Subject: [PATCH v3 00/28] kmem limitation for memcg
Posted by Glauber Costa on Fri, 25 May 2012 13:03:20 GMT
View Forum Message <> Reply to Message

Hello All,

This is my new take for the memcg kmem accounting. This should merge
all of the previous comments from you, plus fix a bunch of bugs.

At this point, I consider the series pretty mature. Since last submission
2 weeks ago, I focused on broadening the testing coverage. Some bugs were
fixed, but that of course doesn't mean no bugs exist.

I believe some of the early patches here are already in some trees around.
I don't know who should pick this, so if everyone agrees with what's in here,
please just ack them and tell me which tree I should aim for (-mm? Hocko's?)
and I'll rebase it.

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. Enabling and disabling of static_keys incorporate
the last fixes for sock memcg, and should be pretty robust.

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.

```
[ v3 ]
 * fixed lockdep bugs in slab (ordering of get_online_cpus() vs slab_mutex)
 * improved style in slab and slub with less #ifdefs in-code
 * tested and fixed hierarchical accounting (memcg: propagate kmem limiting...)
 * some more small bug fixes
 * No longer using res_counter_charge_nofail for GFP_NOFAIL submissions. Those
   go to the root memcg directly.
 * reordered tests in mem_cgroup_get_kmem_cache so we exit even earlier for
   tasks in root memcg
 * no more memcg state for slub initialization
 * do_tune_cpucache will always (only after FULL) propagate to children when
   they exist.
 * slab itself will destroy the kmem_cache string for chained caches, so we
   don't need to bother with consistency between them.
 * other minor issues
[ v2 ]
 * 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.

Glauber Costa (25):
  slab: move FULL state transition to an initcall
  memcg: Always free struct memcg through schedule_work()
  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
  slub: always get the cache from its page in kfree
  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
  slab: Track all the memcg children of a kmem_cache.
  slub: create slabinfo file for memcg
  slub: track all children of a kmem cache
  memcg: propagate kmem limiting information to children
  Documentation: add documentation for slab tracker for memcg

Suleiman Souhlal (3):
  memcg: Make it possible to use the stock for more than one page.
  memcg: Reclaim when more than one page needed.
  memcg: Per-memcg memory.kmem.slabinfo file.

 Documentation/cgroups/memory.txt |   33 ++
 include/linux/memcontrol.h       |  101 +++++
 include/linux/res_counter.h      |    2 +-
 include/linux/sched.h            |    1 +
 include/linux/slab.h             |   32 ++
 include/linux/slab_def.h         |   79 ++++-
 include/linux/slub_def.h         |   68 +++-
 init/Kconfig                     |    2 +-
 mm/memcontrol.c                  |  897 ++++++++++++++++++++++++++++++++++++++--
 mm/slab.c                        |  423 +++++++++++++++---
 mm/slub.c                  |  282 +++++++++++-
 11 files changed, 1787 insertions(+), 133 deletions(-)

--
1.7.7.6
```

Subject: [PATCH v3 01/28] slab: move FULL state transition to an initcall
Posted by Glauber Costa on Fri, 25 May 2012 13:03:21 GMT
View Forum Message <> Reply to Message

During kmem_cache_init_late(), we transition to the LATE state,
and after some more work, to the FULL state, its last state

This is quite different from slub, that will only transition to
its last state (SYSFS), in a (late)initcall, after a lot more of
the kernel is ready.

This means that in slab, we have no way to taking actions dependent
on the initialization of other pieces of the kernel that are supposed
to start way after kmem_init_late(), such as cgroups initialization.

To achieve more consistency in this behavior, that patch only
transitions to the LATE state in kmem_init_late. In my analysis,
setup_cpu_cache() should be happy to test for >= LATE, instead of
== FULL. It also has passed some tests I've made.

We then only mark FULL state after the reap timers are in place,
meaning that no further setup is expected.

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

```
---
 mm/slab.c |    8 ++++----
 1 files changed, 4 insertions(+), 4 deletions(-)

diff --git a/mm/slab.c b/mm/slab.c
index e901a36..8658d72 100644
--- a/mm/slab.c
+++ b/mm/slab.c
@@ -1695,9 +1695,6 @@ void __init kmem_cache_init_late(void)
    BUG();
  mutex_unlock(&cache_chain_mutex);

- /* Done! */
- g_cpucache_up = FULL;
```

```
-
  /*
   * Register a cpu startup notifier callback that initializes
   * cpu_cache_get for all new cpus
@@ -1727,6 +1724,9 @@ static int __init cpucache_init(void)
   */
  for_each_online_cpu(cpu)
   start_cpu_timer(cpu);
+
+ /* Done! */
+ g_cpucache_up = FULL;
  return 0;
 }
 __initcall(cpucache_init);
@@ -2194,7 +2194,7 @@ static size_t calculate_slab_order(struct kmem_cache *cachep,

 static int __init_refok setup_cpu_cache(struct kmem_cache *cachep, gfp_t gfp)
 {
- if (g_cpucache_up == FULL)
+ if (g_cpucache_up >= LATE)
   return enable_cpucache(cachep, gfp);

  if (g_cpucache_up == NONE) {
--
1.7.7.6
```

Subject: [PATCH v3 02/28] memcg: Always free struct memcg through schedule_work()

Posted by Glauber Costa on Fri, 25 May 2012 13:03:22 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 v3 03/28] slab: rename gfpflags to allocflags
Posted by Glauber Costa on Fri, 25 May 2012 13:03:23 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>
Acked-by: 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 8658d72..1057a32 100644
--- a/mm/slab.c
+++ b/mm/slab.c
@@ -1798,7 +1798,7 @@ static void *kmem_getpages(struct kmem_cache *cachep, gfp_t flags,
int nodeid)
  flags |= __GFP_COMP;
 #endif

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

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

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

--
1.7.7.6
```

Subject: [PATCH v3 06/28] slab: use obj_size field of struct kmem_cache when not debugging

Posted by Glauber Costa on Fri, 25 May 2012 13:03:26 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 1057a32..41345f6 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 =
@@ -2418,8 +2432,9 @@ kmem_cache_create (const char *name, size_t size, size_t align,
   goto oops;

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

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

Subject: [PATCH v3 08/28] res_counter: don't force return value checking in res_counter_charge_nofail

Posted by Glauber Costa on Fri, 25 May 2012 13:03:28 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 v3 09/28] kmem slab accounting basic infrastructure
Posted by Glauber Costa on Fri, 25 May 2012 13:03:29 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 |   78 +++++++++++++++++++++++++++++++++++++++++++++++++++++++++-
 1 files changed, 77 insertions(+), 1 deletions(-)

diff --git a/mm/memcontrol.c b/mm/memcontrol.c
index 789ca5a..b6bac5f 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);
--
1.7.7.6
```

Subject: [PATCH v3 10/28] slab/slub: struct memcg_params
Posted by Glauber Costa on Fri, 25 May 2012 13:03:30 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>
Signed-off-by: Suleiman Souhlal <suleiman@google.com>
CC: Christoph Lameter <cl@linux.com>
CC: Pekka Enberg <penberg@cs.helsinki.fi>
CC: Michal Hocko <mhocko@suse.cz>
CC: Kamezawa Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com>
CC: Johannes Weiner <hannes@cmpxchg.org>

```
---
 include/linux/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 v3 11/28] slub: consider a memcg parameter in kmem_create_cache
Posted by Glauber Costa on Fri, 25 May 2012 13:03:31 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                  |   58 +++++++++++++++++++++++++++++++++++++++++++++---
 4 files changed, 101 insertions(+), 4 deletions(-)

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

 /* Stats that can be updated by kernel. */
 enum mem_cgroup_page_stat_item {
@@ -440,7 +441,20 @@ struct sock;
 #ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
 void sock_update_memcg(struct sock *sk);
 void sock_release_memcg(struct sock *sk);
+int memcg_css_id(struct mem_cgroup *memcg);
+void mem_cgroup_register_cache(struct mem_cgroup *memcg,
+          struct kmem_cache *s);
+void mem_cgroup_release_cache(struct kmem_cache *cachep);
 #else
+static inline void mem_cgroup_register_cache(struct mem_cgroup *memcg,
+          struct kmem_cache *s)
+{
+}
+
+static inline void mem_cgroup_release_cache(struct kmem_cache *cachep)
+{
+}
+
 static inline void sock_update_memcg(struct sock *sk)
 {
 }
diff --git a/include/linux/slab.h b/include/linux/slab.h
index 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 b6bac5f..dacd1fb 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 ffe13fd..d79740c 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:
@@ -3193,6 +3194,7 @@ void kmem_cache_destroy(struct kmem_cache *s)
  s->refcount--;
  if (!s->refcount) {
   list_del(&s->list);
+  mem_cgroup_release_cache(s);
   up_write(&slub_lock);
   if (kmem_cache_close(s)) {
    printk(KERN_ERR "SLUB %s: %s called for cache that "
@@ -3880,7 +3882,7 @@ 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 +3918,19 @@ 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 +3938,12 @@ 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 +3971,8 @@ 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 >= SYSFS)
+    mem_cgroup_register_cache(memcg, s);
    if (sysfs_slab_add(s)) {
     down_write(&slub_lock);
     list_del(&s->list);
@@ -3980,6 +3994,12 @@ 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 +5293,11 @@ 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;
 }
@@ -5375,6 +5400,30 @@ static int sysfs_slab_alias(struct kmem_cache *s, const char *name)
  return 0;
 }

+static void __init memcg_slab_register_all(void)
+{
+
+#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
+ 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);
+ }
+
+ list_for_each_entry(s, &slab_caches, list)
+  mem_cgroup_register_cache(NULL, s);
+
+#endif
```

```
+}
+
+
 static int __init slab_sysfs_init(void)
 {
  struct kmem_cache *s;
@@ -5409,6 +5458,7 @@ static int __init slab_sysfs_init(void)
   kfree(al);
  }

+ memcg_slab_register_all();
  up_write(&slub_lock);
  resiliency_test();
  return 0;
--
1.7.7.6
```

Subject: [PATCH v3 12/28] slab: pass memcg parameter to kmem_cache_create
Posted by Glauber Costa on Fri, 25 May 2012 13:03:32 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

[v3: get_online_cpus need to be outside slab mutex.]
[also, register all caches created before FULL state]

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

```
---
 include/linux/slab_def.h |    7 ++++
 mm/slab.c                |   79 +++++++++++++++++++++++++++++++++++++----------
 2 files changed, 69 insertions(+), 17 deletions(-)

diff --git a/include/linux/slab_def.h b/include/linux/slab_def.h
index 06e4a3e..7c0cdd6 100644
```

```
--- a/include/linux/slab_def.h
+++ b/include/linux/slab_def.h
@@ -102,6 +102,13 @@ struct kmem_cache {
   */
 };

+static inline void store_orig_align(struct kmem_cache *cachep, int orig_align)
+{
+#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
+ cachep->memcg_params.orig_align = orig_align;
+#endif
+}
+
 /* Size description struct for general caches. */
 struct cache_sizes {
  size_t    cs_size;
diff --git a/mm/slab.c b/mm/slab.c
index 41345f6..8bff32a1 100644
--- a/mm/slab.c
+++ b/mm/slab.c
@@ -1729,6 +1729,31 @@ void __init kmem_cache_init_late(void)
   */
 }

+static int __init memcg_slab_register_all(void)
+{
+#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
+ struct kmem_cache *cachep;
+ struct cache_sizes *sizes;
+
+ 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);
+
+ mutex_unlock(&cache_chain_mutex);
+#endif /* CONFIG_CGROUP_MEM_RES_CTLR_KMEM */
+ return 0;
+}
```

```
+
 static int __init cpucache_init(void)
 {
  int cpu;
@@ -1739,6 +1764,8 @@ static int __init cpucache_init(void)
  for_each_online_cpu(cpu)
   start_cpu_timer(cpu);

+ memcg_slab_register_all();
+
  /* Done! */
  g_cpucache_up = FULL;
  return 0;
@@ -2287,14 +2314,15 @@ 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.
   */
@@ -2305,15 +2333,6 @@ kmem_cache_create (const char *name, size_t size, size_t align,
   BUG();
  }

- /*
-  * We use cache_chain_mutex to ensure a consistent view of
-  * cpu_online_mask as well.  Please see cpuup_callback
-  */
- if (slab_is_available()) {
-  get_online_cpus();
-  mutex_lock(&cache_chain_mutex);
- }
-
  list_for_each_entry(pc, &cache_chain, next) {
   char tmp;
   int res;
@@ -2331,7 +2350,7 @@ kmem_cache_create (const char *name, size_t size, size_t align,
    continue;
   }

-  if (!strcmp(pc->name, name)) {
+  if (!memcg && !strcmp(pc->name, name)) {
    printk(KERN_ERR
           "kmem_cache_create: duplicate cache %s\n", name);
    dump_stack();
@@ -2434,6 +2453,8 @@ 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);
+
+ store_orig_align(cachep, orig_align);
 #if DEBUG

  /*
@@ -2543,7 +2564,12 @@ kmem_cache_create (const char *name, size_t size, size_t align,
  cachep->ctor = ctor;
  cachep->name = name;

+ if (g_cpucache_up >= FULL)
+  mem_cgroup_register_cache(memcg, cachep);
+
+
  if (setup_cpu_cache(cachep, gfp)) {
+  mem_cgroup_release_cache(cachep);
   __kmem_cache_destroy(cachep);
   cachep = NULL;
   goto oops;
@@ -2565,10 +2591,27 @@ oops:
  if (!cachep && (flags & SLAB_PANIC))
   panic("kmem_cache_create(): failed to create slab `%s'\n",
         name);
- if (slab_is_available()) {
-  mutex_unlock(&cache_chain_mutex);
+ 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;
+
+ /*
+  * We use cache_chain_mutex to ensure a consistent view of
+  * cpu_online_mask as well.  Please see cpuup_callback
+  */
+ if (slab_is_available())
+  get_online_cpus();
+ mutex_lock(&cache_chain_mutex);
+ cachep = __kmem_cache_create(NULL, name, size, align, flags, ctor);
+ mutex_unlock(&cache_chain_mutex);
+ if (slab_is_available())
   put_online_cpus();
- }
+
  return cachep;
 }
 EXPORT_SYMBOL(kmem_cache_create);
@@ -2767,6 +2810,8 @@ void kmem_cache_destroy(struct kmem_cache *cachep)
  if (unlikely(cachep->flags & SLAB_DESTROY_BY_RCU))
   rcu_barrier();

+ mem_cgroup_release_cache(cachep);
+
  __kmem_cache_destroy(cachep);
  mutex_unlock(&cache_chain_mutex);
  put_online_cpus();
--
1.7.7.6
```

Subject: [PATCH v3 13/28] slub: create duplicate cache
Posted by Glauber Costa on Fri, 25 May 2012 13:03:33 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                  |   32 ++++++++++++++++++++++++++++++++
 4 files changed, 53 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 dacd1fb..4689034 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 d79740c..0eb9e72 100644
--- a/mm/slub.c
+++ b/mm/slub.c
@@ -4002,6 +4002,38 @@ 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);
+ }
+ /* slub internals is expected to have held a copy of it */
+ 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 v3 14/28] slab: create duplicate cache
Posted by Glauber Costa on Fri, 25 May 2012 13:03:34 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.

[v3: add get_online cpus before the slab mutex]

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

```
---
 mm/slab.c |   44 ++++++++++++++++++++++++++++++++++++++++++++
 1 files changed, 44 insertions(+), 0 deletions(-)

diff --git a/mm/slab.c b/mm/slab.c
index 8bff32a1..e2227de 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
@@ -2616,6 +2618,42 @@ 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);
+
+ get_online_cpus();
+ 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) {
+  kfree(name);
+  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);
+ put_online_cpus();
+ return new;
+}
+#endif /* CONFIG_CGROUP_MEM_RES_CTLR_KMEM */
+
 #if DEBUG
 static void check_irq_off(void)
 {
```

```
@@ -2811,6 +2849,12 @@ void kmem_cache_destroy(struct kmem_cache *cachep)
   rcu_barrier();

  mem_cgroup_release_cache(cachep);
+#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
+ /* memcg cache: free the name string. Doing it here saves us
+  * a pointer to it outside the slab code */
+ if (cachep->memcg_params.id == -1)
+  kfree(cachep->name);
+#endif

  __kmem_cache_destroy(cachep);
  mutex_unlock(&cache_chain_mutex);
--
1.7.7.6
```

Subject: [PATCH v3 15/28] slub: always get the cache from its page in kfree
Posted by Glauber Costa on Fri, 25 May 2012 13:03:35 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

A parent pointer is added to the slub structure, so we can make sure
the freeing comes from either the right slab, or from its rightful
parent.

[v3: added parent testing with VM_BUG_ON]

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

```
---
 include/linux/slab.h     |    3 +++
 include/linux/slub_def.h |   18 ++++++++++++++++++
 mm/slub.c                |    7 ++++++-
 3 files changed, 27 insertions(+), 1 deletions(-)

diff --git a/include/linux/slab.h b/include/linux/slab.h
index e73ef71..724c143 100644
--- a/include/linux/slab.h
+++ b/include/linux/slab.h
@@ -164,6 +164,9 @@ struct mem_cgroup_cache_params {
  size_t orig_align;

 #endif
```

```
+#ifdef CONFIG_DEBUG_VM
+ struct kmem_cache *parent;
+#endif
 };
 #endif

diff --git a/include/linux/slub_def.h b/include/linux/slub_def.h
index 5f5e942..f822ca2 100644
--- a/include/linux/slub_def.h
+++ b/include/linux/slub_def.h
@@ -115,6 +115,24 @@ struct kmem_cache {
  struct kmem_cache_node *node[MAX_NUMNODES];
 };

+static inline void slab_set_parent(struct kmem_cache *s,
+       struct kmem_cache *parent)
+{
+#if defined(CONFIG_CGROUP_MEM_RES_CTLR_KMEM) && defined(CONFIG_DEBUG_VM)
+ s->memcg_params.parent = parent;
+#endif
+}
+
+static inline bool slab_is_parent(struct kmem_cache *s,
+      struct kmem_cache *candidate)
+{
+#if defined(CONFIG_CGROUP_MEM_RES_CTLR_KMEM) && defined(CONFIG_DEBUG_VM)
+ return candidate == s->memcg_params.parent;
+#else
+ return false;
+#endif
+}
+
 /*
  * Kmalloc subsystem.
  */
diff --git a/mm/slub.c b/mm/slub.c
index 0eb9e72..640872f 100644
--- a/mm/slub.c
+++ b/mm/slub.c
@@ -2598,10 +2598,14 @@ redo:
 void kmem_cache_free(struct kmem_cache *s, void *x)
 {
  struct page *page;
+ bool slab_match;

  page = virt_to_head_page(x);

- slab_free(s, page, x, _RET_IP_);
```

```
+ slab_match = (page->slab == s) | slab_is_parent(page->slab, s);
+ VM_BUG_ON(!slab_match);
+
+ slab_free(page->slab, page, x, _RET_IP_);

  trace_kmem_cache_free(_RET_IP_, x);
 }
@@ -4027,6 +4031,7 @@ struct kmem_cache *kmem_cache_dup(struct mem_cgroup *memcg,
   down_write(&slub_lock);
   s->refcount++;
   up_write(&slub_lock);
+  slab_set_parent(new, s);
  }
  /* slub internals is expected to have held a copy of it */
  kfree(name);
--
1.7.7.6
```

Subject: [PATCH v3 16/28] memcg: kmem controller charge/uncharge infrastructure
Posted by Glauber Costa on Fri, 25 May 2012 13:03:36 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.

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

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

```
---
 include/linux/memcontrol.h |   80 +++++++++
 init/Kconfig               |    2 +-
 mm/memcontrol.c            |  386 +++++++++++++++++++++++++++++++++++++++++++-
 3 files changed, 466 insertions(+), 2 deletions(-)

diff --git a/include/linux/memcontrol.h b/include/linux/memcontrol.h
index f93021a..fbc5ba1 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 +477,72 @@ 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)
+  return cachep;
+ if (!current->mm)
+  return cachep;
+ if (in_interrupt())
+  return cachep;
+ if (gfp & __GFP_NOFAIL)
+  return cachep;
+
+ return __mem_cgroup_get_kmem_cache(cachep, gfp);
+}
+
+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)
+  return true;
+ if (!current->mm)
+  return true;
+ if (in_interrupt())
+  return true;
+ if (gfp & __GFP_NOFAIL)
+  return true;
+ return __mem_cgroup_new_kmem_page(page, gfp);
+}
+
+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 4689034..44589fb 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)
@@ -501,9 +520,304 @@ void mem_cgroup_register_cache(struct mem_cgroup *memcg,

 void mem_cgroup_release_cache(struct kmem_cache *cachep)
 {
+ mem_cgroup_flush_cache_create_queue();
  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;
+
+ /* mem_cgroup_get_kmem_cache should have ruled this out */
+ WARN_ON(gfp & __GFP_NOFAIL);
+
+ 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 +5074,11 @@ 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)
@@ -5776,3 +6094,69 @@ 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)  {
+  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, do it as
+   * a temporary condition. But we can't fail. From a kmem/slab
+   * perspective, the cache has already been selected, by
+   * mem_cgroup_get_kmem_cache(), so it is too late to change our
+   * minds
+   */
+  res_counter_charge_nofail(&memcg->res, delta, &fail_res);
+  if (do_swap_account)
+   res_counter_charge_nofail(&memcg->memsw, delta,
+        &fail_res);
+  ret = 0;
+ } 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 v3 17/28] skip memcg kmem allocations in specified code regions
Posted by Glauber Costa on Fri, 25 May 2012 13:03:37 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..0761dda 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;
+ unsigned int 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 44589fb..f3a3812 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;
+
+ current->memcg_kmem_skip_account++;
+}
+
+static void memcg_resume_kmem_account(void)
+{
+ if (!current->mm)
+  return;
+
+ 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 || 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 || 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 v3 18/28] slub: charge allocation to a memcg
Posted by Glauber Costa on Fri, 25 May 2012 13:03:38 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                |   87 +++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++-----
 2 files changed, 112 insertions(+), 14 deletions(-)

diff --git a/include/linux/slub_def.h b/include/linux/slub_def.h
index f822ca2..ba9c68b 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 */
@@ -228,27 +230,54 @@ 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.
@@ -293,7 +322,7 @@ 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;
@@ -326,7 +355,7 @@ 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 640872f..730e69f 100644
--- a/mm/slub.c
+++ b/mm/slub.c
@@ -1283,11 +1283,39 @@ 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;

@@ -1296,13 +1324,29 @@ static struct page *allocate_slab(struct kmem_cache *s, gfp_t flags, int node)

  flags |= s->allocflags;

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

- page = alloc_slab_page(alloc_gfp, node, oo);
  if (unlikely(!page)) {
   oo = s->min;
   /*
@@ -1313,13 +1357,25 @@ 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))) {
@@ -1419,6 +1475,9 @@ 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      \
@@ -2310,6 +2369,9 @@ static __always_inline void *slab_alloc(struct kmem_cache *s,
  if (slab_pre_alloc_hook(s, gfpflags))
   return NULL;

+ rcu_read_lock();
+ s = mem_cgroup_get_kmem_cache(s, gfpflags);
+ rcu_read_unlock();
 redo:

  /*
@@ -3372,9 +3434,15 @@ static void *kmalloc_large_node(size_t size, gfp_t flags, int node)

  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;
 }
@@ -3476,6 +3544,7 @@ 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 v3 19/28] slab: per-memcg accounting of slab caches
Posted by Glauber Costa on Fri, 25 May 2012 13:03:39 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>
Signed-off-by: Suleiman Souhlal <suleiman@google.com>
CC: Christoph Lameter <cl@linux.com>
CC: Pekka Enberg <penberg@cs.helsinki.fi>
CC: Michal Hocko <mhocko@suse.cz>
CC: Kamezawa Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com>
CC: Johannes Weiner <hannes@cmpxchg.org>

```
---
 include/linux/slab_def.h |   62 +++++++++++++++++++++++++++++
 mm/slab.c                |   98 ++++++++++++++++++++++++++++++++++++++++++++----
 2 files changed, 151 insertions(+), 9 deletions(-)

diff --git a/include/linux/slab_def.h b/include/linux/slab_def.h
index 7c0cdd6..ea9054a 100644
--- a/include/linux/slab_def.h
+++ b/include/linux/slab_def.h
@@ -225,4 +225,66 @@ 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 e2227de..16ad229 100644
--- a/mm/slab.c
+++ b/mm/slab.c
@@ -1845,20 +1845,28 @@ 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);

@@ -1874,6 +1882,14 @@ 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.
  */
@@ -1891,6 +1907,8 @@ 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);
@@ -2855,7 +2873,6 @@ void kmem_cache_destroy(struct kmem_cache *cachep)
  if (cachep->memcg_params.id == -1)
   kfree(cachep->name);
 #endif
-
  __kmem_cache_destroy(cachep);
  mutex_unlock(&cache_chain_mutex);
  put_online_cpus();
@@ -3061,8 +3078,10 @@ 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
@@ -3091,8 +3110,10 @@ 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);

@@ -3105,8 +3126,10 @@ 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;
 }

@@ -3867,11 +3890,15 @@ 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);
@@ -3882,6 +3909,10 @@ 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,
@@ -3894,13 +3925,17 @@ 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);
@@ -3913,6 +3948,9 @@ 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,
@@ -4021,9 +4059,33 @@ 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);
@@ -4051,9 +4113,19 @@ 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);
@@ -4322,6 +4394,13 @@ 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
@@ -4354,6 +4433,7 @@ 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 v3 20/28] memcg: disable kmem code when not in use.
Posted by Glauber Costa on Fri, 25 May 2012 13:03:40 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            |   22 +++++++++++++++++++++-
 2 files changed, 24 insertions(+), 2 deletions(-)

diff --git a/include/linux/memcontrol.h b/include/linux/memcontrol.h
index fbc5ba1..bad8ebd 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 f3a3812..f2f1525 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)
 {
@@ -843,6 +853,10 @@ 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);
@@ -4362,8 +4376,13 @@ static int mem_cgroup_write(struct cgroup *cont, struct cftype *cft,
    *
    * But it is not worth the trouble
    */
-   if (!memcg->kmem_accounted && val != RESOURCE_MAX)
+   mutex_lock(&set_limit_mutex);
+   if (!memcg->kmem_accounted && val != RESOURCE_MAX
+       && !memcg->kmem_accounted) {
+    static_key_slow_inc(&mem_cgroup_kmem_enabled_key);
    memcg->kmem_accounted = true;
+   }
+   mutex_unlock(&set_limit_mutex);
  }
 #endif
  else
@@ -5297,6 +5316,7 @@ 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 v3 21/28] memcg: destroy memcg caches
Posted by Glauber Costa on Fri, 25 May 2012 13:03:41 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 bad8ebd..df049e1 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 724c143..c81a5d3 100644
--- a/include/linux/slab.h
+++ b/include/linux/slab.h
@@ -167,6 +167,7 @@ struct mem_cgroup_cache_params {
 #ifdef CONFIG_DEBUG_VM
  struct kmem_cache *parent;
 #endif
+ struct list_head destroyed_list; /* Used when deleting memcg cache */
 };
 #endif

diff --git a/mm/memcontrol.c b/mm/memcontrol.c
index f2f1525..e2ba527 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;
@@ -857,6 +938,10 @@ 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);
@@ -4136,6 +4221,7 @@ 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);

@@ -4175,8 +4261,10 @@ 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;
@@ -5523,6 +5611,7 @@ 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 16ad229..59f1027 100644
--- a/mm/slab.c
+++ b/mm/slab.c
@@ -1885,8 +1885,9 @@ 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 730e69f..eb0ff97 100644
--- a/mm/slub.c
+++ b/mm/slub.c
@@ -1296,8 +1296,11 @@ 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 v3 22/28] memcg/slub: shrink dead caches
Posted by Glauber Costa on Fri, 25 May 2012 13:03:42 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 |    8 +++++++
 mm/memcontrol.c          |   49 ++++++++++++++++++++++++++++++++++++++++++++--
 mm/slub.c                |    1 +
 4 files changed, 59 insertions(+), 3 deletions(-)

diff --git a/include/linux/slab.h b/include/linux/slab.h
index c81a5d3..25f073e 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 ba9c68b..c1428ee 100644
--- a/include/linux/slub_def.h
+++ b/include/linux/slub_def.h
@@ -135,6 +135,14 @@ static inline bool slab_is_parent(struct kmem_cache *s,
 #endif
 }

+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 e2ba527..e2576c5 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)
   ida_simple_remove(&cache_types, cachep->memcg_params.id);
 }

+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 eb0ff97..f5fc10c 100644
--- a/mm/slub.c
+++ b/mm/slub.c
@@ -2658,6 +2658,7 @@ 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 v3 23/28] slab: Track all the memcg children of a kmem_cache.
Posted by Glauber Costa on Fri, 25 May 2012 13:03:43 GMT
View Forum Message <> Reply to Message

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.

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

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

```
---
 include/linux/memcontrol.h |    1 +
 include/linux/slab.h       |    1 +
 mm/memcontrol.c            |   13 ++++++++-
 mm/slab.c                  |   61 ++++++++++++++++++++++++++++++++++++++++++++-
 4 files changed, 73 insertions(+), 3 deletions(-)

diff --git a/include/linux/memcontrol.h b/include/linux/memcontrol.h
index df049e1..63113de 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 25f073e..714aeab 100644
--- a/include/linux/slab.h
+++ b/include/linux/slab.h
@@ -172,6 +172,7 @@ struct mem_cgroup_cache_params {
  struct kmem_cache *parent;
 #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 e2576c5..08f3c3e 100644
--- a/mm/memcontrol.c
+++ b/mm/memcontrol.c
@@ -542,10 +542,11 @@ void mem_cgroup_register_cache(struct mem_cgroup *memcg,

  cachep->memcg_params.memcg = memcg;

- if (!memcg)
+ if (!memcg) {
   id = ida_simple_get(&cache_types, 0, MAX_KMEM_CACHE_TYPES,
         GFP_KERNEL);
- else
+  INIT_LIST_HEAD(&cachep->memcg_params.sibling_list);
+ } else
   INIT_LIST_HEAD(&cachep->memcg_params.destroyed_list);
  cachep->memcg_params.id = id;
 }
@@ -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 59f1027..cb409ae 100644
--- a/mm/slab.c
+++ b/mm/slab.c
@@ -2661,6 +2661,8 @@ struct kmem_cache *kmem_cache_dup(struct mem_cgroup *memcg,
   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))
```

```
@@ -2829,6 +2831,33 @@ int kmem_cache_shrink(struct kmem_cache *cachep)
 }
 EXPORT_SYMBOL(kmem_cache_shrink);

+static void kmem_cache_destroy_memcg_children(struct kmem_cache *cachep)
+{
+#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
+ struct kmem_cache *c;
+ struct mem_cgroup_cache_params *p, *tmp;
+ int id = cachep->memcg_params.id;
+
+ if (id == -1)
+  return;
+
+ 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 (WARN_ON(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 */
+}
+
 /**
  * kmem_cache_destroy - delete a cache
  * @cachep: the cache to destroy
@@ -2849,6 +2878,9 @@ void kmem_cache_destroy(struct kmem_cache *cachep)
 {
  BUG_ON(!cachep || in_interrupt());

+ /* Destroy all the children caches if we aren't a memcg cache */
+ kmem_cache_destroy_memcg_children(cachep);
+
  /* Find the cache in the chain of caches. */
  get_online_cpus();
  mutex_lock(&cache_chain_mutex);
@@ -2869,6 +2901,7 @@ void kmem_cache_destroy(struct kmem_cache *cachep)

  mem_cgroup_release_cache(cachep);
 #ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
```

```
+ list_del(&cachep->memcg_params.sibling_list);
  /* memcg cache: free the name string. Doing it here saves us
   * a pointer to it outside the slab code */
  if (cachep->memcg_params.id == -1)
@@ -4243,7 +4276,7 @@ static void do_ccupdate_local(void *info)
 }

 /* Always called with the cache_chain_mutex held */
-static int do_tune_cpucache(struct kmem_cache *cachep, int limit,
+static int __do_tune_cpucache(struct kmem_cache *cachep, int limit,
     int batchcount, int shared, gfp_t gfp)
 {
  struct ccupdate_struct *new;
@@ -4286,6 +4319,32 @@ static int do_tune_cpucache(struct kmem_cache *cachep, int limit,
  return alloc_kmemlist(cachep, gfp);
 }

+static int do_tune_cpucache(struct kmem_cache *cachep, int limit,
+    int batchcount, int shared, gfp_t gfp)
+{
+ int ret;
+#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
+ struct kmem_cache *c;
+ struct mem_cgroup_cache_params *p;
+#endif
+
+ ret = __do_tune_cpucache(cachep, limit, batchcount, shared, gfp);
+#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
+ if (g_cpucache_up < FULL)
+  return ret;
+
+ if ((ret < 0) || (cachep->memcg_params.id == -1))
+  return ret;
+
+ list_for_each_entry(p, &cachep->memcg_params.sibling_list, sibling_list) {
+  c = container_of(p, struct kmem_cache, memcg_params);
+  /* return value determined by the parent cache only */
+  __do_tune_cpucache(c, limit, batchcount, shared, gfp);
+ }
+#endif
+ return ret;
+}
+
 /* Called with cache_chain_mutex held always */
 static int enable_cpucache(struct kmem_cache *cachep, gfp_t gfp)
 {
--
1.7.7.6
```

Subject: [PATCH v3 24/28] memcg: Per-memcg memory.kmem.slabinfo file.
Posted by Glauber Costa on Fri, 25 May 2012 13:03:44 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 714aeab..f2f51d8 100644
--- a/include/linux/slab.h
+++ b/include/linux/slab.h
@@ -333,6 +333,7 @@ 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 08f3c3e..3e99c69 100644
--- a/mm/memcontrol.c
+++ b/mm/memcontrol.c
@@ -5229,6 +5229,19 @@ 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",
@@ -5253,6 +5266,10 @@ 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 cb409ae..2f9cf92 100644
--- a/mm/slab.c
+++ b/mm/slab.c
@@ -4550,21 +4550,26 @@ 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)
@@ -4576,31 +4581,43 @@ 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;
@@ -4608,12 +4625,12 @@ 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;
@@ -4647,6 +4664,30 @@ 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 f5fc10c..c8a8cab 100644
--- a/mm/slub.c
+++ b/mm/slub.c
@@ -4110,6 +4110,11 @@ 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 v3 25/28] slub: create slabinfo file for memcg
Posted by Glauber Costa on Fri, 25 May 2012 13:03:45 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 c8a8cab..4c29e5f 100644
--- a/mm/slub.c
+++ b/mm/slub.c
@@ -4113,6 +4113,33 @@ 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 v3 26/28] slub: track all children of a kmem cache
Posted by Glauber Costa on Fri, 25 May 2012 13:03:46 GMT

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 |   64 +++++++++++++++++++++++++++++++++++++++++++++++++++++--------
 1 files changed, 55 insertions(+), 9 deletions(-)

diff --git a/mm/slub.c b/mm/slub.c
index 4c29e5f..8151353 100644
--- a/mm/slub.c
+++ b/mm/slub.c
@@ -3254,6 +3254,54 @@ static inline int kmem_cache_close(struct kmem_cache *s)
  return 0;
 }

+static 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);
+}
+
+static void kmem_cache_destroy_memcg_children(struct kmem_cache *s)
```

```
+{
+#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
+ struct mem_cgroup_cache_params *p, *tmp, *this;
+ struct kmem_cache *c;
+ int id = s->memcg_params.id;
+
+ /* Not a parent cache */
+ if (id == -1)
+  return ;
+
+ 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
+}
 /*
  * Close a cache and release the kmem_cache structure
  * (must be used for caches created using kmem_cache_create)
@@ -3261,19 +3309,15 @@ static inline int kmem_cache_close(struct kmem_cache *s)
 void kmem_cache_destroy(struct kmem_cache *s)
 {
  down_write(&slub_lock);
+ kmem_cache_destroy_memcg_children(s);
  s->refcount--;
  if (!s->refcount) {
   list_del(&s->list);
-  mem_cgroup_release_cache(s);
+#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
+  list_del(&s->memcg_params.sibling_list);
+#endif
   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);
 }
@@ -4102,6 +4146,8 @@ 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);
   slab_set_parent(new, s);
--
1.7.7.6
```

Subject: [PATCH v3 27/28] memcg: propagate kmem limiting information to children
Posted by Glauber Costa on Fri, 25 May 2012 13:03:47 GMT

View Forum Message <> Reply to Message

The current memcg slab cache management fails to present satisfatory hierarchical
behavior in the following scenario:

-> /cgroups/memory/A/B/C

* kmem limit set at A
* A and B empty taskwise
* bash in C does find /

Because kmem_accounted is a boolean that was not set for C, no accounting
would be done. This is, however, not what we expect.

The basic idea, is that when a cgroup is limited, we walk the tree
upwards (something Kame and I already thought about doing for other purposes),
and make sure that we store the information about the parent being limited in
kmem_accounted (that is turned into a bitmap: two booleans would not be space
efficient). The code for that is taken from sched/core.c. My reasons for not
putting it into a common place is to dodge the type issues that would arise
from a common implementation between memcg and the scheduler - but I think

that it should ultimately happen, so if you want me to do it now, let me
know.

We do the reverse operation when a formerly limited cgroup becomes unlimited.

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/memcontrol.c |  147 +++++++++++++++++++++++++++++++++++++++++++++++++++++------
 1 files changed, 131 insertions(+), 16 deletions(-)

diff --git a/mm/memcontrol.c b/mm/memcontrol.c
index 3e99c69..7572cb1 100644
--- a/mm/memcontrol.c
+++ b/mm/memcontrol.c
@@ -259,6 +259,9 @@ struct mem_cgroup {
   * the counter to account for kernel memory usage.
   */
  struct res_counter kmem;
+
+ struct list_head children;
+ struct list_head siblings;
  /*
   * Per cgroup active and inactive list, similar to the
   * per zone LRU lists.
@@ -274,7 +277,11 @@ struct mem_cgroup {
   * Should the accounting and control be hierarchical, per subtree?
   */
  bool use_hierarchy;
- bool kmem_accounted;
+ /*
+  * bit0: accounted by this cgroup
+  * bit1: accounted by a parent.
+  */
+ volatile unsigned long kmem_accounted;

  bool  oom_lock;
  atomic_t under_oom;
@@ -332,6 +339,9 @@ struct mem_cgroup {
 #endif
 };

+#define KMEM_ACCOUNTED_THIS 0
```

```
+#define KMEM_ACCOUNTED_PARENT 1
+
 int memcg_css_id(struct mem_cgroup *memcg)
 {
  return css_id(&memcg->css);
@@ -474,7 +484,7 @@ void sock_release_memcg(struct sock *sk)

 static void disarm_static_keys(struct mem_cgroup *memcg)
 {
- if (memcg->kmem_accounted)
+ if (test_bit(KMEM_ACCOUNTED_THIS, &memcg->kmem_accounted))
   static_key_slow_dec(&mem_cgroup_kmem_enabled_key);
  /*
   * This check can't live in kmem destruction function,
@@ -4472,6 +4482,110 @@ static ssize_t mem_cgroup_read(struct cgroup *cont, struct cftype
*cft,
  len = scnprintf(str, sizeof(str), "%llu\n", (unsigned long long)val);
  return simple_read_from_buffer(buf, nbytes, ppos, str, len);
 }
+
+#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
+typedef int (*memcg_visitor)(struct mem_cgroup*, void *);
+
+/*
+ * This is mostly "inspired" by the code in sched/core.c. I decided to copy it,
+ * instead of factoring it, because of all the typing issues we'd run into.
+ * In particular, grabbing the parent is very different for memcg, because we
+ * may or may not have hierarchy, while cpu cgroups always do. That would lead
+ * to either indirect calls - this is not a fast path for us, but can be for
+ * the scheduler - or a big and ugly macro.
+ *
+ * If we ever get rid of hierarchy, we could iterate over struct cgroup, and
+ * then it would cease to be a problem.
+ */
+int walk_tree_from(struct mem_cgroup *from,
+     memcg_visitor down, memcg_visitor up, void *data)
+{
+ struct mem_cgroup *parent, *child;
+ int ret;
+
+
+ parent = from;
+down:
+ ret = (*down)(parent, data);
+ if (ret)
+  goto out;
+
+ list_for_each_entry_rcu(child, &parent->children, siblings) {
```

```
+  parent = child;
+  goto down;
+
+up:
+  continue;
+ }
+ ret = (*up)(parent, data);
+ if (ret || parent == from)
+  goto out;
+
+ child = parent;
+ parent = parent_mem_cgroup(parent);
+ if (parent)
+  goto up;
+out:
+ return ret;
+}
+
+static int memcg_nop(struct mem_cgroup *memcg, void *data)
+{
+ return 0;
+}
+
+static int memcg_parent_account(struct mem_cgroup *memcg, void *data)
+{
+ if (memcg == data)
+  return 0;
+
+ set_bit(KMEM_ACCOUNTED_PARENT, &memcg->kmem_accounted);
+ return 0;
+}
+
+static int memcg_parent_no_account(struct mem_cgroup *memcg, void *data)
+{
+ if (memcg == data)
+  return 0;
+
+ clear_bit(KMEM_ACCOUNTED_PARENT, &memcg->kmem_accounted);
+ /*
+  * Stop propagation if we are accounted: our children should
+  * be parent-accounted
+  */
+ return test_bit(KMEM_ACCOUNTED_THIS, &memcg->kmem_accounted);
+}
+
+static void mem_cgroup_update_kmem_limit(struct mem_cgroup *memcg, u64 val)
+{
+ mutex_lock(&set_limit_mutex);
+ if (!test_and_set_bit(KMEM_ACCOUNTED_THIS, &memcg->kmem_accounted) &&
+  val != RESOURCE_MAX) {
+
+  /*
+   * 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
+   */
+  static_key_slow_inc(&mem_cgroup_kmem_enabled_key);
+
+  rcu_read_lock();
+  walk_tree_from(memcg, memcg_parent_account, memcg_nop, memcg);
+  rcu_read_unlock();
+ } else if (test_and_clear_bit(KMEM_ACCOUNTED_THIS, &memcg->kmem_accounted)
+  && val == RESOURCE_MAX) {
+
+  rcu_read_lock();
+  walk_tree_from(memcg, memcg_parent_no_account,
+         memcg_nop, memcg);
+  rcu_read_unlock();
+ }
+
+ mutex_unlock(&set_limit_mutex);
+}
+#endif
 /*
  * The user of this function is...
  * RES_LIMIT.
@@ -4509,20 +4623,8 @@ static int mem_cgroup_write(struct cgroup *cont, struct cftype *cft,
    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
-    */
-   mutex_lock(&set_limit_mutex);
-   if (!memcg->kmem_accounted && val != RESOURCE_MAX
-       && !memcg->kmem_accounted) {
-    static_key_slow_inc(&mem_cgroup_kmem_enabled_key);
-    memcg->kmem_accounted = true;
-   }
-   mutex_unlock(&set_limit_mutex);
```

```
+   mem_cgroup_update_kmem_limit(memcg, val);
+   break;
   }
 #endif
   else
@@ -5592,6 +5694,8 @@ err_cleanup:

 }

+static DEFINE_MUTEX(memcg_list_mutex);
+
 static struct cgroup_subsys_state * __ref
 mem_cgroup_create(struct cgroup *cont)
 {
@@ -5607,6 +5711,7 @@ mem_cgroup_create(struct cgroup *cont)
   if (alloc_mem_cgroup_per_zone_info(memcg, node))
    goto free_out;

+ INIT_LIST_HEAD(&memcg->children);
  /* root ? */
  if (cont->parent == NULL) {
   int cpu;
@@ -5645,6 +5750,10 @@ mem_cgroup_create(struct cgroup *cont)
    * mem_cgroup(see mem_cgroup_put).
    */
   mem_cgroup_get(parent);
+
+  mutex_lock(&memcg_list_mutex);
+  list_add_rcu(&memcg->siblings, &parent->children);
+  mutex_unlock(&memcg_list_mutex);
  } else {
   res_counter_init(&memcg->res, NULL);
   res_counter_init(&memcg->memsw, NULL);
@@ -5687,9 +5796,15 @@ static int mem_cgroup_pre_destroy(struct cgroup *cont)
 static void mem_cgroup_destroy(struct cgroup *cont)
 {
  struct mem_cgroup *memcg = mem_cgroup_from_cont(cont);
+ struct mem_cgroup *parent = parent_mem_cgroup(memcg);

  kmem_cgroup_destroy(memcg);

+ mutex_lock(&memcg_list_mutex);
+ if (parent)
+  list_del_rcu(&memcg->siblings);
+ mutex_unlock(&memcg_list_mutex);
+
  mem_cgroup_put(memcg);
 }
```

```
--
1.7.7.6
```

Subject: [PATCH v3 28/28] Documentation: add documentation for slab tracker for memcg
Posted by Glauber Costa on Fri, 25 May 2012 13:03:48 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 v3 00/28] kmem limitation for memcg
Posted by Michal Hocko on Fri, 25 May 2012 13:34:41 GMT

View Forum Message <> Reply to Message

On Fri 25-05-12 17:03:20, Glauber Costa wrote:
> I believe some of the early patches here are already in some trees around.
> I don't know who should pick this, so if everyone agrees with what's in here,
> please just ack them and tell me which tree I should aim for (-mm? Hocko's?)
> and I'll rebase it.

memcg-devel tree is only to make development easier. Everything that
applies on top of this tree should be applicable to both -mm and
linux-next.
So the patches should go via traditional Andrew's channel.
--
Michal Hocko
SUSE Labs
SUSE LINUX s.r.o.
Lihovarska 1060/12
190 00 Praha 9
Czech Republic

Subject: Re: [PATCH v3 00/28] kmem limitation for memcg
Posted by Christoph Lameter on Fri, 25 May 2012 14:34:43 GMT
View Forum Message <> Reply to Message

On Fri, 25 May 2012, Michal Hocko wrote:

> On Fri 25-05-12 17:03:20, Glauber Costa wrote:
> > I believe some of the early patches here are already in some trees around.
> > I don't know who should pick this, so if everyone agrees with what's in here,
> > please just ack them and tell me which tree I should aim for (-mm? Hocko's?)
> > and I'll rebase it.
>
> memcg-devel tree is only to make development easier. Everything that
> applies on top of this tree should be applicable to both -mm and
> linux-next.
> So the patches should go via traditional Andrew's channel.

It would be best to merge these with my patchset to extract common code
from the allocators. The modifications of individual slab allocators would
then be not necessary anymore and it would save us a lot of work.

Subject: Re: [PATCH v3 00/28] kmem limitation for memcg
Posted by Glauber Costa on Mon, 28 May 2012 08:32:28 GMT
View Forum Message <> Reply to Message

On 05/25/2012 06:34 PM, Christoph Lameter wrote:

> On Fri, 25 May 2012, Michal Hocko wrote:
>
>> On Fri 25-05-12 17:03:20, Glauber Costa wrote:
>>> I believe some of the early patches here are already in some trees around.
>>> I don't know who should pick this, so if everyone agrees with what's in here,
>>> please just ack them and tell me which tree I should aim for (-mm? Hocko's?)
>>> and I'll rebase it.
>>
>> memcg-devel tree is only to make development easier. Everything that
>> applies on top of this tree should be applicable to both -mm and
>> linux-next.
>> So the patches should go via traditional Andrew's channel.
>
> It would be best to merge these with my patchset to extract common code
> from the allocators. The modifications of individual slab allocators would
> then be not necessary anymore and it would save us a lot of work.
>
Some of them would not, some of them would still be. But also please
note that the patches here that deal with differences between allocators
are usually the low hanging fruits compared to the rest.

I agree that long term it not only better, but inevitable, if we are
going to merge both.

But right now, I think we should agree with the implementation itself -
so if you have any comments on how I am handling these, I'd be happy to
hear. Then we can probably set up a tree that does both, or get your
patches merged and I'll rebase, etc.

Subject: Re: [PATCH v3 12/28] slab: pass memcg parameter to kmem_cache_create
Posted by Christoph Lameter on Tue, 29 May 2012 14:27:52 GMT
View Forum Message <> Reply to Message

On Fri, 25 May 2012, Glauber Costa wrote:

```
> index 06e4a3e..7c0cdd6 100644
> --- a/include/linux/slab_def.h
> +++ b/include/linux/slab_def.h
> @@ -102,6 +102,13 @@ struct kmem_cache {
>    */
>  };
>
> +static inline void store_orig_align(struct kmem_cache *cachep, int orig_align)
> +{
> +#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
> + cachep->memcg_params.orig_align = orig_align;
> +#endif
> +}
> +
```

Why do you need to store the original alignment? Is the calculated
alignment not enough?

```
> +++ b/mm/slab.c
> @@ -1729,6 +1729,31 @@ void __init kmem_cache_init_late(void)
>    */
>  }
>
> +static int __init memcg_slab_register_all(void)
> +{
> +#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
> + struct kmem_cache *cachep;
> + struct cache_sizes *sizes;
> +
> + 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);
> +
> + mutex_unlock(&cache_chain_mutex);
> +#endif /* CONFIG_CGROUP_MEM_RES_CTLR_KMEM */
> + return 0;
> +}
```

Ok this only duplicates the kmalloc arrays. Why not the others?

```
> @@ -2331,7 +2350,7 @@ kmem_cache_create (const char *name, size_t size, size_t align,
>     continue;
>    }
>
> -  if (!strcmp(pc->name, name)) {
> +  if (!memcg && !strcmp(pc->name, name)) {
>     printk(KERN_ERR
>            "kmem_cache_create: duplicate cache %s\n", name);
>     dump_stack();
```

This implementation means that duplicate cache detection will no longer
work within a cgroup?

```
> @@ -2543,7 +2564,12 @@ kmem_cache_create (const char *name, size_t size, size_t align,
>   cachep->ctor = ctor;
>   cachep->name = name;
>
> + if (g_cpucache_up >= FULL)
> +  mem_cgroup_register_cache(memcg, cachep);
```

What happens if a cgroup was active during creation of slab xxy but
then a process running in a different cgroup uses that slab to allocate
memory? Is it charged to the first cgroup?

Subject: Re: [PATCH v3 13/28] slub: create duplicate cache
Posted by Christoph Lameter on Tue, 29 May 2012 14:36:55 GMT

View Forum Message <> Reply to Message

On Fri, 25 May 2012, Glauber Costa wrote:

```
> index dacd1fb..4689034 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;
> +}
```

Function allocates a string that is supposed to be disposed of by the
caller. That needs to be documented and maybe even the name needs to

reflect that.

```
> --- a/mm/slub.c
> +++ b/mm/slub.c
> @@ -4002,6 +4002,38 @@ 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);
```

Hmmm... A full duplicate of the slab cache? We may have many sparsely
used portions of the per node and per cpu structure as a result.

```
> +  * 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);
> + }
```

Why do you need to increase the refcount? You made a full copy right?

Subject: Re: [PATCH v3 15/28] slub: always get the cache from its page in kfree
Posted by Christoph Lameter on Tue, 29 May 2012 14:42:03 GMT

View Forum Message <> Reply to Message

On Fri, 25 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

Yes but the lookup of the page struct also costs some cycles. SLAB in
!NUMA mode and SLOB avoid these lookups and can improve their freeing
speed because of that.

```
> diff --git a/mm/slub.c b/mm/slub.c
> index 0eb9e72..640872f 100644
> --- a/mm/slub.c
> +++ b/mm/slub.c
> @@ -2598,10 +2598,14 @@ redo:
>  void kmem_cache_free(struct kmem_cache *s, void *x)
>  {
>   struct page *page;
> + bool slab_match;
>
>   page = virt_to_head_page(x);
>
> - slab_free(s, page, x, _RET_IP_);
> + slab_match = (page->slab == s) | slab_is_parent(page->slab, s);
> + VM_BUG_ON(!slab_match);
```

Why add a slab_match bool if you do not really need it?

Subject: Re: [PATCH v3 16/28] memcg: kmem controller charge/uncharge infrastructure
Posted by Christoph Lameter on Tue, 29 May 2012 14:47:38 GMT
View Forum Message <> Reply to Message

On Fri, 25 May 2012, Glauber Costa wrote:

```
> --- 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
```

Ok so SLOB is not supported at all.

Subject: Re: [PATCH v3 18/28] slub: charge allocation to a memcg
Posted by Christoph Lameter on Tue, 29 May 2012 14:51:20 GMT

View Forum Message <> Reply to Message

On Fri, 25 May 2012, Glauber Costa wrote:

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

I am wondering why you need all the other patches. The simplest approach would just to hook into page allocation and freeing from the slab allocators as done here and charge to the currently active cgroup. This avoids all the duplication of slab caches and per node as well as per cpu structures. A certain degree of fuzziness cannot be avoided given that objects are cached and may be served to multiple cgroups. If that can be tolerated then the rest would be just like this patch which could be made more simple and non intrusive.

Subject: Re: [PATCH v3 19/28] slab: per-memcg accounting of slab caches
Posted by Christoph Lameter on Tue, 29 May 2012 14:52:59 GMT
View Forum Message <> Reply to Message

On Fri, 25 May 2012, Glauber Costa wrote:

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

Ok so a requirement is to support tracking of individual slab objects to cgroups? That is going to be quite expensive since it will touch the hotpaths.

Subject: Re: [PATCH v3 00/28] kmem limitation for memcg
Posted by Christoph Lameter on Tue, 29 May 2012 15:07:39 GMT
View Forum Message <> Reply to Message

On Mon, 28 May 2012, Glauber Costa wrote:

> > It would be best to merge these with my patchset to extract common code
> > from the allocators. The modifications of individual slab allocators would
> > then be not necessary anymore and it would save us a lot of work.
> >
> Some of them would not, some of them would still be. But also please note that
> the patches here that deal with differences between allocators are usually the
> low hanging fruits compared to the rest.
>
> I agree that long term it not only better, but inevitable, if we are going to
> merge both.
>

> But right now, I think we should agree with the implementation itself - so if
> you have any comments on how I am handling these, I'd be happy to hear. Then
> we can probably set up a tree that does both, or get your patches merged and
> I'll rebase, etc.

Just looked over the patchset and its quite intrusive. I have never been
fond of cgroups (IMHO hardware needs to be partitioned at physical
boundaries) so I have not too much insight into what is going on in that
area.

The idea to just duplicate the caches leads to some weird stuff like the
refcounting and the recovery of the arguments used during slab creation.

I think it may be simplest to only account for the pages used by a slab in
a memcg. That code could be added to the functions in the slab allocators
that interface with the page allocators. Those are not that performance
critical and would do not much harm.

If you need per object accounting then the cleanest solution would be to
duplicate the per node arrays per memcg (or only the statistics) and have
the kmem_cache structure only once in memory.

Its best if information is only in one place for design and for performance.

Subject: Re: [PATCH v3 00/28] kmem limitation for memcg
Posted by Glauber Costa on Tue, 29 May 2012 15:44:54 GMT

View Forum Message <> Reply to Message

On 05/29/2012 07:07 PM, Christoph Lameter wrote:
> On Mon, 28 May 2012, Glauber Costa wrote:
>
>>> It would be best to merge these with my patchset to extract common code
>>> from the allocators. The modifications of individual slab allocators would
>>> then be not necessary anymore and it would save us a lot of work.
>>>
>> Some of them would not, some of them would still be. But also please note that
>> the patches here that deal with differences between allocators are usually the
>> low hanging fruits compared to the rest.
>>
>> I agree that long term it not only better, but inevitable, if we are going to
>> merge both.
>>
>> But right now, I think we should agree with the implementation itself - so if
>> you have any comments on how I am handling these, I'd be happy to hear. Then
>> we can probably set up a tree that does both, or get your patches merged and
>> I'll rebase, etc.
>

> Just looked over the patchset and its quite intrusive.

Thank you very much, Christoph, appreciate it.

> I have never been
> fond of cgroups (IMHO hardware needs to be partitioned at physical
> boundaries) so I have not too much insight into what is going on in that
> area.

There is certainly a big market for that, and certainly a big market for
what we're doing as well. So there are users interested in Containers
technology, and I don't really see it as "partitioning it here" vs
"partitioning there". It's just different.

Moreover, not everyone doing cgroups are doing containers. Some people
are isolating a service, or a paticular job.

I agree it is an intrusive change, but it used to be even more. I did my
best to diminish its large spread.

> The idea to just duplicate the caches leads to some weird stuff like the
> refcounting and the recovery of the arguments used during slab creation.

The refcounting is only needed so we are sure the parent cache won't go
away without the child caches going away. I can try to find a better way
to do that, specifically.

>
> I think it may be simplest to only account for the pages used by a slab in
> a memcg. That code could be added to the functions in the slab allocators
> that interface with the page allocators. Those are not that performance
> critical and would do not much harm.

No, I don't think so. Well, accounting the page is easy, but when we do
a new allocation, we need to match a process to its correspondent page.
This will likely lead to flushing the internal cpu caches of the slub,
for instance, hurting performance. That is because once we allocate a
page, all objects on that page need to belong to the same cgroup.

Also, you talk about intrusiveness, accounting pages is a lot more
intrusive, since then you need to know a lot about the internal
structure of each cache. Having the cache replicated has exactly the
effect of isolating it better.

I of course agree this is no walk in the park, but accounting something
that is internal to the cache, and that each cache will use and organize
in its own private way, doesn't make it any better.

> If you need per object accounting then the cleanest solution would be to
> duplicate the per node arrays per memcg (or only the statistics) and have
> the kmem_cache structure only once in memory.

No, it's all per-page. Nothing here is per-object, maybe you
misunderstood something?

Subject: Re: [PATCH v3 12/28] slab: pass memcg parameter to kmem_cache_create
Posted by Glauber Costa on Tue, 29 May 2012 15:50:39 GMT
View Forum Message <> Reply to Message

On 05/29/2012 06:27 PM, Christoph Lameter wrote:
> On Fri, 25 May 2012, Glauber Costa wrote:
>
```
>> index 06e4a3e..7c0cdd6 100644
>> --- a/include/linux/slab_def.h
>> +++ b/include/linux/slab_def.h
>> @@ -102,6 +102,13 @@ struct kmem_cache {
>>     */
>>   };
>>
>> +static inline void store_orig_align(struct kmem_cache *cachep, int orig_align)
>> +{
>> +#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
>> + cachep->memcg_params.orig_align = orig_align;
>> +#endif
>> +}
>> +
```
>
> Why do you need to store the original alignment? Is the calculated
> alignment not enough?

I think this one can go. You are right.

```
>> +++ b/mm/slab.c
>> @@ -1729,6 +1729,31 @@ void __init kmem_cache_init_late(void)
>>     */
>>   }
>>
>> +static int __init memcg_slab_register_all(void)
>> +{
>> +#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
>> + struct kmem_cache *cachep;
>> + struct cache_sizes *sizes;
>> +
>> + 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);
>> +
>> + mutex_unlock(&cache_chain_mutex);
>> +#endif /* CONFIG_CGROUP_MEM_RES_CTLR_KMEM */
>> + return 0;
>> +}
>
> Ok this only duplicates the kmalloc arrays. Why not the others?
```

It does duplicate the others.

First it does a while look on the kmalloc caches, then a
list_for_each_entry in the rest. You probably missed it.

```
>> @@ -2331,7 +2350,7 @@ kmem_cache_create (const char *name, size_t size, size_t align,
>>      continue;
>>     }
>>
>> -  if (!strcmp(pc->name, name)) {
>> +  if (!memcg&&  !strcmp(pc->name, name)) {
>>      printk(KERN_ERR
>>             "kmem_cache_create: duplicate cache %s\n", name);
>>      dump_stack();
>
> This implementation means that duplicate cache detection will no longer
> work within a cgroup?
```

For the slab, yes. For the slub, I check to see if they belong to the
same memcg.

That said, this can and should be fixed here too, thanks for spotting.

```
>> @@ -2543,7 +2564,12 @@ kmem_cache_create (const char *name, size_t size, size_t align,
>>    cachep->ctor = ctor;
>>    cachep->name = name;
>>
>> + if (g_cpucache_up>= FULL)
```

>> + mem_cgroup_register_cache(memcg, cachep);
>
> What happens if a cgroup was active during creation of slab xxy but
> then a process running in a different cgroup uses that slab to allocate
> memory? Is it charged to the first cgroup?

I don't see this situation ever happening. kmem_cache_create, when
called directly, will always create a global cache. It doesn't matter
which cgroups are or aren't active at this time or any other. We create
copies per-cgroup, but we create it lazily, when someone will touch it.

At that point, which cache will be used depend on which process is using
it.

Subject: Re: [PATCH v3 13/28] slub: create duplicate cache
Posted by Glauber Costa on Tue, 29 May 2012 15:56:23 GMT

View Forum Message <> Reply to Message

On 05/29/2012 06:36 PM, Christoph Lameter wrote:
> On Fri, 25 May 2012, Glauber Costa wrote:
>
```
>> index dacd1fb..4689034 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;
>> +}
```
>
> Function allocates a string that is supposed to be disposed of by the
> caller. That needs to be documented and maybe even the name needs to

> reflect that.

Okay, I can change it.

```
>> --- a/mm/slub.c
>> +++ b/mm/slub.c
>> @@ -4002,6 +4002,38 @@ 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);
>
```

> Hmmm... A full duplicate of the slab cache? We may have many sparsely
> used portions of the per node and per cpu structure as a result.

I've already commented on patch 0, but I will repeat it here. This
approach leads to more fragmentation, yes, but this is exactly to be
less intrusive.

With a full copy, all I need to do is:

1) relay the allocation to the right cache.
2) account for a new page when it is needed.

How does the cache work from inside? I don't care.

Accounting pages seems just crazy to me. If new allocators come in the
future, organizing the pages in a different way, instead of patching it
here and there, we need to totally rewrite this.

If those allocators happen to depend on a specific placement for
performance, then we're destroying this as well too.

```
>
>> +  * 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);
>> + }
>
```

> Why do you need to increase the refcount? You made a full copy right?

Yes, but I don't want this copy to go away while we have other caches
around.

So, in the memcg internals, I used a different reference counter, to
avoid messing with this one. I could use that, and leave the original
refcnt alone. Would you prefer this?

Subject: Re: [PATCH v3 15/28] slub: always get the cache from its page in kfree
Posted by Glauber Costa on Tue, 29 May 2012 15:59:47 GMT
View Forum Message <> Reply to Message

On 05/29/2012 06:42 PM, Christoph Lameter wrote:
> On Fri, 25 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
>
> Yes but the lookup of the page struct also costs some cycles. SLAB in
> !NUMA mode and SLOB avoid these lookups and can improve their freeing
> speed because of that.

But for our case, I don't really see a way around. What I can do, is
wrap it further, so when we're not using it, code goes exactly the same
way as before, instead of always calculating the page. Would it be better?

```
>> diff --git a/mm/slub.c b/mm/slub.c
>> index 0eb9e72..640872f 100644
>> --- a/mm/slub.c
>> +++ b/mm/slub.c
>> @@ -2598,10 +2598,14 @@ redo:
>>   void kmem_cache_free(struct kmem_cache *s, void *x)
>>   {
>>    struct page *page;
>> + bool slab_match;
```

>>
>>   page = virt_to_head_page(x);
>>
>> - slab_free(s, page, x, _RET_IP_);
>> + slab_match = (page->slab == s) | slab_is_parent(page->slab, s);
>> + VM_BUG_ON(!slab_match);
>
> Why add a slab_match bool if you do not really need it?

style. I find aux variables a very human readable way to deal with the
80-col limitation.

Subject: Re: [PATCH v3 16/28] memcg: kmem controller charge/uncharge infrastructure
Posted by Glauber Costa on Tue, 29 May 2012 16:00:23 GMT
View Forum Message <> Reply to Message

On 05/29/2012 06:47 PM, Christoph Lameter wrote:
> On Fri, 25 May 2012, Glauber Costa wrote:
>
>> --- 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
>
> Ok so SLOB is not supported at all.
>
Yes, at least I see no reason to.

Subject: Re: [PATCH v3 00/28] kmem limitation for memcg
Posted by Christoph Lameter on Tue, 29 May 2012 16:01:03 GMT
View Forum Message <> Reply to Message

On Tue, 29 May 2012, Glauber Costa wrote:

> > I think it may be simplest to only account for the pages used by a slab in
> > a memcg. That code could be added to the functions in the slab allocators
> > that interface with the page allocators. Those are not that performance
> > critical and would do not much harm.
>

> No, I don't think so. Well, accounting the page is easy, but when we do a new
> allocation, we need to match a process to its correspondent page. This will
> likely lead to flushing the internal cpu caches of the slub, for instance,
> hurting performance. That is because once we allocate a page, all objects on
> that page need to belong to the same cgroup.

Matching a process to its page is a complex thing even for pages used by
userspace.

How can you make sure that all objects on a page belong to the same
cgroup? There are various kernel allocations that have uses far beyond a
single context. There is already a certain degree of fuzziness there and
we tolerate that in other contexts as well.

> Also, you talk about intrusiveness, accounting pages is a lot more intrusive,
> since then you need to know a lot about the internal structure of each cache.
> Having the cache replicated has exactly the effect of isolating it better.

Why would you need to know about the internal structure? Just get the
current process context and use the cgroup that is readily available there
to account for the pages.

> > If you need per object accounting then the cleanest solution would be to
> > duplicate the per node arrays per memcg (or only the statistics) and have
> > the kmem_cache structure only once in memory.
>
> No, it's all per-page. Nothing here is per-object, maybe you misunderstood
> something?

There are free/used object counters in each page. You could account for
objects in the l3 lists or kmem_cache_node strcut and thereby avoid
having to deal with the individual objects at the per cpu level.

Subject: Re: [PATCH v3 13/28] slub: create duplicate cache
Posted by Christoph Lameter on Tue, 29 May 2012 16:05:16 GMT

View Forum Message <> Reply to Message

On Tue, 29 May 2012, Glauber Costa wrote:

> Accounting pages seems just crazy to me. If new allocators come in the future,
> organizing the pages in a different way, instead of patching it here and
> there, we need to totally rewrite this.

Quite to the contrary. We could either pass a THIS_IS_A_SLAB page flag to
the page allocator call or have a special call that does the accounting
and then calls the page allocator. The code could be completely in

cgroups. There would be no changes to the allocators aside from setting
the flag or calling the alternate page allocator functions.

> > Why do you need to increase the refcount? You made a full copy right?
>
> Yes, but I don't want this copy to go away while we have other caches around.

You copied all metadata so what is there that you would still need should
the other copy go away?

> So, in the memcg internals, I used a different reference counter, to avoid
> messing with this one. I could use that, and leave the original refcnt alone.
> Would you prefer this?

The refcounter is really not the issue.

I am a bit worried about the various duplicate features here and there.
The approach is not tightened down yet.

Subject: Re: [PATCH v3 18/28] slub: charge allocation to a memcg
Posted by Glauber Costa on Tue, 29 May 2012 16:06:45 GMT

View Forum Message <> Reply to Message

On 05/29/2012 06:51 PM, Christoph Lameter wrote:
> On Fri, 25 May 2012, Glauber Costa wrote:
>
>> This patch charges allocation of a slab object to a particular
>> memcg.
>
> I am wondering why you need all the other patches. The simplest approach
> would just to hook into page allocation and freeing from the slab
> allocators as done here and charge to the currently active cgroup. This
> avoids all the duplication of slab caches and per node as well as per cpu
> structures. A certain degree of fuzziness cannot be avoided given that
> objects are cached and may be served to multiple cgroups. If that can be
> tolerated then the rest would be just like this patch which could be made
> more simple and non intrusive.
>
Just hooking into the page allocation only works for caches with very
big objects. For all the others, we need to relay the process to the
correct cache.

Some objects may be shared, yes, but in reality most won't.

Let me give you an example:

We track task_struct here. So as a nice side effect of this, a fork bomb will be killed because it will not be able to allocate any further.

But if we're accounting only at page allocation time, it is quite
possible to come up with a pattern while I always let other cgroups
pay the price for the page, but I will be the one filling it.

Having an eventual dentry, for instance, shared among caches, is okay.
But the main use case is for process in different cgroups dealing with
totally different parts of the filesystem.

So we can't really afford to charge to the process touching the nth
object where n is the number of objects per page. We need to relay it to
the right one.

Subject: Re: [PATCH v3 19/28] slab: per-memcg accounting of slab caches
Posted by Glauber Costa on Tue, 29 May 2012 16:07:09 GMT
View Forum Message <> Reply to Message

On 05/29/2012 06:52 PM, Christoph Lameter wrote:
> On Fri, 25 May 2012, Glauber Costa wrote:
>
>> > This patch charges allocation of a slab object to a particular
>> > memcg.
> Ok so a requirement is to support tracking of individual slab
> objects to cgroups? That is going to be quite expensive since it will
> touch the hotpaths.
>

No, we track pages. But all the objects in the page belong to the same
cgroup.

Subject: Re: [PATCH v3 19/28] slab: per-memcg accounting of slab caches
Posted by Glauber Costa on Tue, 29 May 2012 16:13:28 GMT
View Forum Message <> Reply to Message

On 05/29/2012 08:07 PM, Glauber Costa wrote:
> On 05/29/2012 06:52 PM, Christoph Lameter wrote:
>> On Fri, 25 May 2012, Glauber Costa wrote:
>>
>>> > This patch charges allocation of a slab object to a particular
>>> > memcg.
>> Ok so a requirement is to support tracking of individual slab
>> objects to cgroups? That is going to be quite expensive since it will
>> touch the hotpaths.
>>

>
> No, we track pages. But all the objects in the page belong to the same
> cgroup.
>

Also, please note the following:

The code that relays us to the right cache, is wrapped inside a static
branch. Whoever is not using more than the root cgroup, will not suffer
a single bit.

If you are, but your process is in the right cgroup, you will
unfortunately pay function call penalty(*), but the code will make and
effort to detect that as early as possible and resume.

(*) Not even then if you fall in the following categories, that are
resolved inline:

```
+       if (!current->mm)
+               return cachep;
+       if (in_interrupt())
+               return cachep;
+       if (gfp & __GFP_NOFAIL)
+               return cachep;
```

Subject: Re: [PATCH v3 12/28] slab: pass memcg parameter to kmem_cache_create
Posted by Christoph Lameter on Tue, 29 May 2012 16:33:17 GMT
View Forum Message <> Reply to Message

On Tue, 29 May 2012, Glauber Costa wrote:

> > Ok this only duplicates the kmalloc arrays. Why not the others?
>
> It does duplicate the others.
>
> First it does a while look on the kmalloc caches, then a list_for_each_entry
> in the rest. You probably missed it.

There is no need to separately duplicate the kmalloc_caches. Those are
included on the cache_chain.

```
> > > @@ -2543,7 +2564,12 @@ kmem_cache_create (const char *name, size_t size,
> > > size_t align,
> > >    cachep->ctor = ctor;
> > >    cachep->name = name;
```

> > >
> > > + if (g_cpucache_up>= FULL)
> > > + mem_cgroup_register_cache(memcg, cachep);
> >
> > What happens if a cgroup was active during creation of slab xxy but
> > then a process running in a different cgroup uses that slab to allocate
> > memory? Is it charged to the first cgroup?
>
> I don't see this situation ever happening. kmem_cache_create, when called
> directly, will always create a global cache. It doesn't matter which cgroups
> are or aren't active at this time or any other. We create copies per-cgroup,
> but we create it lazily, when someone will touch it.

How do you detect that someone is touching it?

Subject: Re: [PATCH v3 12/28] slab: pass memcg parameter to kmem_cache_create
Posted by Glauber Costa on Tue, 29 May 2012 16:36:06 GMT
View Forum Message <> Reply to Message

On 05/29/2012 08:33 PM, Christoph Lameter wrote:
> On Tue, 29 May 2012, Glauber Costa wrote:
>
>>> Ok this only duplicates the kmalloc arrays. Why not the others?
>>
>> It does duplicate the others.
>>
>> First it does a while look on the kmalloc caches, then a list_for_each_entry
>> in the rest. You probably missed it.
>
> There is no need to separately duplicate the kmalloc_caches. Those are
> included on the cache_chain.
>
>>>> @@ -2543,7 +2564,12 @@ kmem_cache_create (const char *name, size_t size,
>>>> size_t align,
>>>>   cachep->ctor = ctor;
>>>>   cachep->name = name;
>>>>
>>>> + if (g_cpucache_up>= FULL)
>>>> + mem_cgroup_register_cache(memcg, cachep);
>>>
>>> What happens if a cgroup was active during creation of slab xxy but
>>> then a process running in a different cgroup uses that slab to allocate
>>> memory? Is it charged to the first cgroup?
>>
>> I don't see this situation ever happening. kmem_cache_create, when called
>> directly, will always create a global cache. It doesn't matter which cgroups

>> are or aren't active at this time or any other. We create copies per-cgroup,
>> but we create it lazily, when someone will touch it.
>
> How do you detect that someone is touching it?

kmem_alloc_cache will create mem_cgroup_get_kmem_cache.
(protected by static_branches, so won't happen if you don't have at
least non-root memcg using it)

* Then it detects which memcg the calling process belongs to,
* if it is the root memcg, go back to the allocation as quickly as we
  can
* otherwise, in the creation process, you will notice that each cache
  has an index. memcg will store pointers to the copies and find them by
  the index.

 From this point on, all the code of the caches is reused (except for
accounting the page)

Subject: Re: [PATCH v3 12/28] slab: pass memcg parameter to kmem_cache_create
Posted by Christoph Lameter on Tue, 29 May 2012 16:52:55 GMT
View Forum Message <> Reply to Message

On Tue, 29 May 2012, Glauber Costa wrote:

> > How do you detect that someone is touching it?
>
> kmem_alloc_cache will create mem_cgroup_get_kmem_cache.
> (protected by static_branches, so won't happen if you don't have at least
> non-root memcg using it)
>
> * Then it detects which memcg the calling process belongs to,
> * if it is the root memcg, go back to the allocation as quickly as we
>   can
> * otherwise, in the creation process, you will notice that each cache
>   has an index. memcg will store pointers to the copies and find them by
>   the index.
>
> From this point on, all the code of the caches is reused (except for
> accounting the page)

Well kmem_cache_alloc cache is the performance critical hotpath.

If you are already there and doing all of that then would it not be better
to simply count the objects allocated and freed per cgroup? Directly
increment and decrement counters in a cgroup? You do not really need to

duplicate the kmem_cache structure and do not need to modify allocators if
you are willing to take that kind of a performance hit. Put a wrapper
around kmem_cache_alloc/free and count things.

Subject: Re: [PATCH v3 12/28] slab: pass memcg parameter to kmem_cache_create
Posted by Glauber Costa on Tue, 29 May 2012 16:59:54 GMT
View Forum Message <> Reply to Message

On 05/29/2012 08:52 PM, Christoph Lameter wrote:
> Well kmem_cache_alloc cache is the performance critical hotpath.
>
> If you are already there and doing all of that then would it not be better
> to simply count the objects allocated and freed per cgroup? Directly
> increment and decrement counters in a cgroup? You do not really need to
> duplicate the kmem_cache structure and do not need to modify allocators if
> you are willing to take that kind of a performance hit. Put a wrapper
> around kmem_cache_alloc/free and count things.

Well, I see it as the difference between being a big slower, and a lot
slower.

Accounting in memcg is hard, specially because it is potentially
hierarchical, (meaning you need to nest downwards until your parents).

I never discussed that this is, unfortunately, a hotpath. However, I did
try to minimize the impact as much as I could.

Not to mention that the current scheme is bound to improvement as
cgroups improve. One of the things being discussed is to having all
cgroups always in the same hierarchy. If that ever happens, we can have
the information about the current cgroup stored in a very accessible
way, so to make this even faster.

This felt like the best way I could do with the current infrastructure,
(and again, I did make it free for people not limiting kmem), and is
way, way cheaper than doing accounting here.

Subject: Re: [PATCH v3 13/28] slub: create duplicate cache
Posted by Glauber Costa on Tue, 29 May 2012 17:05:45 GMT
View Forum Message <> Reply to Message

On 05/29/2012 08:05 PM, Christoph Lameter wrote:
> On Tue, 29 May 2012, Glauber Costa wrote:
>

>> Accounting pages seems just crazy to me. If new allocators come in the future,
>> organizing the pages in a different way, instead of patching it here and
>> there, we need to totally rewrite this.
>
> Quite to the contrary. We could either pass a THIS_IS_A_SLAB page flag to
> the page allocator call or have a special call that does the accounting
> and then calls the page allocator. The code could be completely in
> cgroups. There would be no changes to the allocators aside from setting
> the flag or calling the alternate page allocator functions.

Again: for the page allocation itself, we could do it. (maybe we still
can, keeping the rest of the approach, so as to simplify that particular
piece of code, and reduce the churn inside the cache - I am all for it)

But that only solves the page allocation part. I would still need to
make sure the caller is directed to the right page.

>>> Why do you need to increase the refcount? You made a full copy right?
>>
>> Yes, but I don't want this copy to go away while we have other caches around.
>
> You copied all metadata so what is there that you would still need should
> the other copy go away?

Well, consistency being one, because it sounded weird to have the parent
cache being deleted while the kids are around. You wouldn't be able to
reach them, for once.

But one can argue that if you deleted the cache, why would you want to
ever reach it?

I can try to remove the refcount.

>
>> So, in the memcg internals, I used a different reference counter, to avoid
>> messing with this one. I could use that, and leave the original refcnt alone.
>> Would you prefer this?
>
> The refcounter is really not the issue.
>
> I am a bit worried about the various duplicate features here and there.
> The approach is not tightened down yet.

Well, I still think that duplication of the structure is better - a lot
less intrusive - than messing with the cache internals.

I will try to at least have the page accounting done in a consistent
way. How about that?

Subject: Re: [PATCH v3 13/28] slub: create duplicate cache
Posted by Christoph Lameter on Tue, 29 May 2012 17:25:12 GMT
View Forum Message <> Reply to Message

On Tue, 29 May 2012, Glauber Costa wrote:

> I will try to at least have the page accounting done in a consistent way. How
> about that?

Ok. What do you mean by "consistent"? Since objects and pages can be used
in a shared way and since accounting in many areas of the kernel is
intentional fuzzy to avoid performance issues there is not that much of a
requirement for accuracy. We have problems even just defining what the
exact set of pages belonging to a task/process is.

Subject: Re: [PATCH v3 13/28] slub: create duplicate cache
Posted by Glauber Costa on Tue, 29 May 2012 17:27:02 GMT
View Forum Message <> Reply to Message

On 05/29/2012 09:25 PM, Christoph Lameter wrote:
>> I will try to at least have the page accounting done in a consistent way. How
>> > about that?
> Ok. What do you mean by "consistent"? Since objects and pages can be used
> in a shared way and since accounting in many areas of the kernel is
> intentional fuzzy to avoid performance issues there is not that much of a
> requirement for accuracy. We have problems even just defining what the
> exact set of pages belonging to a task/process is.
>
I mean consistent between allocators, done in a shared place.
Note that what we do here *is* still fuzzy to some degree. We can have a
level of sharing, and always account to the first one to touch, we don't
go worrying about this because that would be hell. We also don't
carry a process' objects around when we send it to a different cgroup.
Those are all already simplications for performance and simplicity sake.

But we really need a page to be filled with objects from the same
cgroup, and the non-shared objects to be accounted to the right place.

Otherwise, I don't think we can meet even the lighter of isolation
guarantees.

Subject: Re: [PATCH v3 13/28] slub: create duplicate cache
Posted by Christoph Lameter on Tue, 29 May 2012 19:26:02 GMT
View Forum Message <> Reply to Message

On Tue, 29 May 2012, Glauber Costa wrote:

> But we really need a page to be filled with objects from the same cgroup, and
> the non-shared objects to be accounted to the right place.

No other subsystem has such a requirement. Even the NUMA nodes are mostly
suggestions and can be ignored by the allocators to use memory from other
pages.

> Otherwise, I don't think we can meet even the lighter of isolation guarantees.

The approach works just fine with NUMA and cpusets. Isolation is mostly
done on the per node boundaries and you already have per node statistics.

Subject: Re: [PATCH v3 13/28] slub: create duplicate cache
Posted by Glauber Costa on Tue, 29 May 2012 19:40:02 GMT

View Forum Message <> Reply to Message

On 05/29/2012 11:26 PM, Christoph Lameter wrote:
> On Tue, 29 May 2012, Glauber Costa wrote:
>
>> But we really need a page to be filled with objects from the same cgroup, and
>> the non-shared objects to be accounted to the right place.
>
> No other subsystem has such a requirement. Even the NUMA nodes are mostly
> suggestions and can be ignored by the allocators to use memory from other
> pages.

Of course it does. Memcg itself has such a requirement. The collective
set of processes needs to have the pages it uses accounted to it, and
never go over limit.

>> Otherwise, I don't think we can meet even the lighter of isolation guarantees.
>
> The approach works just fine with NUMA and cpusets. Isolation is mostly
> done on the per node boundaries and you already have per node statistics.

I don't know about cpusets in details, but at least with NUMA, this is
not an apple-to-apple comparison. a NUMA node is not meant to contain
you. A container is, and that is why it is called a container.

NUMA just means what is the *best* node to put my memory.
Now, if you actually say, through you syscalls "this is the node it
should live in", then you have a constraint, that to the best of my
knowledge is respected.

Now isolation here, is done in the container boundary. (cgroups, to be
generic).

Subject: Re: [PATCH v3 13/28] slub: create duplicate cache
Posted by Christoph Lameter on Tue, 29 May 2012 19:55:49 GMT
View Forum Message <> Reply to Message

On Tue, 29 May 2012, Glauber Costa wrote:

> I don't know about cpusets in details, but at least with NUMA, this is not an
> apple-to-apple comparison. a NUMA node is not meant to contain you. A
> container is, and that is why it is called a container.

Cpusets contains sets of nodes. A cpusets "contains" nodes. These sets are
associated with applications.

> NUMA just means what is the *best* node to put my memory.
> Now, if you actually say, through you syscalls "this is the node it should
> live in", then you have a constraint, that to the best of my knowledge is
> respected.

Eith cpusets it means that memory needs to come from an assigned set of
nodes.

> Now isolation here, is done in the container boundary. (cgroups, to be
> generic).

Yes and with cpusets it is done at the cpuset boundary. Very much the
same.

Subject: Re: [PATCH v3 13/28] slub: create duplicate cache
Posted by Glauber Costa on Tue, 29 May 2012 20:08:38 GMT
View Forum Message <> Reply to Message

On 05/29/2012 11:55 PM, Christoph Lameter wrote:
>> NUMA just means what is the*best* node to put my memory.
>> > Now, if you actually say, through you syscalls "this is the node it should
>> > live in", then you have a constraint, that to the best of my knowledge is
>> > respected.
> Eith cpusets it means that memory needs to come from an assigned set of
> nodes.
>
>> > Now isolation here, is done in the container boundary. (cgroups, to be
>> > generic).
> Yes and with cpusets it is done at the cpuset boundary. Very much the
> same.

Well, I'd have to dive in the code a bit more, but that the impression
that the documentation gives me, by saying:

"Cpusets constrain the CPU and Memory placement of tasks to only
the resources within a task's current cpuset."

is that you can't allocate from a node outside that set. Is this correct?

So extrapolating this to memcg, the situation is as follows:

* You can't use more memory than what you are assigned to.
* In order to do that, you need to account the memory you are using
* and to account the memory you are using, all objects in the page
  must belong to you.

Please note the following:

Having two cgroups touching the same object is something. It tells
something about the relationship between them. This is shared memory.

Now having two cgroups putting objects in the same page, *does not mean
_anything_*. It just mean that one had the luck to allocate just after
the other.

With a predictable enough workload, this is a recipe for working around
the very protection we need to establish: one can DoS a physical box
full of containers, by always allocating in someone else's pages, and
pinning kernel memory down. Never releasing it, so the shrinkers are
useless.

So I still believe that if a page is allocated to a cgroup, all the
objects in there belong to it - unless of course the sharing actually
means something - and identifying this is just too complicated.

Subject: Re: [PATCH v3 13/28] slub: create duplicate cache
Posted by Christoph Lameter on Tue, 29 May 2012 20:21:17 GMT

View Forum Message <> Reply to Message

On Wed, 30 May 2012, Glauber Costa wrote:

> Well, I'd have to dive in the code a bit more, but that the impression that
> the documentation gives me, by saying:
>
> "Cpusets constrain the CPU and Memory placement of tasks to only
> the resources within a task's current cpuset."
>
> is that you can't allocate from a node outside that set. Is this correct?

Basically yes but there are exceptions (like slab queues etc). Look at the
hardwall stuff too that allows more exceptions for kernel allocations to

use memory from other nodes.

> So extrapolating this to memcg, the situation is as follows:
>
> * You can't use more memory than what you are assigned to.
> * In order to do that, you need to account the memory you are using
> * and to account the memory you are using, all objects in the page
>   must belong to you.

Cpusets work at the page boundary and they do not have the requirement you
are mentioning of all objects in the page having to belong to a certain
cpusets. Let that go and things become much easier.

> With a predictable enough workload, this is a recipe for working around the
> very protection we need to establish: one can DoS a physical box full of
> containers, by always allocating in someone else's pages, and pinning kernel
> memory down. Never releasing it, so the shrinkers are useless.

Sure you can construct hyperthetical cases like that. But then that is
true already of other container like logic in the kernel already.

> So I still believe that if a page is allocated to a cgroup, all the objects in
> there belong to it - unless of course the sharing actually means something -
> and identifying this is just too complicated.

We have never worked container like logic like that in the kernel due to
the complicated logic you would have to put in. The requirement that all
objects in a page come from the same container is not necessary. If you
drop this notion then things become very easy and the patches will become
simple.

Subject: Re: [PATCH v3 13/28] slub: create duplicate cache
Posted by Glauber Costa on Tue, 29 May 2012 20:25:36 GMT
View Forum Message <> Reply to Message

On 05/30/2012 12:21 AM, Christoph Lameter wrote:
> On Wed, 30 May 2012, Glauber Costa wrote:
>
>> Well, I'd have to dive in the code a bit more, but that the impression that
>> the documentation gives me, by saying:
>>
>> "Cpusets constrain the CPU and Memory placement of tasks to only
>> the resources within a task's current cpuset."
>>
>> is that you can't allocate from a node outside that set. Is this correct?
>
> Basically yes but there are exceptions (like slab queues etc). Look at the

> hardwall stuff too that allows more exceptions for kernel allocations to
> use memory from other nodes.
>
>> So extrapolating this to memcg, the situation is as follows:
>>
>> * You can't use more memory than what you are assigned to.
>> * In order to do that, you need to account the memory you are using
>> * and to account the memory you are using, all objects in the page
>>   must belong to you.
>
> Cpusets work at the page boundary and they do not have the requirement you
> are mentioning of all objects in the page having to belong to a certain
> cpusets. Let that go and things become much easier.
>
>> With a predictable enough workload, this is a recipe for working around the
>> very protection we need to establish: one can DoS a physical box full of
>> containers, by always allocating in someone else's pages, and pinning kernel
>> memory down. Never releasing it, so the shrinkers are useless.
>
> Sure you can construct hyperthetical cases like that. But then that is
> true already of other container like logic in the kernel already.
>
>> So I still believe that if a page is allocated to a cgroup, all the objects in
>> there belong to it - unless of course the sharing actually means something -
>> and identifying this is just too complicated.
>
> We have never worked container like logic like that in the kernel due to
> the complicated logic you would have to put in. The requirement that all
> objects in a page come from the same container is not necessary. If you
> drop this notion then things become very easy and the patches will become
> simple.

I promise to look at that in more detail and get back to it. In the
meantime, I think it would be enlightening to hear from other parties as
well, specially the ones also directly interested in using the technology.

Subject: Re: [PATCH v3 13/28] slub: create duplicate cache
Posted by Suleiman Souhlal on Tue, 29 May 2012 20:57:41 GMT
View Forum Message <> Reply to Message

Hi Christoph,

On Tue, May 29, 2012 at 1:21 PM, Christoph Lameter <cl@linux.com> wrote:
> On Wed, 30 May 2012, Glauber Costa wrote:
>
>> Well, I'd have to dive in the code a bit more, but that the impression that
>> the documentation gives me, by saying:

>>
>> "Cpusets constrain the CPU and Memory placement of tasks to only
>> the resources within a task's current cpuset."
>>
>> is that you can't allocate from a node outside that set. Is this correct?
>
> Basically yes but there are exceptions (like slab queues etc). Look at the
> hardwall stuff too that allows more exceptions for kernel allocations to
> use memory from other nodes.
>
>> So extrapolating this to memcg, the situation is as follows:
>>
>> * You can't use more memory than what you are assigned to.
>> * In order to do that, you need to account the memory you are using
>> * and to account the memory you are using, all objects in the page
>>   must belong to you.
>
> Cpusets work at the page boundary and they do not have the requirement you
> are mentioning of all objects in the page having to belong to a certain
> cpusets. Let that go and things become much easier.
>
>> With a predictable enough workload, this is a recipe for working around the
>> very protection we need to establish: one can DoS a physical box full of
>> containers, by always allocating in someone else's pages, and pinning kernel
>> memory down. Never releasing it, so the shrinkers are useless.
>
> Sure you can construct hyperthetical cases like that. But then that is
> true already of other container like logic in the kernel already.
>
>> So I still believe that if a page is allocated to a cgroup, all the objects in
>> there belong to it - unless of course the sharing actually means something -
>> and identifying this is just too complicated.
>
> We have never worked container like logic like that in the kernel due to
> the complicated logic you would have to put in. The requirement that all
> objects in a page come from the same container is not necessary. If you
> drop this notion then things become very easy and the patches will become
> simple.

Back when we (Google) started using cpusets for memory isolation (fake
NUMA), we found that there was a significant isolation breakage coming
from slab pages belonging to one cpuset being used by other cpusets,
which caused us problems: It was very easy for one job to cause slab
growth in another container, which would cause it to OOM, despite
being well-behaved.

Because of this, we had to add logic to prevent that from happening
(by making sure we only allocate objects from pages coming from our

allowed nodes).

Now that we're switching to doing containers with memcg, I think this
is a hard requirement, for us. :-(

-- Suleiman

Subject: Re: [PATCH v3 13/28] slub: create duplicate cache
Posted by Tejun Heo on Wed, 30 May 2012 01:29:55 GMT
View Forum Message <> Reply to Message

Hello, Christoph, Glauber.

On Wed, May 30, 2012 at 12:25:36AM +0400, Glauber Costa wrote:
> >We have never worked container like logic like that in the kernel due to
> >the complicated logic you would have to put in. The requirement that all
> >objects in a page come from the same container is not necessary. If you
> >drop this notion then things become very easy and the patches will become
> >simple.
>
> I promise to look at that in more detail and get back to it. In the
> meantime, I think it would be enlightening to hear from other
> parties as well, specially the ones also directly interested in
> using the technology.

I don't think I'm too interested in using the technology ;) and
haven't read the code (just glanced through the descriptions and
discussions), but, in general, I think the approach of duplicating
memory allocator per-memcg is a sane approach. It isn't the most
efficient one with the possibility of wasting considerable amount of
caching area per memcg but it is something which can mostly stay out
of the way if done right and that's how I want cgroup implementations
to be.

The two goals for cgroup controllers that I think are important are
proper (no, not crazy perfect but good enough) isolation and an
implementation which doesn't impact !cg path in an intrusive manner -
if someone who doesn't care about cgroup but knows and wants to work
on the subsystem should be able to mostly ignore cgroup support. If
that means overhead for cgroup users, so be it.

Without looking at the actual code, my rainbow-farting unicorn here
would be having a common slXb interface layer which handles
interfacing with memory allocator users and cgroup and let slXb
implement the same backend interface which doesn't care / know about
cgroup at all (other than using the correct allocation context, that
is). Glauber, would something like that be possible?

Thanks.

--
tejun

Subject: Re: Re: [PATCH v3 13/28] slub: create duplicate cache
Posted by James Bottomley on Wed, 30 May 2012 07:28:26 GMT
View Forum Message <> Reply to Message

On Wed, 2012-05-30 at 10:29 +0900, Tejun Heo wrote:
> Hello, Christoph, Glauber.
>
> On Wed, May 30, 2012 at 12:25:36AM +0400, Glauber Costa wrote:
> > >We have never worked container like logic like that in the kernel due to
> > >the complicated logic you would have to put in. The requirement that all
> > >objects in a page come from the same container is not necessary. If you
> > >drop this notion then things become very easy and the patches will become
> > >simple.
> >
> > I promise to look at that in more detail and get back to it. In the
> > meantime, I think it would be enlightening to hear from other
> > parties as well, specially the ones also directly interested in
> > using the technology.
>
> I don't think I'm too interested in using the technology ;) and
> haven't read the code (just glanced through the descriptions and
> discussions), but, in general, I think the approach of duplicating
> memory allocator per-memcg is a sane approach. It isn't the most
> efficient one with the possibility of wasting considerable amount of
> caching area per memcg but it is something which can mostly stay out
> of the way if done right and that's how I want cgroup implementations
> to be.

Exactly: we at parallels initially disliked the cgroup multipled by slab
approach (Our beancounters do count objects) because we feared memory
wastage and density is very important to us (which tends to mean
efficient use of memory) however, when we ran through the calculations
in Prague, you can show that we have ~200 slabs and if each wastes half
a page, thats ~4MB memory lost per container. Since most virtual
environments are of the order nowadays of 0.5GB, we feel it's an
annoying but acceptable price to pay.

James

> The two goals for cgroup controllers that I think are important are
> proper (no, not crazy perfect but good enough) isolation and an

> implementation which doesn't impact !cg path in an intrusive manner -
> if someone who doesn't care about cgroup but knows and wants to work
> on the subsystem should be able to mostly ignore cgroup support. If
> that means overhead for cgroup users, so be it.
>
> Without looking at the actual code, my rainbow-farting unicorn here
> would be having a common slXb interface layer which handles
> interfacing with memory allocator users and cgroup and let slXb
> implement the same backend interface which doesn't care / know about
> cgroup at all (other than using the correct allocation context, that
> is). Glauber, would something like that be possible?
>
> Thanks.
>

Subject: Re: [PATCH v3 13/28] slub: create duplicate cache
Posted by Glauber Costa on Wed, 30 May 2012 07:54:16 GMT
View Forum Message <> Reply to Message

On 05/30/2012 05:29 AM, Tejun Heo wrote:
> The two goals for cgroup controllers that I think are important are
> proper (no, not crazy perfect but good enough) isolation and an
> implementation which doesn't impact !cg path in an intrusive manner -
> if someone who doesn't care about cgroup but knows and wants to work
> on the subsystem should be able to mostly ignore cgroup support. If
> that means overhead for cgroup users, so be it.

Well, my code in the slab is totally wrapped in static branches. They
only come active when the first group is *limited* (not even created:
you can have a thousand memcg, if none of them are kmem limited, nothing
will happen).

After that, the cost paid is to find out at which cgroup the process is
at. I believe that if we had a faster way for this (like for instance:
if we had a single hierarchy, the scheduler could put this in a percpu
variable after context switch - or any other method), then the cost of
it could be really low, even when this is enabled.

> Without looking at the actual code, my rainbow-farting unicorn here
> would be having a common slXb interface layer which handles
> interfacing with memory allocator users and cgroup and let slXb
> implement the same backend interface which doesn't care / know about
> cgroup at all (other than using the correct allocation context, that
> is). Glauber, would something like that be possible?

It is a matter of degree. There *is* a lot of stuff in common code, and
I tried to do it as much as I could. Christoph gave me a nice idea about

hinting to the page allocator that this page is a slab page before the
allocation, and then we could account from the page allocator directly -
without touching the cache code at all. This could be done, but some
stuff would still be there, basically because of differences in how the
allocator behaves. I think long term Christoph's effort to produce
common code among them will help a lot, if they stabilize their behavior
in certain areas.

I will rework this series to try work more towards this goal, but at
least for now I'll keep duplicating the caches. I still don't believe
that a loose accounting to the extent Christoph proposed will achieve
what we need this to achieve.

Subject: Re: [PATCH v3 13/28] slub: create duplicate cache
Posted by Tejun Heo on Wed, 30 May 2012 08:02:35 GMT
View Forum Message <> Reply to Message

Hello, Glauber.

On Wed, May 30, 2012 at 4:54 PM, Glauber Costa <glommer@parallels.com> wrote:
> On 05/30/2012 05:29 AM, Tejun Heo wrote:
>>
>> The two goals for cgroup controllers that I think are important are
>> proper (no, not crazy perfect but good enough) isolation and an
>> implementation which doesn't impact !cg path in an intrusive manner -
>> if someone who doesn't care about cgroup but knows and wants to work
>> on the subsystem should be able to mostly ignore cgroup support. If
>> that means overhead for cgroup users, so be it.
>
>
> Well, my code in the slab is totally wrapped in static branches. They only
> come active when the first group is *limited* (not even created: you can
> have a thousand memcg, if none of them are kmem limited, nothing will
> happen).

Great, but I'm not sure why you're trying to emphasize that while my
point was about memory overhead and that it's OK to have some
overheads for cg users. :)

> After that, the cost paid is to find out at which cgroup the process is at.
> I believe that if we had a faster way for this (like for instance: if we had
> a single hierarchy, the scheduler could put this in a percpu variable after
> context switch - or any other method), then the cost of it could be really
> low, even when this is enabled.

Someday, hopefully.

> I will rework this series to try work more towards this goal, but at least
> for now I'll keep duplicating the caches. I still don't believe that a loose
> accounting to the extent Christoph proposed will achieve what we need this
> to achieve.

Yeah, I prefer your per-cg cache approach but do hope that it stays as
far from actual allocator code as possible. Christoph, would it be
acceptable if the cg logic is better separated?

Thanks.

--
tejun

Subject: Re: [PATCH v3 12/28] slab: pass memcg parameter to kmem_cache_create
Posted by Frederic Weisbecker on Wed, 30 May 2012 11:01:37 GMT

View Forum Message <> Reply to Message

On Tue, May 29, 2012 at 11:52:55AM -0500, Christoph Lameter wrote:
> On Tue, 29 May 2012, Glauber Costa wrote:
>
> > > How do you detect that someone is touching it?
> >
> > kmem_alloc_cache will create mem_cgroup_get_kmem_cache.
> > (protected by static_branches, so won't happen if you don't have at least
> > non-root memcg using it)
> >
> > * Then it detects which memcg the calling process belongs to,
> > * if it is the root memcg, go back to the allocation as quickly as we
> >   can
> > * otherwise, in the creation process, you will notice that each cache
> >   has an index. memcg will store pointers to the copies and find them by
> >   the index.
> >
> > From this point on, all the code of the caches is reused (except for
> > accounting the page)
>
> Well kmem_cache_alloc cache is the performance critical hotpath.
>
> If you are already there and doing all of that then would it not be better
> to simply count the objects allocated and freed per cgroup? Directly
> increment and decrement counters in a cgroup? You do not really need to
> duplicate the kmem_cache structure and do not need to modify allocators if
> you are willing to take that kind of a performance hit. Put a wrapper
> around kmem_cache_alloc/free and count things.

I believe one of the issues is also that a task can migrate to another cgroup
anytime. But an object that has been charged to a cgroup must be later uncharged
to the same, unless you move the charge as you move the task. But then it means
you need to keep track of the allocations per task, and you also need to be able
to do that reverse mapping (object -> allocating task) because your object can
be allocated by task A but later freed by task B. Then when you do the uncharge
it must happen to the cgroup of A, not the one of B.

That all would be much more complicated and performance sensitive than what this
patchset does. Dealing with duplicate caches for accounting seem to me a good tradeoff
between allocation performance hot path and maintaining cgroups semantics.

Subject: Re: [PATCH v3 16/28] memcg: kmem controller charge/uncharge infrastructure
Posted by Frederic Weisbecker on Wed, 30 May 2012 12:17:09 GMT
View Forum Message <> Reply to Message

On Fri, May 25, 2012 at 05:03:36PM +0400, Glauber Costa wrote:
```
>  #endif /* CONFIG_CGROUP_MEM_RES_CTLR_KMEM */
> +static __always_inline struct kmem_cache *
> +mem_cgroup_get_kmem_cache(struct kmem_cache *cachep, gfp_t gfp)
> +{
> + if (!mem_cgroup_kmem_on)
> +  return cachep;
> + if (!current->mm)
> +  return cachep;
> + if (in_interrupt())
> +  return cachep;
```

Does that mean interrupts are kept out of accounting?

Subject: Re: [PATCH v3 16/28] memcg: kmem controller charge/uncharge infrastructure
Posted by Glauber Costa on Wed, 30 May 2012 12:26:28 GMT
View Forum Message <> Reply to Message

On 05/30/2012 04:17 PM, Frederic Weisbecker wrote:
```
> On Fri, May 25, 2012 at 05:03:36PM +0400, Glauber Costa wrote:
>>   #endif /* CONFIG_CGROUP_MEM_RES_CTLR_KMEM */
>> +static __always_inline struct kmem_cache *
>> +mem_cgroup_get_kmem_cache(struct kmem_cache *cachep, gfp_t gfp)
>> +{
>> + if (!mem_cgroup_kmem_on)
>> +  return cachep;
>> + if (!current->mm)
```

>> + return cachep;
>> + if (in_interrupt())
>> + return cachep;
>
> Does that mean interrupts are kept out of accounting?

Well, since interrupts have no process context, if you are in an
interrupt I can't think of any sane thing to do than relay it to the
root memcg. That's what happen when I return cachep: I return the
original parent cache, and we allocate from that.

Subject: Re: [PATCH v3 16/28] memcg: kmem controller charge/uncharge infrastructure
Posted by Frederic Weisbecker on Wed, 30 May 2012 12:34:08 GMT
View Forum Message <> Reply to Message

On Fri, May 25, 2012 at 05:03:36PM +0400, Glauber Costa wrote:
> +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);

So this takes the memcg of the group owner rather than the
task? I understand why we want this for user memory, but for
kernel?

Subject: Re: [PATCH v3 16/28] memcg: kmem controller charge/uncharge infrastructure
Posted by Glauber Costa on Wed, 30 May 2012 12:38:39 GMT
View Forum Message <> Reply to Message

On 05/30/2012 04:34 PM, Frederic Weisbecker wrote:
> On Fri, May 25, 2012 at 05:03:36PM +0400, Glauber Costa wrote:
>> +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);
>
```

> So this takes the memcg of the group owner rather than the
> task? I understand why we want this for user memory, but for
> kernel?

That was already discussed when this first came up in my last submission
If I recall correctly, Kame pointed out that this would be needed for
proper OOM-scoring and killing.

Now of course we won't oom kernel threads or anything like that. But
since this is also accounted towards memcg, it should at least be
consistent with each memcg it accounts to.

We can't account kmem for the thread's memcg, and mem to the process'.

Subject: Re: [PATCH v3 16/28] memcg: kmem controller charge/uncharge infrastructure
Posted by Frederic Weisbecker on Wed, 30 May 2012 13:04:18 GMT

View Forum Message <> Reply to Message

On Fri, May 25, 2012 at 05:03:36PM +0400, Glauber Costa wrote:

```
> +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;
```

Do you think it's possible that this memcg can be destroyed (like ss->destroy())
concurrently?

Probably not because there is a synchronize_rcu() in cgroup_diput() so as long
as we are in rcu_read_lock() we are fine.

OTOH current->mm->owner can exit() right after we fetched its memcg and thus the css_set
can be freed concurrently? And then the cgroup itself after we call rcu_read_unlock()
due to cgroup_diput().
And yet we are doing the mem_cgroup_get() below unconditionally assuming it's
always fine to get a reference to it.

May be I'm missing something?

```
> + 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);
```

Subject: Re: [PATCH v3 16/28] memcg: kmem controller charge/uncharge infrastructure

Posted by Glauber Costa on Wed, 30 May 2012 13:06:22 GMT

View Forum Message <> Reply to Message

On 05/30/2012 05:04 PM, Frederic Weisbecker wrote:
> Do you think it's possible that this memcg can be destroyed (like ss->destroy())
> concurrently?
>
> Probably not because there is a synchronize_rcu() in cgroup_diput() so as long
> as we are in rcu_read_lock() we are fine.
>
> OTOH current->mm->owner can exit() right after we fetched its memcg and thus the css_set
> can be freed concurrently? And then the cgroup itself after we call rcu_read_unlock()
> due to cgroup_diput().
> And yet we are doing the mem_cgroup_get() below unconditionally assuming it's
> always fine to get a reference to it.
>
> May be I'm missing something?
When a cache is created, we grab a reference to the memcg. So after the
cache is created, no.

When destroy is called, we flush the create queue, so if the cache is
not created yet, it will just disappear.

I think the only problem that might happen is in the following scenario:

* cache gets created, but ref count is not yet taken
* memcg disappears
* we try to inc refcount for a non-existent memcg, and crash.

This would be trivially solvable by grabing the reference earlier.
But even then, I need to audit this further to make sure it is really an
issue.

Subject: Re: [PATCH v3 16/28] memcg: kmem controller charge/uncharge infrastructure
Posted by Glauber Costa on Wed, 30 May 2012 13:09:52 GMT
View Forum Message <> Reply to Message

On 05/30/2012 05:11 PM, Frederic Weisbecker wrote:
> Can we have at least a comment in the code that explain the reasons of taking the
> owner rather than the task? It's not going to be very obvious to future reviewers.

Yes.

Subject: Re: [PATCH v3 16/28] memcg: kmem controller charge/uncharge infrastructure
Posted by Frederic Weisbecker on Wed, 30 May 2012 13:11:01 GMT
View Forum Message <> Reply to Message

```
On Wed, May 30, 2012 at 04:38:39PM +0400, Glauber Costa wrote:
> On 05/30/2012 04:34 PM, Frederic Weisbecker wrote:
> >On Fri, May 25, 2012 at 05:03:36PM +0400, Glauber Costa wrote:
> >>+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);
> >
```

> >So this takes the memcg of the group owner rather than the
> >task? I understand why we want this for user memory, but for
> >kernel?
>
> That was already discussed when this first came up in my last submission
> If I recall correctly, Kame pointed out that this would be needed
> for proper OOM-scoring and killing.

Can we have at least a comment in the code that explain the reasons of taking the
owner rather than the task? It's not going to be very obvious to future reviewers.

> Now of course we won't oom kernel threads or anything like that.

Seems we are not even accounting them anyway.

> But since this is also accounted towards memcg, it should at least be
> consistent with each memcg it accounts to.
>
> We can't account kmem for the thread's memcg, and mem to the process'.

Don't know. This goes a bit against cgroups semantics which group at the task
level and not process. But I personally don't mind much, as long as it's
documented.

Subject: Re: [PATCH v3 16/28] memcg: kmem controller charge/uncharge infrastructure
Posted by Frederic Weisbecker on Wed, 30 May 2012 13:37:39 GMT
View Forum Message <> Reply to Message

On Wed, May 30, 2012 at 05:06:22PM +0400, Glauber Costa wrote:
> On 05/30/2012 05:04 PM, Frederic Weisbecker wrote:
> >Do you think it's possible that this memcg can be destroyed (like ss->destroy())
> >concurrently?
> >
> >Probably not because there is a synchronize_rcu() in cgroup_diput() so as long
> >as we are in rcu_read_lock() we are fine.
> >
> >OTOH current->mm->owner can exit() right after we fetched its memcg and thus the css_set
> >can be freed concurrently? And then the cgroup itself after we call rcu_read_unlock()
> >due to cgroup_diput().
> >And yet we are doing the mem_cgroup_get() below unconditionally assuming it's
> >always fine to get a reference to it.
> >
> >May be I'm missing something?
> When a cache is created, we grab a reference to the memcg. So after
> the cache is created, no.
>

> When destroy is called, we flush the create queue, so if the cache
> is not created yet, it will just disappear.
>
> I think the only problem that might happen is in the following scenario:
>
> * cache gets created, but ref count is not yet taken
> * memcg disappears
> * we try to inc refcount for a non-existent memcg, and crash.
>
> This would be trivially solvable by grabing the reference earlier.
> But even then, I need to audit this further to make sure it is
> really an issue.

Right. __mem_cgroup_get_kmem_cache() fetches the memcg of the owner
and calls memcg_create_cache_enqueue() which does css_tryget(&memcg->css).
After this tryget I think you're fine. And in-between you're safe against
css_set removal due to rcu_read_lock().

I'm less clear with __mem_cgroup_new_kmem_page() though...

Subject: Re: [PATCH v3 16/28] memcg: kmem controller charge/uncharge infrastructure
Posted by Glauber Costa on Wed, 30 May 2012 13:37:57 GMT
View Forum Message <> Reply to Message

On 05/30/2012 05:37 PM, Frederic Weisbecker wrote:
> Right. __mem_cgroup_get_kmem_cache() fetches the memcg of the owner
> and calls memcg_create_cache_enqueue() which does css_tryget(&memcg->css).
> After this tryget I think you're fine. And in-between you're safe against
> css_set removal due to rcu_read_lock().
>
> I'm less clear with __mem_cgroup_new_kmem_page() though...

That one does not get memcg->css but it does call mem_cgroup_get(), that
does prevent against the memcg structure being freed, which I believe to
be good enough.

Subject: Re: [PATCH v3 16/28] memcg: kmem controller charge/uncharge infrastructure
Posted by Frederic Weisbecker on Wed, 30 May 2012 13:53:21 GMT
View Forum Message <> Reply to Message

On Wed, May 30, 2012 at 05:37:57PM +0400, Glauber Costa wrote:
> On 05/30/2012 05:37 PM, Frederic Weisbecker wrote:
> >Right. __mem_cgroup_get_kmem_cache() fetches the memcg of the owner

> >and calls memcg_create_cache_enqueue() which does css_tryget(&memcg->css).
> >After this tryget I think you're fine. And in-between you're safe against
> >css_set removal due to rcu_read_lock().
> >
> >I'm less clear with __mem_cgroup_new_kmem_page() though...
>
> That one does not get memcg->css but it does call mem_cgroup_get(),
> that does prevent against the memcg structure being freed, which I
> believe to be good enough.

What if the owner calls cgroup_exit() between mem_cgroup_from_task()
and mem_cgroup_get()? The css_set which contains the memcg gets freed.
Also the reference on the memcg doesn't even prevent the css_set to
be removed, does it?

Subject: Re: [PATCH v3 16/28] memcg: kmem controller charge/uncharge infrastructure
Posted by Glauber Costa on Wed, 30 May 2012 13:55:38 GMT
View Forum Message <> Reply to Message

On 05/30/2012 05:53 PM, Frederic Weisbecker wrote:
> On Wed, May 30, 2012 at 05:37:57PM +0400, Glauber Costa wrote:
>> On 05/30/2012 05:37 PM, Frederic Weisbecker wrote:
>>> Right. __mem_cgroup_get_kmem_cache() fetches the memcg of the owner
>>> and calls memcg_create_cache_enqueue() which does css_tryget(&memcg->css).
>>> After this tryget I think you're fine. And in-between you're safe against
>>> css_set removal due to rcu_read_lock().
>>>
>>> I'm less clear with __mem_cgroup_new_kmem_page() though...
>>
>> That one does not get memcg->css but it does call mem_cgroup_get(),
>> that does prevent against the memcg structure being freed, which I
>> believe to be good enough.
>
> What if the owner calls cgroup_exit() between mem_cgroup_from_task()
> and mem_cgroup_get()? The css_set which contains the memcg gets freed.
> Also the reference on the memcg doesn't even prevent the css_set to
> be removed, does it?
It doesn't, but we don't really care. The css can go away, if the memcg
structure stays. The caches will outlive the memcg anyway, since it is
possible that you delete it, with some caches still holding objects that
are not freed (they will be marked as dead).

Subject: Re: [PATCH v3 16/28] memcg: kmem controller charge/uncharge infrastructure

Posted by Frederic Weisbecker on Wed, 30 May 2012 15:33:00 GMT
View Forum Message <> Reply to Message

On Wed, May 30, 2012 at 05:55:38PM +0400, Glauber Costa wrote:
> On 05/30/2012 05:53 PM, Frederic Weisbecker wrote:
> >On Wed, May 30, 2012 at 05:37:57PM +0400, Glauber Costa wrote:
> >>On 05/30/2012 05:37 PM, Frederic Weisbecker wrote:
> >>>Right. __mem_cgroup_get_kmem_cache() fetches the memcg of the owner
> >>>and calls memcg_create_cache_enqueue() which does css_tryget(&memcg->css).
> >>>After this tryget I think you're fine. And in-between you're safe against
> >>>css_set removal due to rcu_read_lock().
> >>>
> >>>I'm less clear with __mem_cgroup_new_kmem_page() though...
> >>
> >>That one does not get memcg->css but it does call mem_cgroup_get(),
> >>that does prevent against the memcg structure being freed, which I
> >>believe to be good enough.
> >
> >What if the owner calls cgroup_exit() between mem_cgroup_from_task()
> >and mem_cgroup_get()? The css_set which contains the memcg gets freed.
> >Also the reference on the memcg doesn't even prevent the css_set to
> >be removed, does it?
> It doesn't, but we don't really care. The css can go away, if the
> memcg structure stays.

Ah right, the memcg itself is only freed at destroy time.

> The caches will outlive the memcg anyway,
> since it is possible that you delete it, with some caches still
> holding objects that
> are not freed (they will be marked as dead).

I guess I need to look at how the destroy path is handled in your patchset
then. Or how you ensure that __mem_cgroup_new_kmem_page() can't race against
destroy.

Subject: Re: [PATCH v3 13/28] slub: create duplicate cache
Posted by Christoph Lameter on Wed, 30 May 2012 15:37:49 GMT
View Forum Message <> Reply to Message

On Wed, 30 May 2012, Tejun Heo wrote:

> Yeah, I prefer your per-cg cache approach but do hope that it stays as
> far from actual allocator code as possible. Christoph, would it be
> acceptable if the cg logic is better separated?

Certainly anything that would allow this to be separated out would be

appreciated. I do not anticipate to ever run cgroup in my environment and
that is due to the additional latency created in key OS paths. Memory we
have enough. The increased cache footprint is killing performance.

Subject: Re: [PATCH v3 16/28] memcg: kmem controller charge/uncharge infrastructure
Posted by Glauber Costa on Wed, 30 May 2012 16:16:59 GMT
View Forum Message <> Reply to Message

On 05/30/2012 07:33 PM, Frederic Weisbecker wrote:
> On Wed, May 30, 2012 at 05:55:38PM +0400, Glauber Costa wrote:
>> On 05/30/2012 05:53 PM, Frederic Weisbecker wrote:
>>> On Wed, May 30, 2012 at 05:37:57PM +0400, Glauber Costa wrote:
>>>> On 05/30/2012 05:37 PM, Frederic Weisbecker wrote:
>>>>> Right. __mem_cgroup_get_kmem_cache() fetches the memcg of the owner
>>>>> and calls memcg_create_cache_enqueue() which does css_tryget(&memcg->css).
>>>>> After this tryget I think you're fine. And in-between you're safe against
>>>>> css_set removal due to rcu_read_lock().
>>>>>
>>>>> I'm less clear with __mem_cgroup_new_kmem_page() though...
>>>>
>>>> That one does not get memcg->css but it does call mem_cgroup_get(),
>>>> that does prevent against the memcg structure being freed, which I
>>>> believe to be good enough.
>>>
>>> What if the owner calls cgroup_exit() between mem_cgroup_from_task()
>>> and mem_cgroup_get()? The css_set which contains the memcg gets freed.
>>> Also the reference on the memcg doesn't even prevent the css_set to
>>> be removed, does it?
>> It doesn't, but we don't really care. The css can go away, if the
>> memcg structure stays.
>
> Ah right, the memcg itself is only freed at destroy time.
>
>> The caches will outlive the memcg anyway,
>> since it is possible that you delete it, with some caches still
>> holding objects that
>> are not freed (they will be marked as dead).
>
> I guess I need to look at how the destroy path is handled in your patchset
> then. Or how you ensure that __mem_cgroup_new_kmem_page() can't race against
> destroy.
Appreciate that, thanks.

Subject: Re: [PATCH v3 00/28] kmem limitation for memcg

Posted by Frederic Weisbecker on Thu, 07 Jun 2012 10:26:10 GMT
View Forum Message <> Reply to Message

On Fri, May 25, 2012 at 05:03:20PM +0400, Glauber Costa wrote:
> Hello All,
>
> This is my new take for the memcg kmem accounting. This should merge
> all of the previous comments from you, plus fix a bunch of bugs.
>
> At this point, I consider the series pretty mature. Since last submission
> 2 weeks ago, I focused on broadening the testing coverage. Some bugs were
> fixed, but that of course doesn't mean no bugs exist.
>
> I believe some of the early patches here are already in some trees around.
> I don't know who should pick this, so if everyone agrees with what's in here,
> please just ack them and tell me which tree I should aim for (-mm? Hocko's?)
> and I'll rebase it.
>
> 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. Enabling and disabling of static_keys incorporate
> the last fixes for sock memcg, and should be pretty robust.
>
> 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.

So I believe that if I want to implement a per kernel stack accounting/limitation,
I need to work on top of your patchset.

What do you think about having some sub kmem accounting based on the caches?
For example there could be a specific accounting per kmem cache.

Like if we use a specific kmem cache to allocate the kernel stack
(as is done by some archs but I can generalize that for those who want
kernel stack accounting), allocations are accounted globally in the memcg as
done in your patchset but also on a seperate counter only for this kmem cache
on the memcg, resulting in a kmem.stack.usage somewhere.

The concept of per kmem cache accounting can be expanded more for any
kind of finegrained kmem accounting.

Thoughts?

Subject: Re: [PATCH v3 00/28] kmem limitation for memcg
Posted by Glauber Costa on Thu, 07 Jun 2012 10:53:07 GMT
View Forum Message <> Reply to Message

On 06/07/2012 02:26 PM, Frederic Weisbecker wrote:
> On Fri, May 25, 2012 at 05:03:20PM +0400, Glauber Costa wrote:
>> Hello All,
>>
>> This is my new take for the memcg kmem accounting. This should merge
>> all of the previous comments from you, plus fix a bunch of bugs.
>>
>> At this point, I consider the series pretty mature. Since last submission
>> 2 weeks ago, I focused on broadening the testing coverage. Some bugs were
>> fixed, but that of course doesn't mean no bugs exist.
>>
>> I believe some of the early patches here are already in some trees around.
>> I don't know who should pick this, so if everyone agrees with what's in here,
>> please just ack them and tell me which tree I should aim for (-mm? Hocko's?)
>> and I'll rebase it.
>>
>> 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. Enabling and disabling of static_keys incorporate
>> the last fixes for sock memcg, and should be pretty robust.
>>
>> 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.
>
> So I believe that if I want to implement a per kernel stack accounting/limitation,
> I need to work on top of your patchset.
>
> What do you think about having some sub kmem accounting based on the caches?
> For example there could be a specific accounting per kmem cache.
>
> Like if we use a specific kmem cache to allocate the kernel stack
> (as is done by some archs but I can generalize that for those who want
> kernel stack accounting), allocations are accounted globally in the memcg as
> done in your patchset but also on a seperate counter only for this kmem cache
> on the memcg, resulting in a kmem.stack.usage somewhere.
>
> The concept of per kmem cache accounting can be expanded more for any
> kind of finegrained kmem accounting.
>
> Thoughts?

I believe a general separation is too much, and will lead to knob
explosion. So I don't think it is a good idea.

Now, for the stack itself, it can be justified. The question that
remains to be answered is:

Why do you need to set the stack value separately? Isn't accounting the
stack value, and limiting against the global kmem limit enough?

Subject: Re: [PATCH v3 00/28] kmem limitation for memcg
Posted by Frederic Weisbecker on Thu, 07 Jun 2012 14:00:45 GMT
View Forum Message <> Reply to Message

On Thu, Jun 07, 2012 at 02:53:07PM +0400, Glauber Costa wrote:
> On 06/07/2012 02:26 PM, Frederic Weisbecker wrote:
> >On Fri, May 25, 2012 at 05:03:20PM +0400, Glauber Costa wrote:
> >>Hello All,
> >>
> >>This is my new take for the memcg kmem accounting. This should merge
> >>all of the previous comments from you, plus fix a bunch of bugs.
> >>
> >>At this point, I consider the series pretty mature. Since last submission
> >>2 weeks ago, I focused on broadening the testing coverage. Some bugs were
> >>fixed, but that of course doesn't mean no bugs exist.
> >>
> >>I believe some of the early patches here are already in some trees around.
> >>I don't know who should pick this, so if everyone agrees with what's in here,
> >>please just ack them and tell me which tree I should aim for (-mm? Hocko's?)
> >>and I'll rebase it.
> >>
> >>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. Enabling and disabling of static_keys incorporate
> >>the last fixes for sock memcg, and should be pretty robust.
> >>
> >>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.
> >
> >So I believe that if I want to implement a per kernel stack accounting/limitation,
> >I need to work on top of your patchset.
> >
> >What do you think about having some sub kmem accounting based on the caches?
> >For example there could be a specific accounting per kmem cache.
> >
> >Like if we use a specific kmem cache to allocate the kernel stack
> >(as is done by some archs but I can generalize that for those who want
> >kernel stack accounting), allocations are accounted globally in the memcg as
> >done in your patchset but also on a seperate counter only for this kmem cache
> >on the memcg, resulting in a kmem.stack.usage somewhere.
> >
> >The concept of per kmem cache accounting can be expanded more for any
> >kind of finegrained kmem accounting.

> >
> >Thoughts?
>
> I believe a general separation is too much, and will lead to knob
> explosion. So I don't think it is a good idea.

Right. This could be an option in kmem_cache_create() or something.

>
> Now, for the stack itself, it can be justified. The question that
> remains to be answered is:
>
> Why do you need to set the stack value separately? Isn't accounting
> the stack value, and limiting against the global kmem limit enough?

Well, I may want to let my container have a full access to some kmem
resources (net, file, etc...) but defend against fork bombs or NR_PROC
rlimit exhaustion of other containers.

So I need to be able to set my limit precisely on kstack.

Subject: Re: [PATCH v3 00/28] kmem limitation for memcg
Posted by KAMEZAWA Hiroyuki on Thu, 14 Jun 2012 02:24:53 GMT

View Forum Message <> Reply to Message

(2012/06/07 23:00), Frederic Weisbecker wrote:
> On Thu, Jun 07, 2012 at 02:53:07PM +0400, Glauber Costa wrote:
>> On 06/07/2012 02:26 PM, Frederic Weisbecker wrote:
>>> On Fri, May 25, 2012 at 05:03:20PM +0400, Glauber Costa wrote:
>>>> Hello All,
>>>>
>>>> This is my new take for the memcg kmem accounting. This should merge
>>>> all of the previous comments from you, plus fix a bunch of bugs.
>>>>
>>>> At this point, I consider the series pretty mature. Since last submission
>>>> 2 weeks ago, I focused on broadening the testing coverage. Some bugs were
>>>> fixed, but that of course doesn't mean no bugs exist.
>>>>
>>>> I believe some of the early patches here are already in some trees around.
>>>> I don't know who should pick this, so if everyone agrees with what's in here,
>>>> please just ack them and tell me which tree I should aim for (-mm? Hocko's?)
>>>> and I'll rebase it.
>>>>
>>>> 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. Enabling and disabling of static_keys incorporate
>>>> the last fixes for sock memcg, and should be pretty robust.

>>>>
>>>> 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.
>>>
>>> So I believe that if I want to implement a per kernel stack accounting/limitation,
>>> I need to work on top of your patchset.
>>>
>>> What do you think about having some sub kmem accounting based on the caches?
>>> For example there could be a specific accounting per kmem cache.
>>>
>>> Like if we use a specific kmem cache to allocate the kernel stack
>>> (as is done by some archs but I can generalize that for those who want
>>> kernel stack accounting), allocations are accounted globally in the memcg as
>>> done in your patchset but also on a seperate counter only for this kmem cache
>>> on the memcg, resulting in a kmem.stack.usage somewhere.
>>>
>>> The concept of per kmem cache accounting can be expanded more for any
>>> kind of finegrained kmem accounting.
>>>
>>> Thoughts?
>>
>> I believe a general separation is too much, and will lead to knob
>> explosion. So I don't think it is a good idea.
>
> Right. This could be an option in kmem_cache_create() or something.
>
>>
>> Now, for the stack itself, it can be justified. The question that
>> remains to be answered is:
>>
>> Why do you need to set the stack value separately? Isn't accounting
>> the stack value, and limiting against the global kmem limit enough?
>
> Well, I may want to let my container have a full access to some kmem
> resources (net, file, etc...) but defend against fork bombs or NR_PROC
> rlimit exhaustion of other containers.
>
> So I need to be able to set my limit precisely on kstack.

You explained that the limitation is necessary for fork-bomb, and the bad
point of fork-bomb is that it can cause OOM. So, the problem is OOM not fork-bomb.

If the problem is OOM, IIUC, generic kernel memory limiting will work better than
kernel stack limiting.

Thanks,
-Kame