Subject: [PATCH 00/10] memcg: Kernel Memory Accounting.
Posted by Suleiman Souhlal on Mon, 27 Feb 2012 22:58:43 GMT
View Forum Message <> Reply to Message

This patch series introduces kernel memory accounting to memcg.
It currently only accounts for slab.

It's very similar to the patchset I sent back in October, but
with a number of fixes and improvements.
There is also overlap with Glauber's patchset, but I decided to
send this because there are some pretty significant differences.

With this patchset, kernel memory gets counted in a memcg's
memory.kmem.usage_in_bytes.
It's possible to have a limit to the kernel memory by setting
memory.kmem.independent_kmem_limit to 1 and setting the limit in
memory.kmem.limit_in_bytes.
If memory.kmem.independent_kmem_limit is unset, kernel memory also
gets counted in the cgroup's memory.usage_in_bytes, and kernel
memory allocations may trigger memcg reclaim or OOM.

Slab gets accounted per-page, by using per-cgroup kmem_caches that
get created the first time an allocation of that type is done by
that cgroup.

The main difference with Glauber's patches is here: We try to
track all the slab allocations, while Glauber only tracks ones
that are explicitly marked.
We feel that it's important to track everything, because there
are a lot of different slab allocations that may use significant
amounts of memory, that we may not know of ahead of time.
This is also the main source of complexity in the patchset.

The per-cgroup kmem_cache approach makes it so that we only have
to do charges/uncharges in the slow path of the slab allocator,
which should have low performance impacts.

A per-cgroup kmem_cache will appear in slabinfo named like its
original cache, with the cgroup's name in parenthesis.
On cgroup deletion, the accounting gets moved to the root cgroup
and any existing cgroup kmem_cache gets "dead" appended to its
name, to indicate that its accounting was migrated.
The dead caches get removed once they no longer have any active
objects in them.

This patchset does not include any shrinker changes. We already have
patches for that, but I felt like it's more important to get the
accounting right first, before concentrating on the slab shrinking.

Some caveats:
- Only supports slab.c.
- There is a difference with non-memcg slab allocation in
  that with this, some slab allocations might fail when
  they never failed before. For example, a GFP_NOIO slab
  allocation wouldn't fail, but now it might.
  We have a change that makes slab accounting behave
  the same as non-accounted allocations, but I wasn't sure
  how important it was to include.
- We currently do two atomic operations on every accounted
  slab free, when we increment and decrement the kmem_cache's
  refcount. It's most likely possible to fix this.
- When CONFIG_CGROUP_MEM_RES_CTLR_KMEM is enabled, some
  conditional branches get added to the slab fast paths.
  That said, when the config option is disabled, this
  patchset is essentially a no-op.

I hope either this or Glauber's patchset will evolve into something
that is satisfactory to all the parties.

Patch series, based on Linus HEAD from today:

1/10 memcg: Kernel memory accounting infrastructure.
2/10 memcg: Uncharge all kmem when deleting a cgroup.
3/10 memcg: Reclaim when more than one page needed.
4/10 memcg: Introduce __GFP_NOACCOUNT.
5/10 memcg: Slab accounting.
6/10 memcg: Track all the memcg children of a kmem_cache.
7/10 memcg: Stop res_counter underflows.
8/10 memcg: Add CONFIG_CGROUP_MEM_RES_CTLR_KMEM_ACCT_ROOT.
9/10 memcg: Per-memcg memory.kmem.slabinfo file.
10/10 memcg: Document kernel memory accounting.

```
Documentation/cgroups/memory.txt |   44 +++-
include/linux/gfp.h              |    2 +
include/linux/memcontrol.h       |   30 ++-
include/linux/slab.h             |    1 +
include/linux/slab_def.h         |  102 +++++++-
init/Kconfig                     |    8 +
mm/memcontrol.c                  |  607 ++++++++++++++++++++++++++++++++++++--
mm/slab.c                        |  395 ++++++++++++++++++++++---
8 files changed, 1115 insertions(+), 74 deletions(-)
```

-- Suleiman

Subject: [PATCH 01/10] memcg: Kernel memory accounting infrastructure.
Posted by Suleiman Souhlal on Mon, 27 Feb 2012 22:58:44 GMT
View Forum Message <> Reply to Message

Enabled with CONFIG_CGROUP_MEM_RES_CTLR_KMEM.

Adds the following files:
- memory.kmem.independent_kmem_limit
- memory.kmem.usage_in_bytes
- memory.kmem.limit_in_bytes

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

```
 mm/memcontrol.c |  121 ++++++++++++++++++++++++++++++++++++++++++++++++++++++++-
 1 files changed, 120 insertions(+), 1 deletions(-)

diff --git a/mm/memcontrol.c b/mm/memcontrol.c
index 228d646..11e31d6 100644
--- a/mm/memcontrol.c
+++ b/mm/memcontrol.c
@@ -235,6 +235,10 @@ struct mem_cgroup {
   */
  struct res_counter memsw;
  /*
+  * the counter to account for kernel memory usage.
+  */
+ struct res_counter kmem_bytes;
+ /*
   * Per cgroup active and inactive list, similar to the
   * per zone LRU lists.
   */
@@ -293,6 +297,7 @@ struct mem_cgroup {
 #ifdef CONFIG_INET
  struct tcp_memcontrol tcp_mem;
 #endif
+ int independent_kmem_limit;
 };

 /* Stuffs for move charges at task migration. */
@@ -354,6 +359,7 @@ enum charge_type {
 #define _MEM   (0)
 #define _MEMSWAP  (1)
 #define _OOM_TYPE  (2)
+#define _KMEM   (3)
```

```
 #define MEMFILE_PRIVATE(x, val) (((x) << 16) | (val))
 #define MEMFILE_TYPE(val) (((val) >> 16) & 0xffff)
 #define MEMFILE_ATTR(val) ((val) & 0xffff)
@@ -370,6 +376,8 @@ enum charge_type {

 static void mem_cgroup_get(struct mem_cgroup *memcg);
 static void mem_cgroup_put(struct mem_cgroup *memcg);
+static void memcg_kmem_init(struct mem_cgroup *memcg,
+    struct mem_cgroup *parent);

 /* Writing them here to avoid exposing memcg's inner layout */
 #ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
@@ -1402,6 +1410,10 @@ 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_bytes, RES_USAGE) >> 10,
+  res_counter_read_u64(&memcg->kmem_bytes, RES_LIMIT) >> 10,
+  res_counter_read_u64(&memcg->kmem_bytes, RES_FAILCNT));
 }

 /*
@@ -3840,6 +3852,9 @@ static u64 mem_cgroup_read(struct cgroup *cont, struct cftype *cft)
   else
    val = res_counter_read_u64(&memcg->memsw, name);
   break;
+ case _KMEM:
+  val = res_counter_read_u64(&memcg->kmem_bytes, name);
+  break;
  default:
   BUG();
   break;
@@ -3872,8 +3887,14 @@ static int mem_cgroup_write(struct cgroup *cont, struct cftype *cft,
    break;
   if (type == _MEM)
    ret = mem_cgroup_resize_limit(memcg, val);
-  else
+  else if (type == _MEMSWAP)
    ret = mem_cgroup_resize_memsw_limit(memcg, val);
+  else if (type == _KMEM) {
+   if (!memcg->independent_kmem_limit)
+    return -EINVAL;
+   ret = res_counter_set_limit(&memcg->kmem_bytes, val);
+  } else
+   return -EINVAL;
   break;
  case RES_SOFT_LIMIT:
```

```
  ret = res_counter_memparse_write_strategy(buffer, &val);
@@ -4572,8 +4593,47 @@ static int mem_control_numa_stat_open(struct inode *unused, struct
file *file)
 #endif /* CONFIG_NUMA */

 #ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
+static u64
+mem_cgroup_independent_kmem_limit_read(struct cgroup *cgrp, struct cftype *cft)
+{
+ return mem_cgroup_from_cont(cgrp)->independent_kmem_limit;
+}
+
+static int mem_cgroup_independent_kmem_limit_write(struct cgroup *cgrp,
+    struct cftype *cft, u64 val)
+{
+ mem_cgroup_from_cont(cgrp)->independent_kmem_limit = !!val;
+
+ return 0;
+}
+
+static struct cftype kmem_cgroup_files[] = {
+ {
+  .name = "kmem.independent_kmem_limit",
+  .write_u64 = mem_cgroup_independent_kmem_limit_write,
+  .read_u64 = mem_cgroup_independent_kmem_limit_read,
+ },
+ {
+  .name = "kmem.limit_in_bytes",
+  .private = MEMFILE_PRIVATE(_KMEM, RES_LIMIT),
+  .write_string = mem_cgroup_write,
+  .read_u64 = mem_cgroup_read,
+ },
+ {
+  .name = "kmem.usage_in_bytes",
+  .private = MEMFILE_PRIVATE(_KMEM, RES_USAGE),
+  .read_u64 = mem_cgroup_read,
+ },
+};
+
 static int register_kmem_files(struct cgroup *cont, struct cgroup_subsys *ss)
 {
+ int ret;
+
+ ret = cgroup_add_files(cont, ss, kmem_cgroup_files,
+     ARRAY_SIZE(kmem_cgroup_files));
+ if (ret)
+  return ret;
  /*
```

```
   * Part of this would be better living in a separate allocation
   * function, leaving us with just the cgroup tree population work.
@@ -4587,6 +4647,10 @@ static int register_kmem_files(struct cgroup *cont, struct
cgroup_subsys *ss)
 static void kmem_cgroup_destroy(struct cgroup_subsys *ss,
     struct cgroup *cont)
 {
+ struct mem_cgroup *memcg;
+
+ memcg = mem_cgroup_from_cont(cont);
+ BUG_ON(res_counter_read_u64(&memcg->kmem_bytes, RES_USAGE) != 0);
  mem_cgroup_sockets_destroy(cont, ss);
 }
 #else
@@ -4938,6 +5002,7 @@ mem_cgroup_create(struct cgroup_subsys *ss, struct cgroup *cont)
  }
  memcg->last_scanned_node = MAX_NUMNODES;
  INIT_LIST_HEAD(&memcg->oom_notify);
+ memcg_kmem_init(memcg, parent && parent->use_hierarchy ? parent : NULL);

  if (parent)
   memcg->swappiness = mem_cgroup_swappiness(parent);
@@ -5519,3 +5584,57 @@ static int __init enable_swap_account(char *s)
 __setup("swapaccount=", enable_swap_account);

 #endif
+
+#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
+int
+memcg_charge_kmem(struct mem_cgroup *memcg, gfp_t gfp, long long delta)
+{
+ struct res_counter *fail_res;
+ struct mem_cgroup *_memcg;
+ int may_oom, ret;
+
+ may_oom = (gfp & __GFP_WAIT) && (gfp & __GFP_FS) &&
+     !(gfp & __GFP_NORETRY);
+
+ ret = 0;
+
+ if (memcg && !memcg->independent_kmem_limit) {
+   _memcg = memcg;
+   if (__mem_cgroup_try_charge(NULL, gfp, delta / PAGE_SIZE,
+       &_memcg, may_oom) != 0)
+    return -ENOMEM;
+ }
+
+ if (_memcg)
```

```
+  ret = res_counter_charge(&_memcg->kmem_bytes, delta, &fail_res);
+
+ return ret;
+}
+
+void
+memcg_uncharge_kmem(struct mem_cgroup *memcg, long long delta)
+{
+ if (memcg)
+  res_counter_uncharge(&memcg->kmem_bytes, delta);
+
+ if (memcg && !memcg->independent_kmem_limit)
+  res_counter_uncharge(&memcg->res, delta);
+}
+
+static void
+memcg_kmem_init(struct mem_cgroup *memcg, struct mem_cgroup *parent)
+{
+ struct res_counter *parent_res;
+
+ parent_res = NULL;
+ if (parent && parent != root_mem_cgroup)
+  parent_res = &parent->kmem_bytes;
+ res_counter_init(&memcg->kmem_bytes, parent_res);
+ memcg->independent_kmem_limit = 0;
+}
+#else /* CONFIG_CGROUP_MEM_RES_CTLR_KMEM */
+static void
+memcg_kmem_init(struct mem_cgroup *memcg, struct mem_cgroup *parent)
+{
+}
+#endif /* CONFIG_CGROUP_MEM_RES_CTLR_KMEM */
--
1.7.7.3
```

Subject: [PATCH 02/10] memcg: Uncharge all kmem when deleting a cgroup.
Posted by Suleiman Souhlal on Mon, 27 Feb 2012 22:58:45 GMT
View Forum Message <> Reply to Message

A later patch will also use this to move the accounting to the root
cgroup.

Signed-off-by: Suleiman Souhlal <suleiman@google.com>
---
 mm/memcontrol.c | 30 ++++++++++++++++++++++++++++-
 1 files changed, 29 insertions(+), 1 deletions(-)

```
diff --git a/mm/memcontrol.c b/mm/memcontrol.c
index 11e31d6..6f44fcb 100644
--- a/mm/memcontrol.c
+++ b/mm/memcontrol.c
@@ -378,6 +378,7 @@ static void mem_cgroup_get(struct mem_cgroup *memcg);
 static void mem_cgroup_put(struct mem_cgroup *memcg);
 static void memcg_kmem_init(struct mem_cgroup *memcg,
     struct mem_cgroup *parent);
+static void memcg_kmem_move(struct mem_cgroup *memcg);

 /* Writing them here to avoid exposing memcg's inner layout */
 #ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
@@ -3674,6 +3675,7 @@ static int mem_cgroup_force_empty(struct mem_cgroup *memcg, bool
free_all)
  int ret;
  int node, zid, shrink;
  int nr_retries = MEM_CGROUP_RECLAIM_RETRIES;
+ unsigned long usage;
  struct cgroup *cgrp = memcg->css.cgroup;

  css_get(&memcg->css);
@@ -3693,6 +3695,8 @@ move_account:
   /* This is for making all *used* pages to be on LRU. */
   lru_add_drain_all();
   drain_all_stock_sync(memcg);
+  if (!free_all)
+   memcg_kmem_move(memcg);
   ret = 0;
   mem_cgroup_start_move(memcg);
   for_each_node_state(node, N_HIGH_MEMORY) {
@@ -3714,8 +3718,13 @@ move_account:
   if (ret == -ENOMEM)
    goto try_to_free;
   cond_resched();
+  usage = memcg->res.usage;
+#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
+  if (free_all && !memcg->independent_kmem_limit)
+   usage -= memcg->kmem_bytes.usage;
+#endif
  /* "ret" should also be checked to ensure all lists are empty. */
- } while (memcg->res.usage > 0 || ret);
+ } while (usage > 0 || ret);
 out:
  css_put(&memcg->css);
  return ret;
@@ -5632,9 +5641,28 @@ memcg_kmem_init(struct mem_cgroup *memcg, struct mem_cgroup
*parent)
  res_counter_init(&memcg->kmem_bytes, parent_res);
```

```
  memcg->independent_kmem_limit = 0;
 }
+
+static void
+memcg_kmem_move(struct mem_cgroup *memcg)
+{
+ unsigned long flags;
+ long kmem_bytes;
+
+ spin_lock_irqsave(&memcg->kmem_bytes.lock, flags);
+ kmem_bytes = memcg->kmem_bytes.usage;
+ res_counter_uncharge_locked(&memcg->kmem_bytes, kmem_bytes);
+ spin_unlock_irqrestore(&memcg->kmem_bytes.lock, flags);
+ if (!memcg->independent_kmem_limit)
+  res_counter_uncharge(&memcg->res, kmem_bytes);
+}
 #else /* CONFIG_CGROUP_MEM_RES_CTLR_KMEM */
 static void
 memcg_kmem_init(struct mem_cgroup *memcg, struct mem_cgroup *parent)
 {
 }
+
+static void
+memcg_kmem_move(struct mem_cgroup *memcg)
+{
+}
 #endif /* CONFIG_CGROUP_MEM_RES_CTLR_KMEM */
--
1.7.7.3
```

Subject: [PATCH 03/10] memcg: Reclaim when more than one page needed.
Posted by Suleiman Souhlal on Mon, 27 Feb 2012 22:58:46 GMT
View Forum Message <> Reply to Message

From: Hugh Dickins <hughd@google.com>

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

Fix that by passing down a minsize argument in addition to the csize;
and pass minsize to consume_stock() also, so that it can draw on stock
for higher order slabs, instead of accumulating an increasing surplus
of stock, as its "nr_pages == 1" tests previously caused.

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

Signed-off-by: Hugh Dickins <hughd@google.com>
Signed-off-by: Suleiman Souhlal <suleiman@google.com>
---
 mm/memcontrol.c | 35 +++++++++++++++++++++----------------
 1 files changed, 19 insertions(+), 16 deletions(-)

```
diff --git a/mm/memcontrol.c b/mm/memcontrol.c
index 6f44fcb..c82ca1c 100644
--- a/mm/memcontrol.c
+++ b/mm/memcontrol.c
@@ -1928,19 +1928,19 @@ static DEFINE_PER_CPU(struct memcg_stock_pcp,
memcg_stock);
 static DEFINE_MUTEX(percpu_charge_mutex);

 /*
- * Try to consume stocked charge on this cpu. If success, one page is consumed
- * from local stock and true is returned. If the stock is 0 or charges from a
- * cgroup which is not current target, returns false. This stock will be
- * refilled.
+ * Try to consume stocked charge on this cpu. If success, nr_pages pages are
+ * consumed from local stock and true is returned. If the stock is 0 or
+ * charges from a cgroup which is not current target, returns false.
+ * This stock will be refilled.
  */
-static bool consume_stock(struct mem_cgroup *memcg)
+static bool consume_stock(struct mem_cgroup *memcg, int nr_pages)
 {
  struct memcg_stock_pcp *stock;
  bool ret = true;

  stock = &get_cpu_var(memcg_stock);
- if (memcg == stock->cached && stock->nr_pages)
-  stock->nr_pages--;
+ if (memcg == stock->cached && stock->nr_pages >= nr_pages)
+  stock->nr_pages -= nr_pages;
  else /* need to call res_counter_charge */
   ret = false;
  put_cpu_var(memcg_stock);
@@ -2131,7 +2131,7 @@ enum {
 };
```

```
 static int mem_cgroup_do_charge(struct mem_cgroup *memcg, gfp_t gfp_mask,
-    unsigned int nr_pages, bool oom_check)
+    unsigned int nr_pages, unsigned int min_pages, bool oom_check)
 {
  unsigned long csize = nr_pages * PAGE_SIZE;
  struct mem_cgroup *mem_over_limit;
@@ -2154,18 +2154,18 @@ static int mem_cgroup_do_charge(struct mem_cgroup *memcg,
gfp_t gfp_mask,
  } else
   mem_over_limit = mem_cgroup_from_res_counter(fail_res, res);
  /*
-  * nr_pages can be either a huge page (HPAGE_PMD_NR), a batch
-  * of regular pages (CHARGE_BATCH), or a single regular page (1).
-  *
   * Never reclaim on behalf of optional batching, retry with a
   * single page instead.
   */
- if (nr_pages == CHARGE_BATCH)
+ if (nr_pages > min_pages)
   return CHARGE_RETRY;

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

+ if (gfp_mask & __GFP_NORETRY)
+  return CHARGE_NOMEM;
+
  ret = mem_cgroup_reclaim(mem_over_limit, gfp_mask, flags);
  if (mem_cgroup_margin(mem_over_limit) >= nr_pages)
   return CHARGE_RETRY;
@@ -2178,8 +2178,10 @@ static int mem_cgroup_do_charge(struct mem_cgroup *memcg, gfp_t
gfp_mask,
   * unlikely to succeed so close to the limit, and we fall back
   * to regular pages anyway in case of failure.
   */
- if (nr_pages == 1 && ret)
+ if (nr_pages <= (PAGE_SIZE << PAGE_ALLOC_COSTLY_ORDER) && ret) {
+  cond_resched();
   return CHARGE_RETRY;
+ }

  /*
   * At task move, charge accounts can be doubly counted. So, it's
@@ -2253,7 +2255,7 @@ again:
   VM_BUG_ON(css_is_removed(&memcg->css));
   if (mem_cgroup_is_root(memcg))
    goto done;
-  if (nr_pages == 1 && consume_stock(memcg))
```

```
+  if (consume_stock(memcg, nr_pages))
    goto done;
   css_get(&memcg->css);
  } else {
@@ -2278,7 +2280,7 @@ again:
    rcu_read_unlock();
    goto done;
   }
-  if (nr_pages == 1 && consume_stock(memcg)) {
+  if (consume_stock(memcg, nr_pages)) {
    /*
     * It seems dagerous to access memcg without css_get().
     * But considering how consume_stok works, it's not
@@ -2313,7 +2315,8 @@ again:
    nr_oom_retries = MEM_CGROUP_RECLAIM_RETRIES;
   }

-  ret = mem_cgroup_do_charge(memcg, gfp_mask, batch, oom_check);
+  ret = mem_cgroup_do_charge(memcg, gfp_mask, batch, nr_pages,
+      oom_check);
   switch (ret) {
   case CHARGE_OK:
    break;
--
1.7.7.3
```

Subject: [PATCH 04/10] memcg: Introduce __GFP_NOACCOUNT.
Posted by Suleiman Souhlal on Mon, 27 Feb 2012 22:58:47 GMT
View Forum Message <> Reply to Message

This is used to indicate that we don't want an allocation to be accounted
to the current cgroup.

Signed-off-by: Suleiman Souhlal <suleiman@google.com>
---
 include/linux/gfp.h | 2 ++
 1 files changed, 2 insertions(+), 0 deletions(-)

```
diff --git a/include/linux/gfp.h b/include/linux/gfp.h
index 581e74b..765c20f 100644
--- a/include/linux/gfp.h
+++ b/include/linux/gfp.h
@@ -23,6 +23,7 @@ struct vm_area_struct;
 #define ___GFP_REPEAT  0x400u
 #define ___GFP_NOFAIL  0x800u
 #define ___GFP_NORETRY  0x1000u
+#define ___GFP_NOACCOUNT 0x2000u
```

```
 #define ___GFP_COMP  0x4000u
 #define ___GFP_ZERO  0x8000u
 #define ___GFP_NOMEMALLOC 0x10000u
@@ -76,6 +77,7 @@ struct vm_area_struct;
 #define __GFP_REPEAT ((__force gfp_t)___GFP_REPEAT) /* See above */
 #define __GFP_NOFAIL ((__force gfp_t)___GFP_NOFAIL) /* See above */
 #define __GFP_NORETRY ((__force gfp_t)___GFP_NORETRY) /* See above */
+#define __GFP_NOACCOUNT ((__force gfp_t)___GFP_NOACCOUNT) /* Don't account to the
current cgroup */
 #define __GFP_COMP ((__force gfp_t)___GFP_COMP) /* Add compound page metadata */
 #define __GFP_ZERO ((__force gfp_t)___GFP_ZERO) /* Return zeroed page on success */
 #define __GFP_NOMEMALLOC ((__force gfp_t)___GFP_NOMEMALLOC) /* Don't use
emergency reserves */
--
1.7.7.3
```

Subject: [PATCH 05/10] memcg: Slab accounting.
Posted by Suleiman Souhlal on Mon, 27 Feb 2012 22:58:48 GMT
View Forum Message <> Reply to Message

Introduce per-cgroup kmem_caches for memcg slab accounting, that
get created the first time we do an allocation of that type in the
cgroup.
If we are not permitted to sleep in that allocation, the cache
gets created asynchronously.
The cgroup cache gets used in subsequent allocations, and permits
accounting of slab on a per-page basis.

The per-cgroup kmem_caches get looked up at slab allocation time,
in a MAX_KMEM_CACHE_TYPES-sized array in the memcg structure, based
on the original kmem_cache's id, which gets allocated when the original
cache gets created.

Allocations that cannot be attributed to a cgroup get charged to
the root cgroup.

Each cgroup kmem_cache has a refcount that dictates the lifetime
of the cache: We destroy a cgroup cache when its cgroup has been
destroyed and there are no more active objects in the cache.

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

```
 include/linux/memcontrol.h |   30 ++++-
 include/linux/slab.h       |    1 +
 include/linux/slab_def.h   |   94 +++++++++++++-
 mm/memcontrol.c            |  316 ++++++++++++++++++++++++++++++++++++++++++++++++
 mm/slab.c                  |  266 +++++++++++++++++++++++++++++++++++++----
```

```
 5 files changed, 680 insertions(+), 27 deletions(-)

diff --git a/include/linux/memcontrol.h b/include/linux/memcontrol.h
index 4d34356..f5458b0 100644
--- a/include/linux/memcontrol.h
+++ b/include/linux/memcontrol.h
@@ -421,13 +421,41 @@ struct sock;
 #ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
 void sock_update_memcg(struct sock *sk);
 void sock_release_memcg(struct sock *sk);
-#else
+struct kmem_cache *mem_cgroup_get_kmem_cache(struct kmem_cache *cachep,
+    gfp_t gfp);
+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);
+void mem_cgroup_flush_cache_create_queue(void);
+void mem_cgroup_remove_child_kmem_cache(struct kmem_cache *cachep, int id);
+#else /* CONFIG_CGROUP_MEM_RES_CTLR_KMEM */
 static inline void sock_update_memcg(struct sock *sk)
 {
 }
 static inline void sock_release_memcg(struct sock *sk)
 {
 }
+
+static inline bool
+mem_cgroup_charge_slab(struct kmem_cache *cachep, gfp_t gfp, size_t size)
+{
+ return true;
+}
+
+static inline void
+mem_cgroup_uncharge_slab(struct kmem_cache *cachep, size_t size)
+{
+}
+
+static inline struct kmem_cache *
+mem_cgroup_get_kmem_cache(struct kmem_cache *cachep, gfp_t gfp)
+{
+ return cachep;
+}
+
+static inline void
+mem_cgroup_flush_cache_create_queue(void)
+{
+}
 #endif /* CONFIG_CGROUP_MEM_RES_CTLR_KMEM */
 #endif /* _LINUX_MEMCONTROL_H */
```

```
diff --git a/include/linux/slab.h b/include/linux/slab.h
index 573c809..fe21a91 100644
--- a/include/linux/slab.h
+++ b/include/linux/slab.h
@@ -21,6 +21,7 @@
 #define SLAB_POISON  0x00000800UL /* DEBUG: Poison objects */
 #define SLAB_HWCACHE_ALIGN 0x00002000UL /* Align objs on cache lines */
 #define SLAB_CACHE_DMA  0x00004000UL /* Use GFP_DMA memory */
+#define SLAB_MEMCG  0x00008000UL /* memcg kmem_cache */
 #define SLAB_STORE_USER  0x00010000UL /* DEBUG: Store the last owner for bug hunting */
 #define SLAB_PANIC  0x00040000UL /* Panic if kmem_cache_create() fails */
 /*
diff --git a/include/linux/slab_def.h b/include/linux/slab_def.h
index fbd1117..449a0de 100644
--- a/include/linux/slab_def.h
+++ b/include/linux/slab_def.h
@@ -41,6 +41,10 @@ struct kmem_cache {
  /* force GFP flags, e.g. GFP_DMA */
  gfp_t gfpflags;

+#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
+ int id;    /* id used for slab accounting */
+#endif
+
  size_t colour;   /* cache colouring range */
  unsigned int colour_off; /* colour offset */
  struct kmem_cache *slabp_cache;
@@ -51,7 +55,7 @@ struct kmem_cache {
  void (*ctor)(void *obj);

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

 /* 5) statistics */
@@ -78,9 +82,26 @@ 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
+
```

```
+#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
+ /* Original cache parameters, used when creating a memcg cache */
+ size_t orig_align;
+ unsigned long orig_flags;
+
+ struct mem_cgroup *memcg;
+
+ /* Who we copied from when creating cpuset cache */
+ struct kmem_cache *orig_cache;
+
+ atomic_t refcnt;
+ struct list_head destroyed_list; /* Used when deleting cpuset cache */
+#endif /* CONFIG_CGROUP_MEM_RES_CTLR_KMEM */
+
 /* 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
@@ -212,4 +233,73 @@ found:

 #endif /* CONFIG_NUMA */

+#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
+
+#define MAX_KMEM_CACHE_TYPES 300
+
+struct kmem_cache *kmem_cache_create_memcg(struct kmem_cache *cachep,
+    char *name);
+void kmem_cache_destroy_cpuset(struct kmem_cache *cachep);
+void kmem_cache_drop_ref(struct kmem_cache *cachep);
+
+static inline void
+kmem_cache_get_ref(struct kmem_cache *cachep)
+{
+ if ((cachep->flags & SLAB_MEMCG) &&
+     unlikely(!atomic_add_unless(&cachep->refcnt, 1, 0)))
+  BUG();
+}
+
+static inline void
+mem_cgroup_put_kmem_cache(struct kmem_cache *cachep)
+{
+ rcu_read_unlock();
+}
+
+static inline void
+mem_cgroup_kmem_cache_prepare_sleep(struct kmem_cache *cachep)
+{
+ /*
```

```
+  * Make sure the cache doesn't get freed while we have interrupts
+  * enabled.
+  */
+ kmem_cache_get_ref(cachep);
+ rcu_read_unlock();
+}
+
+static inline void
+mem_cgroup_kmem_cache_finish_sleep(struct kmem_cache *cachep)
+{
+ rcu_read_lock();
+ kmem_cache_drop_ref(cachep);
+}
+
+#else /* CONFIG_CGROUP_MEM_RES_CTLR_KMEM */
+
+static inline void
+kmem_cache_get_ref(struct kmem_cache *cachep)
+{
+}
+
+static inline void
+kmem_cache_drop_ref(struct kmem_cache *cachep)
+{
+}
+
+static inline void
+mem_cgroup_put_kmem_cache(struct kmem_cache *cachep)
+{
+}
+
+static inline void
+mem_cgroup_kmem_cache_prepare_sleep(struct kmem_cache *cachep)
+{
+}
+
+static inline void
+mem_cgroup_kmem_cache_finish_sleep(struct kmem_cache *cachep)
+{
+}
+#endif /* CONFIG_CGROUP_MEM_RES_CTLR_KMEM */
+
 #endif /* _LINUX_SLAB_DEF_H */
diff --git a/mm/memcontrol.c b/mm/memcontrol.c
index c82ca1c..d1c0cd7 100644
--- a/mm/memcontrol.c
+++ b/mm/memcontrol.c
@@ -297,6 +297,11 @@ struct mem_cgroup {
```

```
 #ifdef CONFIG_INET
  struct tcp_memcontrol tcp_mem;
 #endif
+
+#if defined(CONFIG_CGROUP_MEM_RES_CTLR_KMEM) && defined(CONFIG_SLAB)
+ /* Slab accounting */
+ struct kmem_cache *slabs[MAX_KMEM_CACHE_TYPES];
+#endif
  int independent_kmem_limit;
 };

@@ -5633,6 +5638,312 @@ memcg_uncharge_kmem(struct mem_cgroup *memcg, long long
delta)
   res_counter_uncharge(&memcg->res, delta);
 }

+#ifdef CONFIG_SLAB
+static struct kmem_cache *
+memcg_create_kmem_cache(struct mem_cgroup *memcg, int idx,
+    struct kmem_cache *cachep, gfp_t gfp)
+{
+ struct kmem_cache *new_cachep;
+ struct dentry *dentry;
+ char *name;
+ int len;
+
+ if ((gfp & GFP_KERNEL) != GFP_KERNEL)
+  return cachep;
+
+ dentry = memcg->css.cgroup->dentry;
+ BUG_ON(dentry == NULL);
+ len = strlen(cachep->name);
+ len += dentry->d_name.len;
+ len += 7; /* Space for "()", NUL and appending "dead" */
+ name = kmalloc(len, GFP_KERNEL | __GFP_NOACCOUNT);
+
+ if (name == NULL)
+  return cachep;
+
+ snprintf(name, len, "%s(%s)", cachep->name,
+     dentry ? (const char *)dentry->d_name.name : "/");
+ name[len - 5] = '\0'; /* Make sure we can append "dead" later */
+
+ new_cachep = kmem_cache_create_memcg(cachep, name);
+
+ /*
+  * Another CPU is creating the same cache?
+  * We'll use it next time.
```

```
+  */
+ if (new_cachep == NULL) {
+  kfree(name);
+  return cachep;
+ }
+
+ new_cachep->memcg = memcg;
+
+ /*
+  * Make sure someone else hasn't created the new cache in the
+  * meantime.
+  * This should behave as a write barrier, so we should be fine
+  * with RCU.
+  */
+ if (cmpxchg(&memcg->slabs[idx], NULL, new_cachep) != NULL) {
+  kmem_cache_destroy(new_cachep);
+  return cachep;
+ }
+
+ return new_cachep;
+}
+
+struct create_work {
+ struct mem_cgroup *memcg;
+ struct kmem_cache *cachep;
+ struct list_head list;
+};
+
+static DEFINE_SPINLOCK(create_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(&create_queue_lock, flags);
+ list_for_each_entry_safe(cw, tmp, &create_queue, list) {
+  list_del(&cw->list);
+  kfree(cw);
```

```
+ }
+ spin_unlock_irqrestore(&create_queue_lock, flags);
+}
+
+static void
+memcg_create_cache_work_func(struct work_struct *w)
+{
+ struct kmem_cache *cachep;
+ struct create_work *cw;
+
+ spin_lock_irq(&create_queue_lock);
+ while (!list_empty(&create_queue)) {
+  cw = list_first_entry(&create_queue, struct create_work, list);
+  list_del(&cw->list);
+  spin_unlock_irq(&create_queue_lock);
+  cachep = memcg_create_kmem_cache(cw->memcg, cw->cachep->id,
+      cw->cachep, GFP_KERNEL);
+  if (cachep == NULL && printk_ratelimit())
+   printk(KERN_ALERT "%s: Couldn't create memcg-cache for"
+       " %s memcg %s\n", __func__, cw->cachep->name,
+       cw->memcg->css.cgroup->dentry->d_name.name);
+  kfree(cw);
+  spin_lock_irq(&create_queue_lock);
+ }
+ spin_unlock_irq(&create_queue_lock);
+}
+
+static DECLARE_WORK(memcg_create_cache_work, memcg_create_cache_work_func);
+
+static void
+memcg_create_cache_enqueue(struct mem_cgroup *memcg, struct kmem_cache *cachep)
+{
+ struct create_work *cw;
+ unsigned long flags;
+
+ spin_lock_irqsave(&create_queue_lock, flags);
+ list_for_each_entry(cw, &create_queue, list) {
+  if (cw->memcg == memcg && cw->cachep == cachep) {
+   spin_unlock_irqrestore(&create_queue_lock, flags);
+   return;
+  }
+ }
+ spin_unlock_irqrestore(&create_queue_lock, flags);
+
+ cw = kmalloc(sizeof(struct create_work), GFP_NOWAIT | __GFP_NOACCOUNT);
+ if (cw == NULL)
+  return;
+
```

```
+ cw->memcg = memcg;
+ cw->cachep = cachep;
+ spin_lock_irqsave(&create_queue_lock, flags);
+ list_add_tail(&cw->list, &create_queue);
+ spin_unlock_irqrestore(&create_queue_lock, flags);
+
+ schedule_work(&memcg_create_cache_work);
+}
+
+/*
+ * Return the kmem_cache we're supposed to use for a slab allocation.
+ * If we are in interrupt context or otherwise have an allocation that
+ * can't fail, we return the original cache.
+ * Otherwise, we will try to use the current memcg's version of the cache.
+ *
+ * If the cache does not exist yet, if we are the first user of it,
+ * we either create it immediately, if possible, or create it asynchronously
+ * in a workqueue.
+ * In the latter case, we will let the current allocation go through with
+ * the original cache.
+ *
+ * This function returns with rcu_read_lock() held.
+ */
+struct kmem_cache *
+mem_cgroup_get_kmem_cache(struct kmem_cache *cachep, gfp_t gfp)
+{
+ struct kmem_cache *ret;
+ struct mem_cgroup *memcg;
+ int idx;
+
+ rcu_read_lock();
+
+ if (in_interrupt())
+  return cachep;
+ if (current == NULL)
+  return cachep;
+
+ gfp |= cachep->gfpflags;
+ if ((gfp & __GFP_NOACCOUNT) || (gfp & __GFP_NOFAIL))
+  return cachep;
+
+ if (cachep->flags & SLAB_MEMCG)
+  return cachep;
+
+ memcg = mem_cgroup_from_task(current);
+ idx = cachep->id;
+
+ if (memcg == NULL || memcg == root_mem_cgroup)
```

```
+  return cachep;
+
+ VM_BUG_ON(idx == -1);
+
+ if (rcu_access_pointer(memcg->slabs[idx]) == NULL) {
+  if ((gfp & GFP_KERNEL) == GFP_KERNEL) {
+   if (!css_tryget(&memcg->css))
+    return cachep;
+   rcu_read_unlock();
+   ret = memcg_create_kmem_cache(memcg, idx, cachep, gfp);
+   rcu_read_lock();
+   css_put(&memcg->css);
+   return ret;
+  } else {
+   /*
+    * Not a 'normal' slab allocation, so just enqueue
+    * the creation of the memcg cache and let the current
+    * allocation use the normal cache.
+    */
+   memcg_create_cache_enqueue(memcg, cachep);
+   return cachep;
+  }
+ }
+
+ return rcu_dereference(memcg->slabs[idx]);
+}
+
+void
+mem_cgroup_remove_child_kmem_cache(struct kmem_cache *cachep, int id)
+{
+ rcu_assign_pointer(cachep->memcg->slabs[id], NULL);
+}
+
+bool
+mem_cgroup_charge_slab(struct kmem_cache *cachep, gfp_t gfp, size_t size)
+{
+ struct mem_cgroup *memcg;
+ int ret;
+
+ rcu_read_lock();
+ if (cachep->flags & SLAB_MEMCG)
+  memcg = cachep->memcg;
+ else
+  memcg = NULL;
+
+ if (memcg && !css_tryget(&memcg->css))
+  memcg = NULL;
+ rcu_read_unlock();
```

```
+
+ ret = memcg_charge_kmem(memcg, gfp, size);
+ if (memcg)
+  css_put(&memcg->css);
+
+ return ret == 0;
+}
+
+void
+mem_cgroup_uncharge_slab(struct kmem_cache *cachep, size_t size)
+{
+ struct mem_cgroup *memcg;
+
+ rcu_read_lock();
+ if (cachep->flags & SLAB_MEMCG)
+  memcg = cachep->memcg;
+ else
+  memcg = NULL;
+
+ if (memcg && !css_tryget(&memcg->css))
+  memcg = NULL;
+ rcu_read_unlock();
+
+ memcg_uncharge_kmem(memcg, size);
+ if (memcg)
+  css_put(&memcg->css);
+}
+
+static void
+memcg_slab_init(struct mem_cgroup *memcg)
+{
+ int i;
+
+ for (i = 0; i < MAX_KMEM_CACHE_TYPES; i++)
+  rcu_assign_pointer(memcg->slabs[i], NULL);
+}
+
+/*
+ * Mark all of this memcg's kmem_caches as dead and move them to the
+ * root.
+ *
+ * Assumes that the callers are synchronized (only one thread should be
+ * moving a cgroup's slab at the same time).
+ */
+static void
+memcg_slab_move(struct mem_cgroup *memcg)
+{
+ struct kmem_cache *cachep;
```

```
+ int i;
+
+ mem_cgroup_flush_cache_create_queue();
+
+ for (i = 0; i < MAX_KMEM_CACHE_TYPES; i++) {
+  cachep = rcu_access_pointer(memcg->slabs[i]);
+  if (cachep != NULL) {
+   rcu_assign_pointer(memcg->slabs[i], NULL);
+   cachep->memcg = NULL;
+
+   /* The space for this is already allocated */
+   strcat((char *)cachep->name, "dead");
+
+   /*
+    * Drop the initial reference on the cache.
+    * This means that from this point on, the cache will
+    * get destroyed when it no longer has active objects.
+    */
+   kmem_cache_drop_ref(cachep);
+  }
+ }
+}
+#else /* CONFIG_SLAB */
+static void
+memcg_slab_init(struct mem_cgroup *memcg)
+{
+}
+
+static void
+memcg_slab_move(struct mem_cgroup *memcg)
+{
+}
+#endif /* CONFIG_SLAB */
+
 static void
 memcg_kmem_init(struct mem_cgroup *memcg, struct mem_cgroup *parent)
 {
@@ -5642,6 +5953,9 @@ memcg_kmem_init(struct mem_cgroup *memcg, struct mem_cgroup
*parent)
  if (parent && parent != root_mem_cgroup)
   parent_res = &parent->kmem_bytes;
  res_counter_init(&memcg->kmem_bytes, parent_res);
+
+ memcg_slab_init(memcg);
+
  memcg->independent_kmem_limit = 0;
 }
```

```
@@ -5651,6 +5965,8 @@ memcg_kmem_move(struct mem_cgroup *memcg)
  unsigned long flags;
  long kmem_bytes;

+ memcg_slab_move(memcg);
+
  spin_lock_irqsave(&memcg->kmem_bytes.lock, flags);
  kmem_bytes = memcg->kmem_bytes.usage;
  res_counter_uncharge_locked(&memcg->kmem_bytes, kmem_bytes);
diff --git a/mm/slab.c b/mm/slab.c
index f0bd785..6c6bc49 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
@@ -326,6 +328,11 @@ static __always_inline int index_of(const size_t size)
  return 0;
 }

+#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
+/* Bitmap used for allocating the cache id numbers. */
+static DECLARE_BITMAP(cache_types, MAX_KMEM_CACHE_TYPES);
+#endif
+
 static int slab_early_init = 1;

 #define INDEX_AC index_of(sizeof(struct arraycache_init))
@@ -1756,17 +1763,23 @@ 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 (!page) {
+  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);

@@ -1799,6 +1812,8 @@ 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);
@@ -2224,14 +2239,17 @@ 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(const char *name, size_t size, size_t align,
+    unsigned long flags, void (*ctor)(void *), bool memcg)
 {
- size_t left_over, slab_size, ralign;
+ size_t left_over, orig_align, ralign, slab_size;
  struct kmem_cache *cachep = NULL, *pc;
+ unsigned long orig_flags;
  gfp_t gfp;

+ orig_align = align;
+ orig_flags = flags;
  /*
   * Sanity checks... these are all serious usage bugs.
   */
@@ -2248,7 +2266,6 @@ kmem_cache_create (const char *name, size_t size, size_t align,
   */
  if (slab_is_available()) {
```

```
  get_online_cpus();
-  mutex_lock(&cache_chain_mutex);
  }

  list_for_each_entry(pc, &cache_chain, next) {
@@ -2269,10 +2286,12 @@ kmem_cache_create (const char *name, size_t size, size_t align,
  }

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

@@ -2359,9 +2378,9 @@ kmem_cache_create (const char *name, size_t size, size_t align,
  align = ralign;

  if (slab_is_available())
-  gfp = GFP_KERNEL;
+  gfp = GFP_KERNEL | __GFP_NOACCOUNT;
  else
-  gfp = GFP_NOWAIT;
+  gfp = GFP_NOWAIT | __GFP_NOACCOUNT;

  /* Get cache's description obj. */
  cachep = kmem_cache_zalloc(&cache_cache, gfp);
@@ -2369,9 +2388,15 @@ kmem_cache_create (const char *name, size_t size, size_t align,
  goto oops;

  cachep->nodelists = (struct kmem_list3 **)&cachep->array[nr_cpu_ids];
-#if DEBUG
+
+#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
  cachep->obj_size = size;
+ cachep->orig_align = orig_align;
+ cachep->orig_flags = orig_flags;
+#endif

+#if DEBUG
+ cachep->obj_size = size;
```

```
 /*
  * Both debugging options require word-alignment which is calculated
  * into align above.
@@ -2477,7 +2502,23 @@ kmem_cache_create (const char *name, size_t size, size_t align,
  BUG_ON(ZERO_OR_NULL_PTR(cachep->slabp_cache));
 }
 cachep->ctor = ctor;
- cachep->name = name;
+ cachep->name = (char *)name;
+
+#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
+ cachep->orig_cache = NULL;
+ atomic_set(&cachep->refcnt, 1);
+ INIT_LIST_HEAD(&cachep->destroyed_list);
+
+ if (!memcg) {
+  int id;
+
+  id = find_first_zero_bit(cache_types, MAX_KMEM_CACHE_TYPES);
+  BUG_ON(id < 0 || id >= MAX_KMEM_CACHE_TYPES);
+  __set_bit(id, cache_types);
+  cachep->id = id;
+ } else
+  cachep->id = -1;
+#endif

 if (setup_cpu_cache(cachep, gfp)) {
  __kmem_cache_destroy(cachep);
@@ -2502,13 +2543,54 @@ oops:
  panic("kmem_cache_create(): failed to create slab `%s'\n",
        name);
 if (slab_is_available()) {
-  mutex_unlock(&cache_chain_mutex);
  put_online_cpus();
 }
 return cachep;
}
+
+struct kmem_cache *
+kmem_cache_create(const char *name, size_t size, size_t align,
+    unsigned long flags, void (*ctor)(void *))
+{
+ struct kmem_cache *cachep;
+
+ mutex_lock(&cache_chain_mutex);
+ cachep = __kmem_cache_create(name, size, align, flags, ctor, false);
+ mutex_unlock(&cache_chain_mutex);
+
```

```
+ return cachep;
+}
 EXPORT_SYMBOL(kmem_cache_create);

+#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
+struct kmem_cache *
+kmem_cache_create_memcg(struct kmem_cache *cachep, char *name)
+{
+ struct kmem_cache *new;
+ int flags;
+
+ flags = cachep->orig_flags & ~SLAB_PANIC;
+ mutex_lock(&cache_chain_mutex);
+ new = __kmem_cache_create(name, cachep->obj_size, cachep->orig_align,
+     flags, cachep->ctor, 1);
+ if (new == NULL) {
+  mutex_unlock(&cache_chain_mutex);
+  return NULL;
+ }
+ new->flags |= SLAB_MEMCG;
+ new->orig_cache = cachep;
+
+ 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 | __GFP_NOACCOUNT);
+ mutex_unlock(&cache_chain_mutex);
+
+ return new;
+}
+#endif /* CONFIG_CGROUP_MEM_RES_CTLR_KMEM */
+
 #if DEBUG
 static void check_irq_off(void)
 {
@@ -2703,12 +2785,73 @@ void kmem_cache_destroy(struct kmem_cache *cachep)
  if (unlikely(cachep->flags & SLAB_DESTROY_BY_RCU))
   rcu_barrier();

+#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
+ /* Not a memcg cache */
+ if (cachep->id != -1) {
+  __clear_bit(cachep->id, cache_types);
+  mem_cgroup_flush_cache_create_queue();
+ }
+#endif
  __kmem_cache_destroy(cachep);
```

```
 mutex_unlock(&cache_chain_mutex);
 put_online_cpus();
}
EXPORT_SYMBOL(kmem_cache_destroy);

+#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
+static DEFINE_SPINLOCK(destroy_lock);
+static LIST_HEAD(destroyed_caches);
+
+static void
+kmem_cache_destroy_work_func(struct work_struct *w)
+{
+ struct kmem_cache *cachep;
+ char *name;
+
+ spin_lock_irq(&destroy_lock);
+ while (!list_empty(&destroyed_caches)) {
+  cachep = list_first_entry(&destroyed_caches, struct kmem_cache,
+      destroyed_list);
+  name = (char *)cachep->name;
+  list_del(&cachep->destroyed_list);
+  spin_unlock_irq(&destroy_lock);
+  synchronize_rcu();
+  kmem_cache_destroy(cachep);
+  kfree(name);
+  spin_lock_irq(&destroy_lock);
+ }
+ spin_unlock_irq(&destroy_lock);
+}
+
+static DECLARE_WORK(kmem_cache_destroy_work, kmem_cache_destroy_work_func);
+
+static void
+kmem_cache_destroy_memcg(struct kmem_cache *cachep)
+{
+ unsigned long flags;
+
+ BUG_ON(!(cachep->flags & SLAB_MEMCG));
+
+ /*
+  * We have to defer the actual destroying to a workqueue, because
+  * we might currently be in a context that cannot sleep.
+  */
+ spin_lock_irqsave(&destroy_lock, flags);
+ list_add(&cachep->destroyed_list, &destroyed_caches);
+ spin_unlock_irqrestore(&destroy_lock, flags);
+
+ schedule_work(&kmem_cache_destroy_work);
```

```
+}
+
+void
+kmem_cache_drop_ref(struct kmem_cache *cachep)
+{
+ if ((cachep->flags & SLAB_MEMCG) &&
+     unlikely(atomic_dec_and_test(&cachep->refcnt)))
+  kmem_cache_destroy_memcg(cachep);
+}
+#endif /* CONFIG_CGROUP_MEM_RES_CTLR_KMEM */
+
 /*
  * Get the memory for a slab management obj.
  * For a slab cache when the slab descriptor is off-slab, slab descriptors
@@ -2908,8 +3051,10 @@ 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
@@ -2920,6 +3065,13 @@ static int cache_grow(struct kmem_cache *cachep,
  kmem_flagcheck(cachep, flags);

  /*
+  * alloc_slabmgmt() might invoke the slab allocator itself, so
+  * make sure we don't recurse in slab accounting.
+  */
+ if (flags & __GFP_NOACCOUNT)
+  local_flags |= __GFP_NOACCOUNT;
+
+ /*
   * Get mem for the objs.  Attempt to allocate a physical page from
   * 'nodeid'.
   */
@@ -2938,8 +3090,10 @@ 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);

@@ -2952,8 +3106,10 @@ 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;
 }

@@ -3712,10 +3868,14 @@ 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;
+
+ cachep = mem_cgroup_get_kmem_cache(cachep, flags);
+ ret = __cache_alloc(cachep, flags, __builtin_return_address(0));

  trace_kmem_cache_alloc(_RET_IP_, ret,
          obj_size(cachep), cachep->buffer_size, flags);
+ mem_cgroup_put_kmem_cache(cachep);

  return ret;
 }
@@ -3727,10 +3887,12 @@ kmem_cache_alloc_trace(size_t size, struct kmem_cache *cachep,
gfp_t flags)
 {
  void *ret;

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

  trace_kmalloc(_RET_IP_, ret,
        size, slab_buffer_size(cachep), flags);
+ mem_cgroup_put_kmem_cache(cachep);
  return ret;
 }
 EXPORT_SYMBOL(kmem_cache_alloc_trace);
@@ -3739,12 +3901,16 @@ 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;
+
+ cachep = mem_cgroup_get_kmem_cache(cachep, flags);
+ 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);
+ mem_cgroup_put_kmem_cache(cachep);

  return ret;
 }
@@ -3758,11 +3924,13 @@ void *kmem_cache_alloc_node_trace(size_t size,
 {
  void *ret;

+ cachep = mem_cgroup_get_kmem_cache(cachep, flags);
  ret = __cache_alloc_node(cachep, flags, nodeid,
       __builtin_return_address(0));
  trace_kmalloc_node(_RET_IP_, ret,
       size, slab_buffer_size(cachep),
       flags, nodeid);
+ mem_cgroup_put_kmem_cache(cachep);
  return ret;
 }
 EXPORT_SYMBOL(kmem_cache_alloc_node_trace);
@@ -3772,11 +3940,16 @@ static __always_inline void *
 __do_kmalloc_node(size_t size, gfp_t flags, int node, void *caller)
 {
  struct kmem_cache *cachep;
+ void *ret;

  cachep = kmem_find_general_cachep(size, flags);
  if (unlikely(ZERO_OR_NULL_PTR(cachep)))
   return cachep;
- return kmem_cache_alloc_node_trace(size, cachep, flags, node);
+ cachep = mem_cgroup_get_kmem_cache(cachep, flags);
+ ret = kmem_cache_alloc_node_trace(size, cachep, flags, node);
+ mem_cgroup_put_kmem_cache(cachep);
+
+ return ret;
 }
```

```
 #if defined(CONFIG_DEBUG_SLAB) || defined(CONFIG_TRACING)
@@ -3822,10 +3995,12 @@ static __always_inline void *__do_kmalloc(size_t size, gfp_t flags,
  cachep = __find_general_cachep(size, flags);
  if (unlikely(ZERO_OR_NULL_PTR(cachep)))
   return cachep;
+ cachep = mem_cgroup_get_kmem_cache(cachep, flags);
  ret = __cache_alloc(cachep, flags, caller);

  trace_kmalloc((unsigned long) caller, ret,
        size, cachep->buffer_size, flags);
+ mem_cgroup_put_kmem_cache(cachep);

  return ret;
 }
@@ -3866,9 +4041,34 @@ 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->flags & SLAB_MEMCG));
+   VM_BUG_ON(actual_cachep->orig_cache != cachep);
+   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);
@@ -3896,9 +4096,19 @@ 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);
@@ -4167,6 +4377,13 @@ 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->flags & SLAB_MEMCG) &&
+      !atomic_add_unless(&searchp->refcnt, 1, 0))
+   continue;
+#endif
+
  /*
   * We only take the l3 lock if absolutely necessary and we
   * have established with reasonable certainty that
@@ -4199,6 +4416,7 @@ static void cache_reap(struct work_struct *w)
   STATS_ADD_REAPED(searchp, freed);
  }
next:
+  kmem_cache_drop_ref(searchp);
  cond_resched();
 }
 check_irq_on();
@@ -4412,8 +4630,8 @@ static ssize_t slabinfo_write(struct file *file, const char __user *buffer,
    res = 0;
   } else {
    res = do_tune_cpucache(cachep, limit,
-            batchcount, shared,
```

```
-           GFP_KERNEL);
+       batchcount, shared, GFP_KERNEL |
+       __GFP_NOACCOUNT);
   }
   break;
  }
--
1.7.7.3
```

Subject: [PATCH 06/10] memcg: Track all the memcg children of a kmem_cache.
Posted by Suleiman Souhlal on Mon, 27 Feb 2012 22:58:49 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.

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

```
---
 include/linux/slab_def.h |    1 +
 mm/slab.c                |   45 +++++++++++++++++++++++++++++++++++++++++----
 2 files changed, 42 insertions(+), 4 deletions(-)

diff --git a/include/linux/slab_def.h b/include/linux/slab_def.h
index 449a0de..185e4a2 100644
--- a/include/linux/slab_def.h
+++ b/include/linux/slab_def.h
@@ -100,6 +100,7 @@ struct kmem_cache {

  atomic_t refcnt;
  struct list_head destroyed_list; /* Used when deleting cpuset cache */
+ struct list_head sibling_list;
 #endif /* CONFIG_CGROUP_MEM_RES_CTLR_KMEM */

 /* 6) per-cpu/per-node data, touched during every alloc/free */
diff --git a/mm/slab.c b/mm/slab.c
index 6c6bc49..bf38af6 100644
--- a/mm/slab.c
+++ b/mm/slab.c
@@ -2508,6 +2508,7 @@ __kmem_cache_create(const char *name, size_t size, size_t align,
  cachep->orig_cache = NULL;
  atomic_set(&cachep->refcnt, 1);
```

```
  INIT_LIST_HEAD(&cachep->destroyed_list);
+ INIT_LIST_HEAD(&cachep->sibling_list);

  if (!memcg) {
   int id;
@@ -2580,6 +2581,7 @@ kmem_cache_create_memcg(struct kmem_cache *cachep, char
*name)
  new->flags |= SLAB_MEMCG;
  new->orig_cache = cachep;

+ list_add(&new->sibling_list, &cachep->sibling_list);
  if ((cachep->limit != new->limit) ||
      (cachep->batchcount != new->batchcount) ||
      (cachep->shared != new->shared))
@@ -2767,6 +2769,26 @@ void kmem_cache_destroy(struct kmem_cache *cachep)
 {
  BUG_ON(!cachep || in_interrupt());

+#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
+ /* Destroy all the children caches if we aren't a memcg cache */
+ if (cachep->id != -1) {
+  struct kmem_cache *c, *tmp;
+
+  mutex_lock(&cache_chain_mutex);
+  list_for_each_entry_safe(c, tmp, &cachep->sibling_list,
+      sibling_list) {
+   if (c == cachep)
+    continue;
+   mutex_unlock(&cache_chain_mutex);
+   BUG_ON(c->id != -1);
+   mem_cgroup_remove_child_kmem_cache(c, cachep->id);
+   kmem_cache_destroy(c);
+   mutex_lock(&cache_chain_mutex);
+  }
+  mutex_unlock(&cache_chain_mutex);
+ }
+#endif /* CONFIG_CGROUP_MEM_RES_CTLR_KMEM */
+
  /* Find the cache in the chain of caches. */
  get_online_cpus();
  mutex_lock(&cache_chain_mutex);
@@ -2774,6 +2796,9 @@ void kmem_cache_destroy(struct kmem_cache *cachep)
   * the chain is never empty, cache_cache is never destroyed
   */
  list_del(&cachep->next);
+#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
+ list_del(&cachep->sibling_list);
+#endif
```

```
  if (__cache_shrink(cachep)) {
   slab_error(cachep, "Can't free all objects");
   list_add(&cachep->next, &cache_chain);
@@ -4628,11 +4653,23 @@ static ssize_t slabinfo_write(struct file *file, const char __user
*buffer,
    if (limit < 1 || batchcount < 1 ||
      batchcount > limit || shared < 0) {
     res = 0;
-   } else {
-    res = do_tune_cpucache(cachep, limit,
-        batchcount, shared, GFP_KERNEL |
-        __GFP_NOACCOUNT);
+    break;
    }
+
+   res = do_tune_cpucache(cachep, limit, batchcount,
+       shared, GFP_KERNEL | __GFP_NOACCOUNT);
+
+#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
+   {
+    struct kmem_cache *c;
+
+    list_for_each_entry(c, &cachep->sibling_list,
+        sibling_list)
+     do_tune_cpucache(c, limit, batchcount,
+         shared, GFP_KERNEL |
+         __GFP_NOACCOUNT);
+   }
+#endif
    break;
   }
  }
--
1.7.7.3
```

Subject: [PATCH 07/10] memcg: Stop res_counter underflows.
Posted by Suleiman Souhlal on Mon, 27 Feb 2012 22:58:50 GMT
View Forum Message <> Reply to Message

From: Hugh Dickins <hughd@google.com>

If __mem_cgroup_try_charge() goes the "bypass" route in charging slab
(typically when the task has been OOM-killed), that later results in
res_counter_uncharge_locked() underflows - a stream of warnings from
kernel/res_counter.c:96!

Solve this by accounting kmem_bypass when we shift that charge to root,

and whenever a memcg has any kmem_bypass outstanding, deduct from that when unaccounting kmem, before deducting from kmem_bytes: so that its kmem_bytes soon returns to being a fair account.

The amount of memory bypassed is shown in memory.stat as kernel_bypassed_memory.

Signed-off-by: Hugh Dickins <hughd@google.com>
Signed-off-by: Suleiman Souhlal <suleiman@google.com>
---
 mm/memcontrol.c | 43 ++---
 1 files changed, 40 insertions(+), 3 deletions(-)

```
diff --git a/mm/memcontrol.c b/mm/memcontrol.c
index d1c0cd7..6a475ed 100644
--- a/mm/memcontrol.c
+++ b/mm/memcontrol.c
@@ -302,6 +302,9 @@ struct mem_cgroup {
  /* Slab accounting */
  struct kmem_cache *slabs[MAX_KMEM_CACHE_TYPES];
 #endif
+#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
+ atomic64_t kmem_bypassed;
+#endif
  int independent_kmem_limit;
 };

@@ -4037,6 +4040,7 @@ enum {
  MCS_INACTIVE_FILE,
  MCS_ACTIVE_FILE,
  MCS_UNEVICTABLE,
+ MCS_KMEM_BYPASSED,
  NR_MCS_STAT,
 };

@@ -4060,7 +4064,8 @@ struct {
  {"active_anon", "total_active_anon"},
  {"inactive_file", "total_inactive_file"},
  {"active_file", "total_active_file"},
- {"unevictable", "total_unevictable"}
+ {"unevictable", "total_unevictable"},
+ {"kernel_bypassed_memory", "total_kernel_bypassed_memory"}
 };


@@ -4100,6 +4105,10 @@ mem_cgroup_get_local_stat(struct mem_cgroup *memcg, struct
mcs_total_stat *s)
  s->stat[MCS_ACTIVE_FILE] += val * PAGE_SIZE;
```

```
  val = mem_cgroup_nr_lru_pages(memcg, BIT(LRU_UNEVICTABLE));
  s->stat[MCS_UNEVICTABLE] += val * PAGE_SIZE;
+
+#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
+ s->stat[MCS_KMEM_BYPASSED] += atomic64_read(&memcg->kmem_bypassed);
+#endif
 }

 static void
@@ -5616,14 +5625,24 @@ memcg_charge_kmem(struct mem_cgroup *memcg, gfp_t gfp, long
long delta)
  ret = 0;

  if (memcg && !memcg->independent_kmem_limit) {
+  /*
+   * __mem_cgroup_try_charge may decide to bypass the charge and
+   * set _memcg to NULL, in which case we need to account to the
+   * root.
+   */
   _memcg = memcg;
   if (__mem_cgroup_try_charge(NULL, gfp, delta / PAGE_SIZE,
       &_memcg, may_oom) != 0)
    return -ENOMEM;
+
+  if (!_memcg && memcg != root_mem_cgroup) {
+   atomic64_add(delta, &memcg->kmem_bypassed);
+   memcg = NULL;
+  }
  }

- if (_memcg)
-  ret = res_counter_charge(&_memcg->kmem_bytes, delta, &fail_res);
+ if (memcg)
+  ret = res_counter_charge(&memcg->kmem_bytes, delta, &fail_res);

  return ret;
 }
@@ -5631,6 +5650,22 @@ memcg_charge_kmem(struct mem_cgroup *memcg, gfp_t gfp, long
long delta)
 void
 memcg_uncharge_kmem(struct mem_cgroup *memcg, long long delta)
 {
+ long long bypassed;
+
+ if (memcg) {
+  bypassed = atomic64_read(&memcg->kmem_bypassed);
+  if (bypassed > 0) {
+   if (bypassed > delta)
```

```
+    bypassed = delta;
+   do {
+    memcg_uncharge_kmem(NULL, bypassed);
+    delta -= bypassed;
+    bypassed = atomic64_sub_return(bypassed,
+        &memcg->kmem_bypassed);
+   } while (bypassed < 0); /* Might have raced */
+  }
+ }
+
  if (memcg)
   res_counter_uncharge(&memcg->kmem_bytes, delta);

@@ -5956,6 +5991,7 @@ memcg_kmem_init(struct mem_cgroup *memcg, struct mem_cgroup
*parent)

  memcg_slab_init(memcg);

+ atomic64_set(&memcg->kmem_bypassed, 0);
  memcg->independent_kmem_limit = 0;
 }

@@ -5967,6 +6003,7 @@ memcg_kmem_move(struct mem_cgroup *memcg)

  memcg_slab_move(memcg);

+ atomic64_set(&memcg->kmem_bypassed, 0);
  spin_lock_irqsave(&memcg->kmem_bytes.lock, flags);
  kmem_bytes = memcg->kmem_bytes.usage;
  res_counter_uncharge_locked(&memcg->kmem_bytes, kmem_bytes);
--
1.7.7.3
```

Subject: [PATCH 08/10] memcg: Add CONFIG_CGROUP_MEM_RES_CTLR_KMEM_ACCT_ROOT.
Posted by Suleiman Souhlal on Mon, 27 Feb 2012 22:58:51 GMT

View Forum Message <> Reply to Message

This config option dictates whether or not kernel memory in the
root cgroup should be accounted.

This may be useful in an environment where everything is supposed to be
in a cgroup and accounted for. Large amounts of kernel memory in the
root cgroup would indicate problems with memory isolation or accounting.

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

```
 init/Kconfig    |    8 ++++++++
 mm/memcontrol.c |   44 +++++++++++++++++++++++++++++++++++++++++++++
 2 files changed, 52 insertions(+), 0 deletions(-)

diff --git a/init/Kconfig b/init/Kconfig
index 3f42cd6..a119270 100644
--- a/init/Kconfig
+++ b/init/Kconfig
@@ -714,6 +714,14 @@ config CGROUP_MEM_RES_CTLR_KMEM
    Memory Controller, which are page-based, and can be swapped. Users of
    the kmem extension can use it to guarantee that no group of processes
    will ever exhaust kernel resources alone.
+config CGROUP_MEM_RES_CTLR_KMEM_ACCT_ROOT
+ bool "Root Cgroup Kernel Memory Accounting (EXPERIMENTAL)"
+ depends on CGROUP_MEM_RES_CTLR_KMEM
+ default n
+ help
+   Account for kernel memory used by the root cgroup. This may be useful
+   to know how much kernel memory isn't currently accounted to any
+   cgroup.

 config CGROUP_PERF
  bool "Enable perf_event per-cpu per-container group (cgroup) monitoring"
diff --git a/mm/memcontrol.c b/mm/memcontrol.c
index 6a475ed..d4cdb8e 100644
--- a/mm/memcontrol.c
+++ b/mm/memcontrol.c
@@ -61,6 +61,10 @@ struct cgroup_subsys mem_cgroup_subsys __read_mostly;
 #define MEM_CGROUP_RECLAIM_RETRIES 5
 struct mem_cgroup *root_mem_cgroup __read_mostly;

+#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM_ACCT_ROOT
+atomic64_t pre_memcg_kmem_bytes; /* kmem usage before memcg is enabled */
+#endif
+
 #ifdef CONFIG_CGROUP_MEM_RES_CTLR_SWAP
 /* Turned on only when memory cgroup is enabled && really_do_swap_account = 1 */
 int do_swap_account __read_mostly;
@@ -5643,6 +5647,13 @@ memcg_charge_kmem(struct mem_cgroup *memcg, gfp_t gfp, long
long delta)

  if (memcg)
   ret = res_counter_charge(&memcg->kmem_bytes, delta, &fail_res);
+#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM_ACCT_ROOT
+ else if (root_mem_cgroup != NULL)
+  ret = res_counter_charge(&root_mem_cgroup->kmem_bytes, delta,
+      &fail_res);
+ else
```

```
+  atomic64_add(delta, &pre_memcg_kmem_bytes);
+#endif

  return ret;
 }
@@ -5668,6 +5679,12 @@ memcg_uncharge_kmem(struct mem_cgroup *memcg, long long
delta)

  if (memcg)
   res_counter_uncharge(&memcg->kmem_bytes, delta);
+#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM_ACCT_ROOT
+ else if (root_mem_cgroup != NULL)
+  res_counter_uncharge(&root_mem_cgroup->kmem_bytes, delta);
+ else
+  atomic64_sub(delta, &pre_memcg_kmem_bytes);
+#endif

  if (memcg && !memcg->independent_kmem_limit)
   res_counter_uncharge(&memcg->res, delta);
@@ -5953,7 +5970,12 @@ memcg_slab_move(struct mem_cgroup *memcg)
   cachep = rcu_access_pointer(memcg->slabs[i]);
   if (cachep != NULL) {
    rcu_assign_pointer(memcg->slabs[i], NULL);
+
+#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM_ACCT_ROOT
+   cachep->memcg = root_mem_cgroup;
+#else
    cachep->memcg = NULL;
+#endif

    /* The space for this is already allocated */
    strcat((char *)cachep->name, "dead");
@@ -5991,6 +6013,15 @@ memcg_kmem_init(struct mem_cgroup *memcg, struct mem_cgroup
*parent)

  memcg_slab_init(memcg);

+#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM_ACCT_ROOT
+ if (memcg == root_mem_cgroup) {
+  long kmem_bytes;
+
+  kmem_bytes = atomic64_xchg(&pre_memcg_kmem_bytes, 0);
+  memcg->kmem_bytes.usage = kmem_bytes;
+ }
+#endif
+
  atomic64_set(&memcg->kmem_bypassed, 0);
  memcg->independent_kmem_limit = 0;
```

```
 }
@@ -6010,6 +6041,19 @@ memcg_kmem_move(struct mem_cgroup *memcg)
  spin_unlock_irqrestore(&memcg->kmem_bytes.lock, flags);
  if (!memcg->independent_kmem_limit)
   res_counter_uncharge(&memcg->res, kmem_bytes);
+
+#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM_ACCT_ROOT
+ {
+  struct res_counter *dummy;
+  int err;
+
+  /* Can't fail because it's the root cgroup */
+  err = res_counter_charge(&root_mem_cgroup->kmem_bytes,
+       kmem_bytes, &dummy);
+  err = res_counter_charge(&root_mem_cgroup->res, kmem_bytes,
+       &dummy);
+ }
+#endif
 }
 #else /* CONFIG_CGROUP_MEM_RES_CTLR_KMEM */
 static void
--
1.7.7.3
```

Subject: [PATCH 09/10] memcg: Per-memcg memory.kmem.slabinfo file.
Posted by Suleiman Souhlal on Mon, 27 Feb 2012 22:58:52 GMT

View Forum Message <> Reply to Message

This file shows all the kmem_caches used by a memcg.

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

```
---
 include/linux/slab_def.h |    7 ++++
 mm/memcontrol.c          |   22 +++++++++++
 mm/slab.c                |   88 +++++++++++++++++++++++++++++++++++++++++------------
 3 files changed, 94 insertions(+), 23 deletions(-)

diff --git a/include/linux/slab_def.h b/include/linux/slab_def.h
index 185e4a2..407eaf5 100644
--- a/include/linux/slab_def.h
+++ b/include/linux/slab_def.h
@@ -242,6 +242,7 @@ struct kmem_cache *kmem_cache_create_memcg(struct kmem_cache
*cachep,
     char *name);
 void kmem_cache_destroy_cpuset(struct kmem_cache *cachep);
 void kmem_cache_drop_ref(struct kmem_cache *cachep);
+int mem_cgroup_slabinfo(struct mem_cgroup *mem, struct seq_file *m);
```

```
 static inline void
 kmem_cache_get_ref(struct kmem_cache *cachep)
@@ -301,6 +302,12 @@ static inline void
 mem_cgroup_kmem_cache_finish_sleep(struct kmem_cache *cachep)
 {
 }
+
+static inline int
+memcg_slabinfo(void *unused, struct seq_file *m)
+{
+ return 0;
+}
 #endif /* CONFIG_CGROUP_MEM_RES_CTLR_KMEM */

 #endif /* _LINUX_SLAB_DEF_H */
diff --git a/mm/memcontrol.c b/mm/memcontrol.c
index d4cdb8e..d4a6053 100644
--- a/mm/memcontrol.c
+++ b/mm/memcontrol.c
@@ -4637,6 +4637,22 @@ static int mem_cgroup_independent_kmem_limit_write(struct cgroup
*cgrp,
  return 0;
 }

+#ifdef CONFIG_SLAB
+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);
+}
+#endif
+
 static struct cftype kmem_cgroup_files[] = {
  {
   .name = "kmem.independent_kmem_limit",
@@ -4654,6 +4670,12 @@ static struct cftype kmem_cgroup_files[] = {
   .private = MEMFILE_PRIVATE(_KMEM, RES_USAGE),
   .read_u64 = mem_cgroup_read,
  },
```

```
+#ifdef CONFIG_SLAB
+ {
+  .name = "kmem.slabinfo",
+  .read_seq_string = mem_cgroup_slabinfo_show,
+ },
+#endif
 };

 static int register_kmem_files(struct cgroup *cont, struct cgroup_subsys *ss)
diff --git a/mm/slab.c b/mm/slab.c
index bf38af6..6d9f069 100644
--- a/mm/slab.c
+++ b/mm/slab.c
@@ -4498,21 +4498,26 @@ 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)
@@ -4524,31 +4529,43 @@ 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;
@@ -4556,12 +4573,12 @@ 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;
@@ -4692,6 +4709,31 @@ static const struct file_operations proc_slabinfo_operations = {
  .release = seq_release,
 };

+#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
+int
+mem_cgroup_slabinfo(struct mem_cgroup *mem, 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 != mem)
+   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 */
+
 #ifdef CONFIG_DEBUG_SLAB_LEAK

 static void *leaks_start(struct seq_file *m, loff_t *pos)
--
1.7.7.3
```

Subject: [PATCH 10/10] memcg: Document kernel memory accounting.
Posted by Suleiman Souhlal on Mon, 27 Feb 2012 22:58:53 GMT
View Forum Message <> Reply to Message

```
Signed-off-by: Suleiman Souhlal <suleiman@google.com>
---
 Documentation/cgroups/memory.txt |   44 ++++++++++++++++++++++++++++++++++++--
 1 files changed, 41 insertions(+), 3 deletions(-)

diff --git a/Documentation/cgroups/memory.txt b/Documentation/cgroups/memory.txt
index 4c95c00..64c6cc8 100644
--- a/Documentation/cgroups/memory.txt
+++ b/Documentation/cgroups/memory.txt
@@ -74,6 +74,11 @@ 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.usage_in_bytes  # show current kernel memory usage
+ memory.kmem.limit_in_bytes  # show/set limit of kernel memory usage
+ memory.kmem.independent_kmem_limit # show/set control of kernel memory limit
+        (See 2.7 for details)
+ memory.kmem.slabinfo   # show cgroup's slabinfo

 1. History

@@ -266,10 +271,20 @@ different than user memory, since it can't be swapped out, which
makes it
 possible to DoS the system by consuming too much of this precious resource.

 Kernel memory limits are not imposed for the root cgroup. Usage for the root
-cgroup may or may not be accounted.
+cgroup is accounted if CONFIG_CGROUP_MEM_RES_CTLR_KMEM_ACCT_ROOT is set.

-Currently no soft limit is implemented for kernel memory. It is future work
-to trigger slab reclaim when those limits are reached.
+A cgroup's kernel memory is counted into its memory.kmem.usage_in_bytes.
```

+
+memory.kmem.independent_kmem_limit controls whether or not kernel memory
+should also be counted into the cgroup's memory.usage_in_bytes.
+If it is set, it is possible to specify a limit for kernel memory with
+memory.kmem.limit_in_bytes.
+
+Upon cgroup deletion, all the remaining kernel memory gets moved to the
+root cgroup (if CONFIG_CGROUP_MEM_RES_CTLR_KMEM_ACCT_ROOT is set).
+
+An accounted kernel memory allocation may trigger reclaim in that cgroup,
+and may also OOM.

2.7.1 Current Kernel Memory resources accounted

@@ -279,6 +294,26 @@ per cgroup, instead of globally.

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

+* slab memory.
+
+2.7.1.1 Slab memory accounting
+
+Slab gets accounted on a per-page basis, which is done by using per-cgroup
+kmem_caches. These per-cgroup kmem_caches get created on-demand, the first
+time a specific kmem_cache gets used by a cgroup.
+
+Slab memory that cannot be attributed to a cgroup gets charged to the root
+cgroup.
+
+A per-cgroup kmem_cache is named like the original, with the cgroup's name
+in parethesis.
+
+When a cgroup is destroyed, all its kmem_caches get migrated to the root
+cgroup, and "dead" is appended to their name, to indicate that they are not
+going to be used for new allocations.
+These dead caches automatically get removed once there are no more active
+slab objects in them.
+
3. User Interface

0. Configuration
@@ -423,6 +458,8 @@ active_anon - # of bytes of anonymous and swap cache memory on active
inactive_file - # of bytes of file-backed memory on inactive LRU list.
active_file - # of bytes of file-backed memory on active LRU list.
unevictable - # of bytes of memory that cannot be reclaimed (mlocked etc).
+kernel_bypassed_memory - # of bytes of kernel memory that should have been
+ accounted, but got bypassed to the root cgroup.

```
 # status considering hierarchy (see memory.use_hierarchy settings)

@@ -442,6 +479,7 @@ total_active_anon - sum of all children's "active_anon"
 total_inactive_file - sum of all children's "inactive_file"
 total_active_file - sum of all children's "active_file"
 total_unevictable - sum of all children's "unevictable"
+total_kernel_bypassed_memory - sum of all children's "kernel_bypassed_memory"

 # The following additional stats are dependent on CONFIG_DEBUG_VM.

--
1.7.7.3
```

Subject: Re: [PATCH 00/10] memcg: Kernel Memory Accounting.
Posted by Pekka Enberg on Tue, 28 Feb 2012 08:49:46 GMT
View Forum Message <> Reply to Message

On Mon, 27 Feb 2012, Suleiman Souhlal wrote:
> The main difference with Glauber's patches is here: We try to
> track all the slab allocations, while Glauber only tracks ones
> that are explicitly marked.
> We feel that it's important to track everything, because there
> are a lot of different slab allocations that may use significant
> amounts of memory, that we may not know of ahead of time.
> This is also the main source of complexity in the patchset.

Well, what are the performance implications of your patches? Can we
reasonably expect distributions to be able to enable this thing on
generic kernels and leave the feature disabled by default? Can we
accommodate your patches to support Glauber's use case?

Pekka

Subject: Re: [PATCH 00/10] memcg: Kernel Memory Accounting.
Posted by Suleiman Souhlal on Tue, 28 Feb 2012 22:12:47 GMT
View Forum Message <> Reply to Message

Hello,

On Tue, Feb 28, 2012 at 12:49 AM, Pekka Enberg <penberg@kernel.org> wrote:
> On Mon, 27 Feb 2012, Suleiman Souhlal wrote:
>> The main difference with Glauber's patches is here: We try to
>> track all the slab allocations, while Glauber only tracks ones
>> that are explicitly marked.

>> We feel that it's important to track everything, because there
>> are a lot of different slab allocations that may use significant
>> amounts of memory, that we may not know of ahead of time.
>> This is also the main source of complexity in the patchset.
>
> Well, what are the performance implications of your patches? Can we
> reasonably expect distributions to be able to enable this thing on
> generic kernels and leave the feature disabled by default? Can we
> accommodate your patches to support Glauber's use case?

I don't have up to date performance numbers, but we haven't found any critical performance degradations on our workloads, with our internal versions of this patchset.

There are some conditional branches added to the slab fast paths, but I think it should be possible to come with a way to get rid of those when the feature is disabled, maybe by using a static_branch. This should hopefully make it possible to keep the feature compiled in but disabled at runtime.

I think it's definitely possible to accommodate my patches to support Glauber's use case, with a bit of work.

-- Suleiman