_lock);
   }
+ }
 #endif
+
+ if (!s->refcount) {
+  list_del(&s->list);
   up_write(&slub_lock);
-  if (kmem_cache_close(s)) {
-   printk(KERN_ERR "SLUB %s: %s called for cache that "
-    "still has objects.\n", s->name, __func__);
-   dump_stack();
-  }
-  if (s->flags & SLAB_DESTROY_BY_RCU)
-   rcu_barrier();
-  sysfs_slab_remove(s);
+  kmem_cache_destroy_unlocked(s);
  } else
   up_write(&slub_lock);
 }
@@ -4150,6 +4181,8 @@ struct kmem_cache *kmem_cache_dup(struct mem_cgroup *memcg,
   */
  if (new) {
   down_write(&slub_lock);
+  list_add(&new->memcg_params.sibling_list,
+    &s->memcg_params.sibling_list);
   s->refcount++;
   up_write(&slub_lock);
  }
--
1.7.7.6
```

Subject: [PATCH v2 29/29] Documentation: add documentation for slab tracker for memcg

Posted by Glauber Costa on Fri, 11 May 2012 17:44:31 GMT

View Forum Message <> Reply to Message

In a separate patch, to aid reviewers.

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>
CC: Randy Dunlap <rdunlap@xenotime.net>

```
---
 Documentation/cgroups/memory.txt |   33 +++++++++++++++++++++++++++++++++
 1 files changed, 33 insertions(+), 0 deletions(-)

diff --git a/Documentation/cgroups/memory.txt b/Documentation/cgroups/memory.txt
index 4c95c00..9accaa1 100644
--- a/Documentation/cgroups/memory.txt
+++ b/Documentation/cgroups/memory.txt
@@ -75,6 +75,12 @@ Brief summary of control files.
  memory.kmem.tcp.limit_in_bytes  # set/show hard limit for tcp buf memory
  memory.kmem.tcp.usage_in_bytes  # show current tcp buf memory allocation

+ memory.kmem.limit_in_bytes  # set/show hard limit for general kmem memory
+ memory.kmem.usage_in_bytes  # show current general kmem memory allocation
+ memory.kmem.failcnt   # show current number of kmem limit hits
+ memory.kmem.max_usage_in_bytes  # show max kmem usage
+ memory.kmem.slabinfo   # show cgroup-specific slab usage information
+
 1. History

 The memory controller has a long history. A request for comments for the memory
@@ -271,6 +277,14 @@ cgroup may or may not be accounted.
 Currently no soft limit is implemented for kernel memory. It is future work
 to trigger slab reclaim when those limits are reached.
```

```
+Kernel memory is not accounted until it is limited. Users that want to just
+track kernel memory usage can set the limit value to a big enough value so
+the limit is guaranteed to never hit. A kernel memory limit bigger than the
+current memory limit will have this effect as well.
+
+This guarantes that this extension is backwards compatible to any previous
+memory cgroup version.
+
 2.7.1 Current Kernel Memory resources accounted

 * sockets memory pressure: some sockets protocols have memory pressure
@@ -279,6 +293,24 @@ per cgroup, instead of globally.

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

+* slab/kmalloc:
+
+When slab memory is tracked (memory.kmem.limit_in_bytes != -1ULL), both
+memory.kmem.usage_in_bytes and memory.usage_in_bytes are updated. When
+memory.kmem.limit_in_bytes is left alone, no tracking of slab caches takes
+place.
+
+Because a slab page is shared among many tasks, it is not possible to take
+any meaningful action upon task migration. Slabs created in a cgroup stay
+around until the cgroup is destructed. Information about the slabs used
+by the cgroup is displayed in the cgroup file memory.kmem.slabinfo. The format
+of this file is and should remain compatible with /proc/slabinfo.
+
+Upon cgroup destruction, slabs that holds no live references are destructed.
+Workers are fired to destroy the remaining caches as they objects are freed.
+
+Memory used by dead caches are shown in the proc file /proc/dead_slabinfo
+
 3. User Interface

 0. Configuration
@@ -287,6 +319,7 @@ a. Enable CONFIG_CGROUPS
 b. Enable CONFIG_RESOURCE_COUNTERS
 c. Enable CONFIG_CGROUP_MEM_RES_CTLR
 d. Enable CONFIG_CGROUP_MEM_RES_CTLR_SWAP (to use swap extension)
+d. Enable CONFIG_CGROUP_MEM_RES_CTLR_KMEM (to use experimental kmem extension)

 1. Prepare the cgroups (see cgroups.txt, Why are cgroups needed?)
 # mount -t tmpfs none /sys/fs/cgroup
--
1.7.7.6
```

Subject: Re: [PATCH v2 02/29] slub: fix slab_state for slub
Posted by Christoph Lameter on Fri, 11 May 2012 17:51:10 GMT
View Forum Message <> Reply to Message

Acked-by: Christoph Lameter <cl@linux.com>

Subject: Re: [PATCH v2 04/29] slub: always get the cache from its page in kfree
Posted by Christoph Lameter on Fri, 11 May 2012 17:53:31 GMT
View Forum Message <> Reply to Message

On Fri, 11 May 2012, Glauber Costa wrote:

> 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

Other allocators may not have that information and this patch may
cause bugs to go unnoticed if the caller specifies the wrong slab cache.

Adding a VM_BUG_ON may be useful to make sure that kmem_cache_free is
always passed the correct slab cache.

Subject: Re: [PATCH v2 05/29] slab: rename gfpflags to allocflags
Posted by Christoph Lameter on Fri, 11 May 2012 17:54:03 GMT
View Forum Message <> Reply to Message

On Fri, 11 May 2012, Glauber Costa wrote:

> 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.

Acked-by: Christoph Lameter <cl@linux.com>

Subject: Re: [PATCH v2 04/29] slub: always get the cache from its page in kfree
Posted by Glauber Costa on Fri, 11 May 2012 17:57:49 GMT
View Forum Message <> Reply to Message

On 05/11/2012 02:53 PM, Christoph Lameter wrote:
> On Fri, 11 May 2012, Glauber Costa wrote:
>
>> 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

>
> Other allocators may not have that information and this patch may
> cause bugs to go unnoticed if the caller specifies the wrong slab cache.
>
> Adding a VM_BUG_ON may be useful to make sure that kmem_cache_free is
> always passed the correct slab cache.

Well, problem is , it isn't always passed the "correct" slab cache.
At least not after this series, since we'll have child caches associated
with the main cache.

So we'll pass, for instance, kmem_cache_free(dentry_cache...), but will
in fact free from the memcg copy of the dentry cache.

We can, of course, verify if the cache at least belongs to the same
"family". But that's quite expensive.

Subject: Re: [PATCH v2 00/29] kmem limitation for memcg
Posted by Glauber Costa on Fri, 11 May 2012 18:05:21 GMT
View Forum Message <> Reply to Message

On 05/11/2012 02:44 PM, Glauber Costa wrote:
> Hello All,
>
> This is my new take for the memcg kmem accounting.
> At this point, I consider the series pretty mature - although of course,
> bugs are always there...
>
> As a disclaimer, however, I must say that the slub code is much more stressed
> by me, since I know it better. If you have no more objections to the concepts
> presented, the remaining edges can probably be polished in a rc cycle,
> at the maintainers discretion, of course.
>
> Otherwise, I'll be happy to address any concerns of yours.
>
> Since last submission:
>
>   * memcgs can be properly removed.
>   * We are not charging based on current->mm->owner instead of current
>   * kmem_large allocations for slub got some fixes, specially for the free case
>   * A cache that is registered can be properly removed (common module case)
>   even if it spans memcg children. Slab had some code for that, now it works
>   well with both
>   * A new mechanism for skipping allocations is proposed (patch posted
>   separately already). Now instead of having kmalloc_no_account, we mark
>   a region as non-accountable for memcg.
>

Forgot to mention the ida-based index allocation, instead of keeping our
own bitmap.

Subject: Re: [PATCH v2 04/29] slub: always get the cache from its page in kfree
Posted by Christoph Lameter on Fri, 11 May 2012 18:06:36 GMT
View Forum Message <> Reply to Message

On Fri, 11 May 2012, Glauber Costa wrote:

> > Adding a VM_BUG_ON may be useful to make sure that kmem_cache_free is
> > always passed the correct slab cache.
>
> Well, problem is , it isn't always passed the "correct" slab cache.
> At least not after this series, since we'll have child caches associated with
> the main cache.
>
> So we'll pass, for instance, kmem_cache_free(dentry_cache...), but will in
> fact free from the memcg copy of the dentry cache.

Urg. But then please only do this for the MEMCG case and add a fat big
warning in kmem_cache_free.

Subject: Re: [PATCH v2 04/29] slub: always get the cache from its page in kfree
Posted by Glauber Costa on Fri, 11 May 2012 18:11:56 GMT
View Forum Message <> Reply to Message

On 05/11/2012 03:06 PM, Christoph Lameter wrote:
> On Fri, 11 May 2012, Glauber Costa wrote:
>
>>> Adding a VM_BUG_ON may be useful to make sure that kmem_cache_free is
>>> always passed the correct slab cache.
>>
>> Well, problem is , it isn't always passed the "correct" slab cache.
>> At least not after this series, since we'll have child caches associated with
>> the main cache.
>>
>> So we'll pass, for instance, kmem_cache_free(dentry_cache...), but will in
>> fact free from the memcg copy of the dentry cache.
>
> Urg. But then please only do this for the MEMCG case and add a fat big
> warning in kmem_cache_free.

I can do that, of course.
Another option if you don't oppose, is to add another field in the
kmem_cache structure (I tried to keep them at a minimum),

to record the parent cache we got created from.

Then, it gets trivial to do the following:

VM_BUG_ON(page->slab != s && page->slab != s->parent_cache);

Subject: Re: [PATCH v2 04/29] slub: always get the cache from its page in kfree
Posted by Christoph Lameter on Fri, 11 May 2012 18:17:41 GMT

View Forum Message <> Reply to Message

On Fri, 11 May 2012, Glauber Costa wrote:

> On 05/11/2012 03:06 PM, Christoph Lameter wrote:
> > On Fri, 11 May 2012, Glauber Costa wrote:
> >
> > > > Adding a VM_BUG_ON may be useful to make sure that kmem_cache_free is
> > > > always passed the correct slab cache.
> > >
> > > Well, problem is , it isn't always passed the "correct" slab cache.
> > > At least not after this series, since we'll have child caches associated
> > > with
> > > the main cache.
> > >
> > > So we'll pass, for instance, kmem_cache_free(dentry_cache...), but will in
> > > fact free from the memcg copy of the dentry cache.
> >
> > Urg. But then please only do this for the MEMCG case and add a fat big
> > warning in kmem_cache_free.
>
> I can do that, of course.
> Another option if you don't oppose, is to add another field in the kmem_cache
> structure (I tried to keep them at a minimum),
> to record the parent cache we got created from.
>
> Then, it gets trivial to do the following:
>
> VM_BUG_ON(page->slab != s && page->slab != s->parent_cache);

Sounds ok but I need to catch up on what this whole memcg thing in slab
allocators should accomplish in order to say something definite.

Subject: Re: [PATCH v2 04/29] slub: always get the cache from its page in kfree
Posted by Glauber Costa on Fri, 11 May 2012 18:20:10 GMT

View Forum Message <> Reply to Message

On 05/11/2012 03:17 PM, Christoph Lameter wrote:
> On Fri, 11 May 2012, Glauber Costa wrote:
>
>> On 05/11/2012 03:06 PM, Christoph Lameter wrote:
>>> On Fri, 11 May 2012, Glauber Costa wrote:
>>>
>>>>> Adding a VM_BUG_ON may be useful to make sure that kmem_cache_free is
>>>>> always passed the correct slab cache.
>>>>
>>>> Well, problem is , it isn't always passed the "correct" slab cache.
>>>> At least not after this series, since we'll have child caches associated
>>>> with
>>>> the main cache.
>>>>
>>>> So we'll pass, for instance, kmem_cache_free(dentry_cache...), but will in
>>>> fact free from the memcg copy of the dentry cache.
>>>
>>> Urg. But then please only do this for the MEMCG case and add a fat big
>>> warning in kmem_cache_free.
>>
>> I can do that, of course.
>> Another option if you don't oppose, is to add another field in the kmem_cache
>> structure (I tried to keep them at a minimum),
>> to record the parent cache we got created from.
>>
>> Then, it gets trivial to do the following:
>>
>> VM_BUG_ON(page->slab != s&& page->slab != s->parent_cache);
>
> Sounds ok but I need to catch up on what this whole memcg thing in slab
> allocators should accomplish in order to say something definite.
>
Fair enough.

Thank you in advance for your time reviewing this!

Subject: Re: [PATCH v2 04/29] slub: always get the cache from its page in kfree
Posted by Christoph Lameter on Fri, 11 May 2012 18:32:48 GMT
View Forum Message <> Reply to Message

On Fri, 11 May 2012, Glauber Costa wrote:

> Thank you in advance for your time reviewing this!

Where do I find the rationale for all of this? Trouble is that pages can
contain multiple objects f.e. so accounting of pages to groups is a bit fuzzy.
I have not followed memcg too much since it is not relevant (actual

it is potentially significantly harmful given the performance
impact) to the work loads that I am using.

Subject: Re: [PATCH v2 04/29] slub: always get the cache from its page in kfree
Posted by Glauber Costa on Fri, 11 May 2012 18:42:42 GMT

View Forum Message <> Reply to Message

On 05/11/2012 03:32 PM, Christoph Lameter wrote:
> On Fri, 11 May 2012, Glauber Costa wrote:
>
>> Thank you in advance for your time reviewing this!
>
> Where do I find the rationale for all of this? Trouble is that pages can
> contain multiple objects f.e. so accounting of pages to groups is a bit fuzzy.
> I have not followed memcg too much since it is not relevant (actual
> it is potentially significantly harmful given the performance
> impact) to the work loads that I am using.
>
It's been spread during last discussions. The user-visible part is
documented in the last patch, but I'll try to use this space here to
summarize more of the internals (it can also go somewhere in the tree
if needed):

We want to limit the amount of kernel memory tasks inside a memory
cgroup use. slab is not the only one of them, but it is quite significant.

For that, the least invasive, and most reasonable way we found to do it,
is to create a copy of each slab inside the memcg. Or almost: we lazy
create them, so only slabs that are touched by the memcg are created.

So we don't mix pages from multiple memcgs in the same cache - we
believe that would be too confusing.

/proc/slabinfo reflects this information, by listing the memcg-specific
slabs.

This also appears in a memcg-specific memory.kmem.slabinfo.

Also note that accounting is not done until kernel memory is limited.
And if no memcg is limited, the code is wrapped inside static_key
branches. So it should be completely patched out if you don't put stuff
inside memcg.

Subject: Re: [PATCH v2 04/29] slub: always get the cache from its page in kfree
Posted by Christoph Lameter on Fri, 11 May 2012 18:56:38 GMT

View Forum Message <> Reply to Message

On Fri, 11 May 2012, Glauber Costa wrote:

> So we don't mix pages from multiple memcgs in the same cache - we believe that
> would be too confusing.

Well subsystem create caches and other things that are shared between
multiple processes. How can you track that?

> /proc/slabinfo reflects this information, by listing the memcg-specific slabs.

What about /sys/kernel/slab/*?

Subject: Re: [PATCH v2 04/29] slub: always get the cache from its page in kfree
Posted by Glauber Costa on Fri, 11 May 2012 18:58:49 GMT

View Forum Message <> Reply to Message

On 05/11/2012 03:56 PM, Christoph Lameter wrote:
> On Fri, 11 May 2012, Glauber Costa wrote:
>
>> So we don't mix pages from multiple memcgs in the same cache - we believe that
>> would be too confusing.
>
> Well subsystem create caches and other things that are shared between
> multiple processes. How can you track that?

Each process that belongs to a memcg triggers the creation of a new
child kmem cache.

>> /proc/slabinfo reflects this information, by listing the memcg-specific slabs.
>
> What about /sys/kernel/slab/*?

From the PoV of the global system, what you'll see is something like:
dentry , dentry(2:memcg1), dentry(2:memcg2), etc.

No attempt is made to provide any view of those caches in a logically
grouped way - the global system can view them independently.

Subject: Re: [PATCH v2 04/29] slub: always get the cache from its page in kfree
Posted by Christoph Lameter on Fri, 11 May 2012 19:09:15 GMT

View Forum Message <> Reply to Message

On Fri, 11 May 2012, Glauber Costa wrote:

> On 05/11/2012 03:56 PM, Christoph Lameter wrote:
> > On Fri, 11 May 2012, Glauber Costa wrote:
> >
> > > So we don't mix pages from multiple memcgs in the same cache - we believe
> > > that
> > > would be too confusing.
> >
> > Well subsystem create caches and other things that are shared between
> > multiple processes. How can you track that?
>
> Each process that belongs to a memcg triggers the creation of a new child kmem
> cache.

I see that. But there are other subsystems from slab allocators that do
the same. There are also objects that may be used by multiple processes.
F.e what about shm?

> > > /proc/slabinfo reflects this information, by listing the memcg-specific
> > > slabs.
> >
> > What about /sys/kernel/slab/*?
>
> From the PoV of the global system, what you'll see is something like:
> dentry , dentry(2:memcg1), dentry(2:memcg2), etc.

Hmmm.. Would be better to have a hierachy there. /proc/slabinfo is more
legacy.

Subject: Re: [PATCH v2 04/29] slub: always get the cache from its page in kfree
Posted by Glauber Costa on Fri, 11 May 2012 19:11:05 GMT

View Forum Message <> Reply to Message

On 05/11/2012 04:09 PM, Christoph Lameter wrote:
> On Fri, 11 May 2012, Glauber Costa wrote:
>
>> On 05/11/2012 03:56 PM, Christoph Lameter wrote:
>>> On Fri, 11 May 2012, Glauber Costa wrote:
>>>
>>>> So we don't mix pages from multiple memcgs in the same cache - we believe
>>>> that
>>>> would be too confusing.
>>>
>>> Well subsystem create caches and other things that are shared between
>>> multiple processes. How can you track that?
>>
>> Each process that belongs to a memcg triggers the creation of a new child kmem
>> cache.

>
> I see that. But there are other subsystems from slab allocators that do
> the same. There are also objects that may be used by multiple processes.

This is also true for normal user pages. And then, we do what memcg
does: first one to touch, gets accounted. I don't think deviating from
the memcg behavior for user pages makes much sense here.

A cache won't go away while it still have objects, even after the memcg
is removed (it is marked as dead)

> F.e what about shm?
>
>>>> /proc/slabinfo reflects this information, by listing the memcg-specific
>>>> slabs.
>>>
>>> What about /sys/kernel/slab/*?
>>
>> From the PoV of the global system, what you'll see is something like:
>> dentry , dentry(2:memcg1), dentry(2:memcg2), etc.
>
> Hmmm.. Would be better to have a hierachy there. /proc/slabinfo is more
> legacy.

I can take a look at that then. Assuming you agree with all the rest, is
looking into that a pre-requisite for merging, or is something that can
be deferred for a phase2 ? (We still don't do shrinkers, for instance,
so this is sure to have a phase2)

Subject: Re: [PATCH v2 04/29] slub: always get the cache from its page in kfree
Posted by Christoph Lameter on Fri, 11 May 2012 19:20:36 GMT
View Forum Message <> Reply to Message

On Fri, 11 May 2012, Glauber Costa wrote:

> > I see that. But there are other subsystems from slab allocators that do
> > the same. There are also objects that may be used by multiple processes.
>
> This is also true for normal user pages. And then, we do what memcg does:
> first one to touch, gets accounted. I don't think deviating from the memcg
> behavior for user pages makes much sense here.
>
> A cache won't go away while it still have objects, even after the memcg is
> removed (it is marked as dead)

Ok so we will have some dead pages around that are then repatriated to
the / set?

> > Hmmm.. Would be better to have a hierachy there. /proc/slabinfo is more
> > legacy.
>
> I can take a look at that then. Assuming you agree with all the rest, is
> looking into that a pre-requisite for merging, or is something that can be
> deferred for a phase2 ? (We still don't do shrinkers, for instance, so this is
> sure to have a phase2)

Not a prerequisite for merging but note that I intend to rework the
allocators to extract common code so that they have the same sysfs
interface, error reporting and failure scenarios. We can at that time
also add support for /sys/kernel/slab to memcg. (/sys/memcg/<name>/slab/* ?)

Subject: Re: [PATCH v2 04/29] slub: always get the cache from its page in kfree
Posted by Glauber Costa on Fri, 11 May 2012 19:24:11 GMT

View Forum Message <> Reply to Message

On 05/11/2012 04:20 PM, Christoph Lameter wrote:
> On Fri, 11 May 2012, Glauber Costa wrote:
>
>>> I see that. But there are other subsystems from slab allocators that do
>>> the same. There are also objects that may be used by multiple processes.
>>
>> This is also true for normal user pages. And then, we do what memcg does:
>> first one to touch, gets accounted. I don't think deviating from the memcg
>> behavior for user pages makes much sense here.
>>
>> A cache won't go away while it still have objects, even after the memcg is
>> removed (it is marked as dead)
>
> Ok so we will have some dead pages around that are then repatriated to
> the / set?

No, they are not repatriated. I actually wrote code for that once in my
first series, but it was the general feeling at the time that it was too
complicated. (and I only tried for the slub, not slab)

So instead, we just keep the cache around, until the objects go away.
It will show in slabinfo as dentry(css_id:memcgname)dead

For the record, I wrote that code because I found a nice feature, but I
totally agree with the complicated part.

Also, in normal scenarios, dead caches are not expected to be common.
Most of them should go away as memcg dies.

>>> Hmmm.. Would be better to have a hierachy there. /proc/slabinfo is more
>>> legacy.
>>
>> I can take a look at that then. Assuming you agree with all the rest, is
>> looking into that a pre-requisite for merging, or is something that can be
>> deferred for a phase2 ? (We still don't do shrinkers, for instance, so this is
>> sure to have a phase2)
>
> Not a prerequisite for merging but note that I intend to rework the
> allocators to extract common code so that they have the same sysfs
> interface, error reporting and failure scenarios. We can at that time
> also add support for /sys/kernel/slab to memcg. (/sys/memcg/<name>/slab/* ?)

Yes, that would be a good plan.

Subject: Re: [PATCH v2 19/29] skip memcg kmem allocations in specified code regions
Posted by KAMEZAWA Hiroyuki on Tue, 15 May 2012 02:46:48 GMT

View Forum Message <> Reply to Message

(2012/05/12 2:44), Glauber Costa wrote:

> This patch creates a mechanism that skip memcg allocations during
> certain pieces of our core code. It basically works in the same way
> as preempt_disable()/preempt_enable(): By marking a region under
> which all allocations will be accounted to the root memcg.
>
> We need this to prevent races in early cache creation, when we
> allocate data using caches that are not necessarily created already.
>
> 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>

The concept seems okay to me but...

```
> ---
>  include/linux/sched.h |    1 +
>  mm/memcontrol.c       |   25 +++++++++++++++++++++++++
>  2 files changed, 26 insertions(+), 0 deletions(-)
>
> diff --git a/include/linux/sched.h b/include/linux/sched.h
> index 81a173c..0501114 100644
> --- a/include/linux/sched.h
> +++ b/include/linux/sched.h
> @@ -1613,6 +1613,7 @@ struct task_struct {
>    unsigned long nr_pages; /* uncharged usage */
>    unsigned long memsw_nr_pages; /* uncharged mem+swap usage */
>   } memcg_batch;
> + atomic_t memcg_kmem_skip_account;
```

If only 'current' thread touch this, you don't need to make this atomic counter.
you can use 'long'.

Thanks,
-Kame

Subject: Re: [PATCH v2 18/29] memcg: kmem controller charge/uncharge infrastructure

Posted by KAMEZAWA Hiroyuki on Tue, 15 May 2012 02:57:35 GMT

View Forum Message <> Reply to Message

(2012/05/12 2:44), Glauber Costa 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 |   67 ++++++++
>  init/Kconfig               |    2 +-
>  mm/memcontrol.c            |  379 ++++++++++++++++++++++++++++++++++++++++++++-
>  3 files changed, 446 insertions(+), 2 deletions(-)
>
> diff --git a/include/linux/memcontrol.h b/include/linux/memcontrol.h
> index f93021a..c555799 100644
> --- a/include/linux/memcontrol.h
> +++ b/include/linux/memcontrol.h
> @@ -21,6 +21,7 @@
>  #define _LINUX_MEMCONTROL_H
>  #include <linux/cgroup.h>
>  #include <linux/vm_event_item.h>
> +#include <linux/hardirq.h>
>
>  struct mem_cgroup;
>  struct page_cgroup;
> @@ -447,6 +448,19 @@ void mem_cgroup_register_cache(struct mem_cgroup *memcg,
>  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);
> +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_new_kmem_page(struct page *page, gfp_t gfp);
> +void __mem_cgroup_free_kmem_page(struct page *page);
> +
> +struct kmem_cache *
> +__mem_cgroup_get_kmem_cache(struct kmem_cache *cachep, gfp_t gfp);
> +
> +#define mem_cgroup_kmem_on 1
>  #else
>  static inline void mem_cgroup_register_cache(struct mem_cgroup *memcg,
>          struct kmem_cache *s)
> @@ -463,6 +477,59 @@ 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_new_kmem_page(a, gfp) false
> +#define __mem_cgroup_uncharge_slab(a, b)
> +#define __mem_cgroup_free_kmem_page(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 && current->mm && !in_interrupt())
> +  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_new_kmem_page(struct page *page, gfp_t gfp)
> +{
> + if (mem_cgroup_kmem_on && current->mm && !in_interrupt())
> +  return __mem_cgroup_new_kmem_page(page, gfp);
> + return true;
> +}
> +
> +static __always_inline
> +void mem_cgroup_free_kmem_page(struct page *page)
> +{
> + if (mem_cgroup_kmem_on)
> +  __mem_cgroup_free_kmem_page(page);
> +}
>  #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 a8171cb..5a7416b 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) {
> @@ -484,7 +496,14 @@ char *mem_cgroup_cache_name(struct mem_cgroup *memcg, struct
kmem_cache *cachep)
>   return name;
>  }
>
> +static inline bool mem_cgroup_kmem_enabled(struct mem_cgroup *memcg)
> +{
> + return !mem_cgroup_disabled() && memcg &&
> +        !mem_cgroup_is_root(memcg) && memcg->kmem_accounted;
> +}
> +
>  struct ida cache_types;
> +static DEFINE_MUTEX(memcg_cache_mutex);
>
>  void mem_cgroup_register_cache(struct mem_cgroup *memcg,
>           struct kmem_cache *cachep)
> @@ -504,6 +523,298 @@ void mem_cgroup_release_cache(struct kmem_cache *cachep)
>   if (cachep->memcg_params.id != -1)
>    ida_simple_remove(&cache_types, cachep->memcg_params.id);
>  }
> +
> +
> +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;
> + atomic_set(&new_cachep->memcg_params.refcnt, 1);
> +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);
> +
> +/*
> + * 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 create_work *cw, *tmp;
> + unsigned long flags;
> + LIST_HEAD(create_unlocked);
> +
> + spin_lock_irqsave(&cache_queue_lock, flags);
> + list_for_each_entry_safe(cw, tmp, &create_queue, list)
> +  list_move(&cw->list, &create_unlocked);
> + spin_unlock_irqrestore(&cache_queue_lock, flags);
> +
> + list_for_each_entry_safe(cw, tmp, &create_unlocked, list) {
> +  list_del(&cw->list);
> +  memcg_create_kmem_cache(cw->memcg, cw->cachep);
> +  /* Drop the reference gotten when we enqueued. */
> +  css_put(&cw->memcg->css);
> +  kfree(cw);
> + }
> +}
> +
> +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(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.
> + * We 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.
> + *
> + * Can't be called in interrupt context or from kernel threads.
> + * This function needs to be called 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;
> + struct task_struct *p;
> +
> + gfp |=  cachep->allocflags;
> +
> + if (cachep->memcg_params.memcg)
> +  return cachep;
> +
> + idx = cachep->memcg_params.id;
> + VM_BUG_ON(idx == -1);
> +
> + p = rcu_dereference(current->mm->owner);
> + memcg = mem_cgroup_from_task(p);
> +
> + if (!mem_cgroup_kmem_enabled(memcg))
> +  return cachep;
> +
> + if (memcg->slabs[idx] == NULL) {
> +  memcg_create_cache_enqueue(memcg, cachep);
> +  return cachep;
> + }
> +
> + return memcg->slabs[idx];
> +}
> +EXPORT_SYMBOL(__mem_cgroup_get_kmem_cache);
> +
> +bool __mem_cgroup_new_kmem_page(struct page *page, gfp_t gfp)
> +{
> + struct mem_cgroup *memcg;
> + struct page_cgroup *pc;
> + bool ret = true;
> + size_t size;
> + struct task_struct *p;
> +
> + if (!current->mm || in_interrupt())
> +  return true;
> +
> + rcu_read_lock();
> + p = rcu_dereference(current->mm->owner);
> + memcg = mem_cgroup_from_task(p);
> +
> + if (!mem_cgroup_kmem_enabled(memcg))
> +  goto out;
> +
> + mem_cgroup_get(memcg);
> +
> + size = (1 << compound_order(page)) << PAGE_SHIFT;
> +
> + ret = memcg_charge_kmem(memcg, gfp, size) == 0;
> + if (!ret) {
> +  mem_cgroup_put(memcg);
> +  goto out;
> + }
> +
> + pc = lookup_page_cgroup(page);
> + lock_page_cgroup(pc);
> + pc->mem_cgroup = memcg;
> + SetPageCgroupUsed(pc);
> + unlock_page_cgroup(pc);
> +
> +out:
> + rcu_read_unlock();
> + return ret;
> +}
> +EXPORT_SYMBOL(__mem_cgroup_new_kmem_page);
> +
> +void __mem_cgroup_free_kmem_page(struct page *page)
> +{
> + struct mem_cgroup *memcg;
> + size_t size;
> + struct page_cgroup *pc;
> +
> + if (mem_cgroup_disabled())
> +  return;
> +
> + pc = lookup_page_cgroup(page);
> + lock_page_cgroup(pc);
> + memcg = pc->mem_cgroup;
> + pc->mem_cgroup = NULL;
> + if (!PageCgroupUsed(pc)) {
> +  unlock_page_cgroup(pc);
> +  return;
> + }
> + ClearPageCgroupUsed(pc);
> + unlock_page_cgroup(pc);
> +
> + /*
> +  * The classical disabled check won't work
> +  * for uncharge, since it is possible that the user enabled
> +  * kmem tracking, allocated, and then disabled.
> +  *
> +  * We trust if there is a memcg associated with the page,
> +  * it is a valid allocation
> +  */
> +
> + if (!memcg)
> +  return;
> +
> + WARN_ON(mem_cgroup_is_root(memcg));
> + size = (1 << compound_order(page)) << PAGE_SHIFT;
> + memcg_uncharge_kmem(memcg, size);
> + mem_cgroup_put(memcg);
> +}
> +EXPORT_SYMBOL(__mem_cgroup_free_kmem_page);
> +
> +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;
> + rcu_read_unlock();
> +
> + /*
> +  * The classical disabled check won't work
> +  * for uncharge, since it is possible that the user enabled
> +  * kmem tracking, allocated, and then disabled.
> +  *
> +  * We trust if there is a memcg associated with the slab,
> +  * it is a valid allocation
> +  */
> + if (!memcg)
> +  return;
> +
> + 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++)
> +  memcg->slabs[i] = NULL;
> +}
>  #endif /* CONFIG_CGROUP_MEM_RES_CTLR_KMEM */
>
>  static void drain_all_stock_async(struct mem_cgroup *memcg);
> @@ -4760,7 +5071,11 @@ 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)
> @@ -5777,3 +6092,65 @@ 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 == -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 (do_swap_account)
> +   res_counter_charge_nofail(&memcg->memsw, delta,
> +        &fail_res);
> + } else if (ret == -ENOMEM)
> +  return ret;
> +
> + if (nofail)
> +  res_counter_charge_nofail(&memcg->kmem, delta, &fail_res);
> + else
> +  ret = res_counter_charge(&memcg->kmem, delta, &fail_res);
```

Ouch, you allow usage > limit ? It's BUG.

IMHO, if GFP_NOFAIL, memcg accounting should be skipped. Please

```
if (gfp_mask & __GFP_NOFAIL)
 return 0;
```

Or avoid calling memcg_charge_kmem() you can do that as you do in patch 19/29,
I guess you can use a trick like

```
== in 19/29
+ if (!current->mm || atomic_read(&current->memcg_kmem_skip_account))
+  return cachep;
+
 gfp |=  cachep->allocflags;
==
```

```
== change like this
 gfp |= cachep->allocflags;

 if (!current->mm || current->memcg_kmem_skip_account || gfp & __GFP_NOFAIL))
==
```

Is this difficult ?

Thanks,
-Kame

Subject: Re: [PATCH v2 11/29] cgroups: ability to stop res charge propagation on bounded ancestor
Posted by KAMEZAWA Hiroyuki on Tue, 15 May 2012 02:59:45 GMT

View Forum Message <> Reply to Message

(2012/05/12 2:44), Glauber Costa wrote:

> From: Frederic Weisbecker <fweisbec@gmail.com>
>
> Moving a task from a cgroup to another may require to substract its

> resource charge from the old cgroup and add it to the new one.
>
> For this to happen, the uncharge/charge propagation can just stop when we
> reach the common ancestor for the two cgroups. Further the performance
> reasons, we also want to avoid to temporarily overload the common
> ancestors with a non-accurate resource counter usage if we charge first
> the new cgroup and uncharge the old one thereafter. This is going to be a
> requirement for the coming max number of task subsystem.
>
> To solve this, provide a pair of new API that can charge/uncharge a
> resource counter until we reach a given ancestor.
>
> Signed-off-by: Frederic Weisbecker <fweisbec@gmail.com>
> Acked-by: Paul Menage <paul@paulmenage.org>
> Acked-by: Glauber Costa <glommer@parallels.com>
> Cc: Li Zefan <lizf@cn.fujitsu.com>
> Cc: Johannes Weiner <hannes@cmpxchg.org>
> Cc: Aditya Kali <adityakali@google.com>
> Cc: Oleg Nesterov <oleg@redhat.com>
> Cc: Kay Sievers <kay.sievers@vrfy.org>
> Cc: Tim Hockin <thockin@hockin.org>
> Cc: Tejun Heo <htejun@gmail.com>
> Acked-by: Kirill A. Shutemov <kirill@shutemov.name>
> Signed-off-by: Andrew Morton <akpm@linux-foundation.org>

Where is this function called in this series ?

Thanks,
-Kame

Subject: Re: [PATCH v2 02/29] slub: fix slab_state for slub
Posted by David Rientjes on Tue, 15 May 2012 21:55:04 GMT
View Forum Message <> Reply to Message

On Fri, 11 May 2012, Glauber Costa wrote:

> When the slub code wants to know if the sysfs state has already been
> initialized, it tests for slab_state == SYSFS. This is quite fragile,
> since new state can be added in the future (it is, in fact, for
> memcg caches). This patch fixes this behavior so the test matches
> >= SYSFS, as all other state does.
>
> Signed-off-by: Glauber Costa <glommer@parallels.com>

Acked-by: David Rientjes <rientjes@google.com>

Can be merged now, there's no dependency on the rest of this patchset.

Subject: Re: [PATCH v2 05/29] slab: rename gfpflags to allocflags
Posted by David Rientjes on Tue, 15 May 2012 21:57:05 GMT

View Forum Message <> Reply to Message

On Fri, 11 May 2012, Glauber Costa wrote:

> 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>

Acked-by: David Rientjes <rientjes@google.com>

Same, can be merged now with no dependency on the rest of this patchset.

Subject: Re: [PATCH v2 01/29] slab: dup name string
Posted by David Rientjes on Tue, 15 May 2012 22:04:14 GMT

View Forum Message <> Reply to Message

On Fri, 11 May 2012, Glauber Costa wrote:

```
> diff --git a/mm/slab.c b/mm/slab.c
> index e901a36..91b9c13 100644
> --- a/mm/slab.c
> +++ b/mm/slab.c
> @@ -2118,6 +2118,7 @@ static void __kmem_cache_destroy(struct kmem_cache *cachep)
>     kfree(l3);
>    }
>   }
> + kfree(cachep->name);
>   kmem_cache_free(&cache_cache, cachep);
>  }
>
> @@ -2526,7 +2527,7 @@ 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 = kstrdup(name, GFP_KERNEL);
>
>   if (setup_cpu_cache(cachep, gfp)) {
>    __kmem_cache_destroy(cachep);
```

Couple problems:

 - allocating memory for a string of an unknown, unchecked size, and

 - could potentially return NULL which I suspect will cause problems
   later.

Subject: Re: [PATCH v2 02/29] slub: fix slab_state for slub
Posted by Glauber Costa on Wed, 16 May 2012 06:10:24 GMT

View Forum Message <> Reply to Message

On 05/16/2012 01:55 AM, David Rientjes wrote:
> On Fri, 11 May 2012, Glauber Costa wrote:
>
>> When the slub code wants to know if the sysfs state has already been
>> initialized, it tests for slab_state == SYSFS. This is quite fragile,
>> since new state can be added in the future (it is, in fact, for
>> memcg caches). This patch fixes this behavior so the test matches
>>> = SYSFS, as all other state does.
>>
>> Signed-off-by: Glauber Costa<glommer@parallels.com>
>
> Acked-by: David Rientjes<rientjes@google.com>
>
> Can be merged now, there's no dependency on the rest of this patchset.

Agreed.

If anyone is willing to, would make my life easier in the future.

Valid for all patches that fall in this category (there are quite a few
in the purely memcg land as well)

Subject: Re: [PATCH v2 01/29] slab: dup name string
Posted by Glauber Costa on Wed, 16 May 2012 06:12:04 GMT

View Forum Message <> Reply to Message

On 05/16/2012 02:04 AM, David Rientjes wrote:
> On Fri, 11 May 2012, Glauber Costa wrote:
>
>> diff --git a/mm/slab.c b/mm/slab.c
>> index e901a36..91b9c13 100644
>> --- a/mm/slab.c
>> +++ b/mm/slab.c

```
>> @@ -2118,6 +2118,7 @@ static void __kmem_cache_destroy(struct kmem_cache *cachep)
>>      kfree(l3);
>>     }
>>    }
>> + kfree(cachep->name);
>>    kmem_cache_free(&cache_cache, cachep);
>>   }
>>
>> @@ -2526,7 +2527,7 @@ 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 = kstrdup(name, GFP_KERNEL);
>>
>>    if (setup_cpu_cache(cachep, gfp)) {
>>     __kmem_cache_destroy(cachep);
>
> Couple problems:
>
>   - allocating memory for a string of an unknown, unchecked size, and
>
>   - could potentially return NULL which I suspect will cause problems
>     later.
```

Well, this is what slub does.

I sent already two patches for it: One removing this from the slub, one adding this to the slab.

Right now I am comfortable with this one, because it makes it slightly easier in the latest patches of my series.

But note the word: slightest.

I am comfortable with any, provided slub and slab start behaving the same.

So whatever you guys decide between yourselves is fine, provided there is a decision.

Thanks for your review, David!

Subject: Re: [PATCH v2 11/29] cgroups: ability to stop res charge propagation on bounded ancestor
Posted by Glauber Costa on Wed, 16 May 2012 06:16:51 GMT
View Forum Message <> Reply to Message

On 05/15/2012 06:59 AM, KAMEZAWA Hiroyuki wrote:
> (2012/05/12 2:44), Glauber Costa wrote:
>
>> From: Frederic Weisbecker<fweisbec@gmail.com>
>>
>> Moving a task from a cgroup to another may require to substract its
>> resource charge from the old cgroup and add it to the new one.
>>
>> For this to happen, the uncharge/charge propagation can just stop when we
>> reach the common ancestor for the two cgroups. Further the performance
>> reasons, we also want to avoid to temporarily overload the common
>> ancestors with a non-accurate resource counter usage if we charge first
>> the new cgroup and uncharge the old one thereafter. This is going to be a
>> requirement for the coming max number of task subsystem.
>>
>> To solve this, provide a pair of new API that can charge/uncharge a
>> resource counter until we reach a given ancestor.
>>
>> Signed-off-by: Frederic Weisbecker<fweisbec@gmail.com>
>> Acked-by: Paul Menage<paul@paulmenage.org>
>> Acked-by: Glauber Costa<glommer@parallels.com>
>> Cc: Li Zefan<lizf@cn.fujitsu.com>
>> Cc: Johannes Weiner<hannes@cmpxchg.org>
>> Cc: Aditya Kali<adityakali@google.com>
>> Cc: Oleg Nesterov<oleg@redhat.com>
>> Cc: Kay Sievers<kay.sievers@vrfy.org>
>> Cc: Tim Hockin<thockin@hockin.org>
>> Cc: Tejun Heo<htejun@gmail.com>
>> Acked-by: Kirill A. Shutemov<kirill@shutemov.name>
>> Signed-off-by: Andrew Morton<akpm@linux-foundation.org>
>
>
> Where is this function called in this series ?
>
> Thanks,
> -Kame
>
It is not... anymore!
But I forgot the patch among the "pre-requisite" patches I had.

Thanks, this can be dropped.

Subject: Re: [PATCH v2 19/29] skip memcg kmem allocations in specified code regions
Posted by Glauber Costa on Wed, 16 May 2012 06:19:15 GMT
View Forum Message <> Reply to Message

On 05/15/2012 06:46 AM, KAMEZAWA Hiroyuki wrote:
> (2012/05/12 2:44), Glauber Costa wrote:
>
>> This patch creates a mechanism that skip memcg allocations during
>> certain pieces of our core code. It basically works in the same way
>> as preempt_disable()/preempt_enable(): By marking a region under
>> which all allocations will be accounted to the root memcg.
>>
>> We need this to prevent races in early cache creation, when we
>> allocate data using caches that are not necessarily created already.
>>
>> 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>
>
>
> The concept seems okay to me but...
>

```
>> ---
>>   include/linux/sched.h |    1 +
>>   mm/memcontrol.c       |   25 +++++++++++++++++++++++++
>>   2 files changed, 26 insertions(+), 0 deletions(-)
>>
>> diff --git a/include/linux/sched.h b/include/linux/sched.h
>> index 81a173c..0501114 100644
>> --- a/include/linux/sched.h
>> +++ b/include/linux/sched.h
>> @@ -1613,6 +1613,7 @@ struct task_struct {
>>     unsigned long nr_pages; /* uncharged usage */
>>     unsigned long memsw_nr_pages; /* uncharged mem+swap usage */
>>    } memcg_batch;
>> + atomic_t memcg_kmem_skip_account;
```

>
>
> If only 'current' thread touch this, you don't need to make this atomic counter.
> you can use 'long'.
>

You're absolutely right, Kame, thanks.
I first used atomic_t because I had it tested against current->mm->owner.

Do you, btw, agree to use current instead of owner here?
You can find the rationale in earlier mails between me and Suleiman.

Subject: Re: [PATCH v2 18/29] memcg: kmem controller charge/uncharge infrastructure
Posted by Glauber Costa on Wed, 16 May 2012 06:42:05 GMT
View Forum Message <> Reply to Message

On 05/15/2012 06:57 AM, KAMEZAWA Hiroyuki wrote:

```
>> +#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 == -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 (do_swap_account)
>> >  +   res_counter_charge_nofail(&memcg->memsw, delta,
>> >  +      &fail_res);
>> >  + } else if (ret == -ENOMEM)
>> >  +  return ret;
>> >  +
>> >  + if (nofail)
>> >  +  res_counter_charge_nofail(&memcg->kmem, delta,&fail_res);
>> >  + else
>> >  +  ret = res_counter_charge(&memcg->kmem, delta,&fail_res);
>
> Ouch, you allow usage>  limit ? It's BUG.
>
```

> IMHO, if GFP_NOFAIL, memcg accounting should be skipped. Please
>
> if (gfp_mask& __GFP_NOFAIL)
> return 0;
>
> Or avoid calling memcg_charge_kmem() you can do that as you do in patch 19/29,
> I guess you can use a trick like
>
> == in 19/29
> + if (!current->mm || atomic_read(&current->memcg_kmem_skip_account))
> + return cachep;
> +
>   gfp |= cachep->allocflags;
> ==
>
> == change like this
>   gfp |= cachep->allocflags;
>
>   if (!current->mm || current->memcg_kmem_skip_account || gfp& __GFP_NOFAIL))
> ==
>
> Is this difficult ?
>
> Thanks,
> -Kame

Well, we disagree with that.
I actually voiced this earlier to Suleiman in the thread, but it is good
that you brought this up again - this is quite important.

I will repeat my rationale here, and if you still are not convinced,
tell me and I will be happy to switch over.

I believe that the whole reasoning behind this, is to have allocations
failing if we go over limit. If the allocation won't fail anyway, it
doesn't really matter who we charge this to.

However, if the allocation still came from a particular memcg, those
nofail allocation may prevent it to use more memory when a normal
allocation takes place.

Consider this:

limit = 4M
usage = 4M - 4k

If at this point the memcg hits a NOFAIL allocation worth 2 pages, by
the method I am using, the memcg will be at 4M + 4k after the

allocation. Charging it to the root memcg will leave it at 4M - 4k.

This means that to be able to allocate a page again, you need to free
two other pages, be it the 2 pages used by the GFP allocation or any
other. In other words: the memcg that originated the charge is held
accountable for it. If he says it can't fail for whatever reason, fine,
we respect that, but we punish it later for other allocations.

Without that GFP_NOFAIL becomes just a nice way for people to bypass
those controls altogether, since after a ton of GFP_NOFAIL allocations,
normal allocations will still succeed.

The change you propose is totally doable. I just don't believe it should
be done.

But let me know where you stand.

Subject: Re: [PATCH v2 19/29] skip memcg kmem allocations in specified code regions
Posted by KAMEZAWA Hiroyuki on Wed, 16 May 2012 07:55:49 GMT

View Forum Message <> Reply to Message

(2012/05/16 15:19), Glauber Costa wrote:

> On 05/15/2012 06:46 AM, KAMEZAWA Hiroyuki wrote:
>> (2012/05/12 2:44), Glauber Costa wrote:
>>
>>> This patch creates a mechanism that skip memcg allocations during
>>> certain pieces of our core code. It basically works in the same way
>>> as preempt_disable()/preempt_enable(): By marking a region under
>>> which all allocations will be accounted to the root memcg.
>>>
>>> We need this to prevent races in early cache creation, when we
>>> allocate data using caches that are not necessarily created already.
>>>
>>> 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>
>>
>>
>> The concept seems okay to me but...
>>
>>> ---

```
>>>   include/linux/sched.h |    1 +
>>>   mm/memcontrol.c       |   25 +++++++++++++++++++++++++
>>>   2 files changed, 26 insertions(+), 0 deletions(-)
>>>
>>> diff --git a/include/linux/sched.h b/include/linux/sched.h
>>> index 81a173c..0501114 100644
>>> --- a/include/linux/sched.h
>>> +++ b/include/linux/sched.h
>>> @@ -1613,6 +1613,7 @@ struct task_struct {
>>>     unsigned long nr_pages; /* uncharged usage */
>>>     unsigned long memsw_nr_pages; /* uncharged mem+swap usage */
>>>    } memcg_batch;
>>> + atomic_t memcg_kmem_skip_account;
>>
>>
>> If only 'current' thread touch this, you don't need to make this atomic counter.
>> you can use 'long'.
>>
> You're absolutely right, Kame, thanks.
> I first used atomic_t because I had it tested against current->mm->owner.
>
> Do you, btw, agree to use current instead of owner here?
> You can find the rationale in earlier mails between me and Suleiman.
```

I agree to use current. This information depends on the context of callers.

Thanks,
-Kame

Subject: Re: [PATCH v2 18/29] memcg: kmem controller charge/uncharge infrastructure

Posted by KAMEZAWA Hiroyuki on Wed, 16 May 2012 08:18:28 GMT

View Forum Message <> Reply to Message

(2012/05/16 15:42), Glauber Costa wrote:

```
> On 05/15/2012 06:57 AM, KAMEZAWA Hiroyuki wrote:
>>> +#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 == -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 (do_swap_account)
>>>>  +   res_counter_charge_nofail(&memcg->memsw, delta,
>>>>  +      &fail_res);
>>>>  + } else if (ret == -ENOMEM)
>>>>  +  return ret;
>>>>  +
>>>>  + if (nofail)
>>>>  +  res_counter_charge_nofail(&memcg->kmem, delta,&fail_res);
>>>>  + else
>>>>  +  ret = res_counter_charge(&memcg->kmem, delta,&fail_res);
>>
>> Ouch, you allow usage>  limit ? It's BUG.
>>
>> IMHO, if GFP_NOFAIL, memcg accounting should be skipped. Please
>>
>> if (gfp_mask&  __GFP_NOFAIL)
>>  return 0;
>>
>> Or avoid calling memcg_charge_kmem() you can do that as you do in patch 19/29,
>> I guess you can use a trick like
>>
>> == in 19/29
>> + if (!current->mm || atomic_read(&current->memcg_kmem_skip_account))
>> +  return cachep;
>> +
>>   gfp |=  cachep->allocflags;
>> ==
>>
```

>> == change like this
>>   gfp |= cachep->allocflags;
>>
>>   if (!current->mm || current->memcg_kmem_skip_account || gfp& __GFP_NOFAIL))
>> ==
>>
>> Is this difficult ?
>>
>> Thanks,
>> -Kame
>
> Well, we disagree with that.
> I actually voiced this earlier to Suleiman in the thread, but it is good
> that you brought this up again - this is quite important.
>
> I will repeat my rationale here, and if you still are not convinced,
> tell me and I will be happy to switch over.
>
> I believe that the whole reasoning behind this, is to have allocations
> failing if we go over limit. If the allocation won't fail anyway, it
> doesn't really matter who we charge this to.
>
> However, if the allocation still came from a particular memcg, those
> nofail allocation may prevent it to use more memory when a normal
> allocation takes place.
>
> Consider this:
>
> limit = 4M
> usage = 4M - 4k
>
> If at this point the memcg hits a NOFAIL allocation worth 2 pages, by
> the method I am using, the memcg will be at 4M + 4k after the
> allocation. Charging it to the root memcg will leave it at 4M - 4k.
>
> This means that to be able to allocate a page again, you need to free
> two other pages, be it the 2 pages used by the GFP allocation or any
> other. In other words: the memcg that originated the charge is held
> accountable for it. If he says it can't fail for whatever reason, fine,
> we respect that, but we punish it later for other allocations.
>

I personally think 'we punish it later' is bad thing at resource accounting.
We have 'hard limit'. It's not soft limit.

> Without that GFP_NOFAIL becomes just a nice way for people to bypass
> those controls altogether, since after a ton of GFP_NOFAIL allocations,
> normal allocations will still succeed.

>

Allowing people to bypass is not bad because they're kernel.

But, IIUC, from gfp.h
==
 * __GFP_NOFAIL: The VM implementation _must_ retry infinitely: the caller
 * cannot handle allocation failures. This modifier is deprecated and no new
 * users should be added.
==

GFP_NOFAIL will go away and no new user is recommended.

So, please skip GFP_NOFAIL accounting and avoid to write
"usage may go over limit if you're unfortune, sorry" into memcg documentation.

> The change you propose is totally doable. I just don't believe it should
> be done.
>
> But let me know where you stand.
>

My stand point is keeping "usage <= limit" is the spec. and
important in enterprise system. So, please avoid usage > limit.

Thanks,
-Kame

Subject: Re: [PATCH v2 18/29] memcg: kmem controller charge/uncharge infrastructure
Posted by Glauber Costa on Wed, 16 May 2012 08:25:42 GMT

View Forum Message <> Reply to Message

On 05/16/2012 12:18 PM, KAMEZAWA Hiroyuki wrote:
>> If at this point the memcg hits a NOFAIL allocation worth 2 pages, by
>> > the method I am using, the memcg will be at 4M + 4k after the
>> > allocation. Charging it to the root memcg will leave it at 4M - 4k.
>> >
>> > This means that to be able to allocate a page again, you need to free
>> > two other pages, be it the 2 pages used by the GFP allocation or any
>> > other. In other words: the memcg that originated the charge is held
>> > accountable for it. If he says it can't fail for whatever reason, fine,
>> > we respect that, but we punish it later for other allocations.
>> >
> I personally think 'we punish it later' is bad thing at resource accounting.
> We have 'hard limit'. It's not soft limit.

That only makes sense if you will fail the allocation. If you won't, you
are over your hard limit anyway. You are just masquerading that.

>> > Without that GFP_NOFAIL becomes just a nice way for people to bypass
>> > those controls altogether, since after a ton of GFP_NOFAIL allocations,
>> > normal allocations will still succeed.
>> >
> Allowing people to bypass is not bad because they're kernel.

No, they are not. They are in process context, on behalf of a process
that belongs to a valid memcg. If they happen to be a kernel thread,
!current->mm test will send the allocation to the root memcg already.

>
> But, IIUC, from gfp.h
> ==
>   * __GFP_NOFAIL: The VM implementation_must_ retry infinitely: the caller
>   * cannot handle allocation failures. This modifier is deprecated and no new
>   * users should be added.
> ==
>
> GFP_NOFAIL will go away and no new user is recommended.
>
Yes, I am aware of that. That's actually why I don't plan to insist on
this too much - although your e-mail didn't really convince me.

It should not matter in practice.

> So, please skip GFP_NOFAIL accounting and avoid to write
> "usage may go over limit if you're unfortune, sorry" into memcg documentation.

I won't write that, because that's not true. Is more like: "Allocations
that can fail will fail if you go over limit".

>
>> > The change you propose is totally doable. I just don't believe it should
>> > be done.
>> >
>> > But let me know where you stand.
>> >
> My stand point is keeping "usage<= limit" is the spec. and
> important in enterprise system. So, please avoid usage> limit.
>
As I said, I won't make a case here because those allocations shouldn't
matter in real life anyway. I can change it.

Subject: Re: [PATCH v2 18/29] memcg: kmem controller charge/uncharge infrastructure
Posted by KAMEZAWA Hiroyuki on Wed, 16 May 2012 09:15:37 GMT
View Forum Message <> Reply to Message

(2012/05/16 17:25), Glauber Costa wrote:

> On 05/16/2012 12:18 PM, KAMEZAWA Hiroyuki wrote:
>>> If at this point the memcg hits a NOFAIL allocation worth 2 pages, by
>>>> the method I am using, the memcg will be at 4M + 4k after the
>>>> allocation. Charging it to the root memcg will leave it at 4M - 4k.
>>>>
>>>> This means that to be able to allocate a page again, you need to free
>>>> two other pages, be it the 2 pages used by the GFP allocation or any
>>>> other. In other words: the memcg that originated the charge is held
>>>> accountable for it. If he says it can't fail for whatever reason, fine,
>>>> we respect that, but we punish it later for other allocations.
>>>>
>> I personally think 'we punish it later' is bad thing at resource accounting.
>> We have 'hard limit'. It's not soft limit.
>
> That only makes sense if you will fail the allocation. If you won't, you
> are over your hard limit anyway. You are just masquerading that.
>

'showing usage > limit to user' and 'avoid accounting'
is totally different user experience.

>>>> Without that GFP_NOFAIL becomes just a nice way for people to bypass
>>>> those controls altogether, since after a ton of GFP_NOFAIL allocations,
>>>> normal allocations will still succeed.
>>>>
>> Allowing people to bypass is not bad because they're kernel.
>
> No, they are not. They are in process context, on behalf of a process
> that belongs to a valid memcg. If they happen to be a kernel thread,
> !current->mm test will send the allocation to the root memcg already.
>

Yes, but it's kernel code. There will be some special reason to use __GFP_NOFAIL.

>>
>> But, IIUC, from gfp.h
>> ==
>>   * __GFP_NOFAIL: The VM implementation_must_ retry infinitely: the caller

>>   * cannot handle allocation failures. This modifier is deprecated and no new
>>   * users should be added.
>> ==
>>
>> GFP_NOFAIL will go away and no new user is recommended.
>>
> Yes, I am aware of that. That's actually why I don't plan to insist on
> this too much - although your e-mail didn't really convince me.
>
> It should not matter in practice.
>
>> So, please skip GFP_NOFAIL accounting and avoid to write
>> "usage may go over limit if you're unfortune, sorry" into memcg documentation.
>
> I won't write that, because that's not true. Is more like: "Allocations
> that can fail will fail if you go over limit".
>

>>
>>>> The change you propose is totally doable. I just don't believe it should
>>>> be done.
>>>>
>>>> But let me know where you stand.
>>>>
>> My stand point is keeping "usage<= limit" is the spec. and
>> important in enterprise system. So, please avoid usage> limit.
>>
> As I said, I won't make a case here because those allocations shouldn't
> matter in real life anyway. I can change it.
>

My standing point is that 'usage > limit' is bug. So please avoid it if
__GFP_NOFAIL allocation is not very important.

Thanks,
-Kame

Subject: Re: [PATCH v2 02/29] slub: fix slab_state for slub
Posted by Glauber Costa on Thu, 17 May 2012 10:14:18 GMT

View Forum Message <> Reply to Message

On 05/16/2012 01:55 AM, David Rientjes wrote:
> On Fri, 11 May 2012, Glauber Costa wrote:
>
>> When the slub code wants to know if the sysfs state has already been
>> initialized, it tests for slab_state == SYSFS. This is quite fragile,

>> since new state can be added in the future (it is, in fact, for
>> memcg caches). This patch fixes this behavior so the test matches
>>> = SYSFS, as all other state does.
>>
>> Signed-off-by: Glauber Costa<glommer@parallels.com>
>
> Acked-by: David Rientjes<rientjes@google.com>
>
> Can be merged now, there's no dependency on the rest of this patchset.

So, is anyone taking this? I plan another submission this week, let me
know if I should include these two patches or not.