Subject: [PATCH v2 00/13] Memcg Kernel Memory Tracking.
Posted by Suleiman Souhlal on Fri, 09 Mar 2012 20:39:03 GMT
View Forum Message <> Reply to Message

This is v2 of my kernel memory tracking patchset for memcg.

Lots of changes based on feedback from Glauber and Kamezawa.
In particular, I changed it to be opt-in instead of opt-out:
In order for a slab type to be tracked, it has to be marked with
SLAB_MEMCG_ACCT at kmem_cache_create() time.
Currently, only dentries and kmalloc are tracked.

Planned for v3:
- Slub support.
- Using a static_branch to remove overhead when no cgroups have been
  created.
- Getting rid of kmem_cache_get_ref/drop_ref pair in kmem_cache_free.

Detailed change list from v1 (http://marc.info/?l=linux-mm&m=133038361014525):
- Fixed misspelling in documentation.
- Added flags field to struct mem_cgroup.
- Moved independent_kmem_limit into flags.
- Renamed kmem_bytes to kmem.
- Divided consume_stock changes into two changes.
- Fixed crash at boot when not every commit is applied.
- Moved the new fields in kmem_cache into their own struct.
- Got rid of SLAB_MEMCG slab flag.
- Dropped accounting to root.
- Added css_id into memcg slab name.
- Changed memcg cache creation to always be deferred to workqueue.
- Replaced bypass_bytes with overcharging the cgroup.
- Got rid of #ifdef CONFIG_SLAB from memcontrol.c.
- Got rid of __GFP_NOACCOUNT, changing to an opt-in model.
- Remove kmem limit when turning off independent limit.
- Moved the accounting of kmalloc to its own patch.
- Removed useless parameters from memcg_create_kmem_cache().
- Get a ref to the css when enqueing cache for creation.
- increased MAX_KMEM_CACHE_TYPES to 400.

Suleiman Souhlal (13):
  memcg: Consolidate various flags into a single flags field.
  memcg: Kernel memory accounting infrastructure.
  memcg: Uncharge all kmem when deleting a cgroup.
  memcg: Make it possible to use the stock for more than one page.
  memcg: Reclaim when more than one page needed.
  slab: Add kmem_cache_gfp_flags() helper function.
  memcg: Slab accounting.
  memcg: Make dentry slab memory accounted in kernel memory accounting.

memcg: Account for kmalloc in kernel memory accounting.
memcg: Track all the memcg children of a kmem_cache.
memcg: Handle bypassed kernel memory charges.
memcg: Per-memcg memory.kmem.slabinfo file.
memcg: Document kernel memory accounting.

Documentation/cgroups/memory.txt | 44 +++-
fs/dcache.c | 4 +-
include/linux/memcontrol.h | 30 ++-
include/linux/slab.h | 56 ++++
include/linux/slab_def.h | 79 +++++-
include/linux/slob_def.h | 6 +
include/linux/slub_def.h | 9 +
init/Kconfig | 2 +-
mm/memcontrol.c | 633 ++---
mm/slab.c | 431 ++++++++++++++++++++++++++++---
10 files changed, 1183 insertions(+), 111 deletions(-)

-- Suleiman

Subject: [PATCH v2 01/13] memcg: Consolidate various flags into a single flags field.
Posted by Suleiman Souhlal on Fri, 09 Mar 2012 20:39:04 GMT
View Forum Message <> Reply to Message

Since there is an ever-increasing number of flags in the memcg
struct, consolidate them into a single flags field.
The flags that we consolidate are:
- use_hierarchy
- memsw_is_minimum
- oom_kill_disable

Signed-off-by: Suleiman Souhlal <suleiman@google.com>
---
mm/memcontrol.c | 112 ++----------
1 files changed, 78 insertions(+), 34 deletions(-)

diff --git a/mm/memcontrol.c b/mm/memcontrol.c
index 5585dc3..37ad2cb 100644
--- a/mm/memcontrol.c
+++ b/mm/memcontrol.c
@@ -213,6 +213,15 @@ struct mem_cgroup_eventfd_list {
static void mem_cgroup_threshold(struct mem_cgroup *memcg);
static void mem_cgroup_oom_notify(struct mem_cgroup *memcg);

+enum memcg_flags {
+ MEMCG_USE_HIERARCHY, /*

```
+     * Should the accounting and control be
+     * hierarchical, per subtree?
+     */
+ MEMCG_MEMSW_IS_MINIMUM, /* Set when res.limit == memsw.limit */
+ MEMCG_OOM_KILL_DISABLE, /* OOM-Killer disable */
+};
+
 /*
  * The memory controller data structure. The memory controller controls both
  * page cache and RSS per cgroup. We would eventually like to provide
@@ -245,10 +254,7 @@ struct mem_cgroup {
  atomic_t numainfo_events;
  atomic_t numainfo_updating;
 #endif
- /*
-  * Should the accounting and control be hierarchical, per subtree?
-  */
- bool use_hierarchy;
+ unsigned long flags;

  bool  oom_lock;
  atomic_t under_oom;
@@ -256,11 +262,6 @@ struct mem_cgroup {
  atomic_t refcnt;

  int swappiness;
- /* OOM-Killer disable */
- int  oom_kill_disable;
-
- /* set when res.limit == memsw.limit */
- bool  memsw_is_minimum;

  /* protect arrays of thresholds */
  struct mutex thresholds_lock;
@@ -371,6 +372,24 @@ enum charge_type {
 static void mem_cgroup_get(struct mem_cgroup *memcg);
 static void mem_cgroup_put(struct mem_cgroup *memcg);

+static inline bool
+mem_cgroup_test_flag(const struct mem_cgroup *memcg, enum memcg_flags flag)
+{
+ return test_bit(flag, &memcg->flags);
+}
+
+static inline void
+mem_cgroup_set_flag(struct mem_cgroup *memcg, enum memcg_flags flag)
+{
+ set_bit(flag, &memcg->flags);
```

```
+}
+
+static inline void
+mem_cgroup_clear_flag(struct mem_cgroup *memcg, enum memcg_flags flag)
+{
+ clear_bit(flag, &memcg->flags);
+}
+
 /* Writing them here to avoid exposing memcg's inner layout */
 #ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
 #include <net/sock.h>
@@ -876,7 +895,8 @@ struct mem_cgroup *mem_cgroup_iter(struct mem_cgroup *root,
  if (prev && prev != root)
   css_put(&prev->css);

- if (!root->use_hierarchy && root != root_mem_cgroup) {
+ if (!mem_cgroup_test_flag(root, MEMCG_USE_HIERARCHY) && root !=
+     root_mem_cgroup) {
   if (prev)
    return NULL;
   return root;
@@ -1126,8 +1146,8 @@ static bool mem_cgroup_same_or_subtree(const struct mem_cgroup
*root_memcg,
   struct mem_cgroup *memcg)
 {
  if (root_memcg != memcg) {
-  return (root_memcg->use_hierarchy &&
-   css_is_ancestor(&memcg->css, &root_memcg->css));
+  return mem_cgroup_test_flag(root_memcg, MEMCG_USE_HIERARCHY) &&
+   css_is_ancestor(&memcg->css, &root_memcg->css);
  }

  return true;
@@ -1460,7 +1480,8 @@ static unsigned long mem_cgroup_reclaim(struct mem_cgroup
*memcg,

  if (flags & MEM_CGROUP_RECLAIM_NOSWAP)
   noswap = true;
- if (!(flags & MEM_CGROUP_RECLAIM_SHRINK) && memcg->memsw_is_minimum)
+ if (!(flags & MEM_CGROUP_RECLAIM_SHRINK) && mem_cgroup_test_flag(memcg,
+     MEMCG_MEMSW_IS_MINIMUM))
   noswap = true;

  for (loop = 0; loop < MEM_CGROUP_MAX_RECLAIM_LOOPS; loop++) {
@@ -1813,7 +1834,7 @@ bool mem_cgroup_handle_oom(struct mem_cgroup *memcg, gfp_t
mask)
   * under OOM is always welcomed, use TASK_KILLABLE here.
   */
```

```
  prepare_to_wait(&memcg_oom_waitq, &owait.wait, TASK_KILLABLE);
- if (!locked || memcg->oom_kill_disable)
+ if (!locked || mem_cgroup_test_flag(memcg, MEMCG_OOM_KILL_DISABLE))
   need_to_kill = false;
  if (locked)
   mem_cgroup_oom_notify(memcg);
@@ -3416,9 +3437,11 @@ static int mem_cgroup_resize_limit(struct mem_cgroup *memcg,
   ret = res_counter_set_limit(&memcg->res, val);
   if (!ret) {
    if (memswlimit == val)
-    memcg->memsw_is_minimum = true;
+    mem_cgroup_set_flag(memcg,
+        MEMCG_MEMSW_IS_MINIMUM);
    else
-    memcg->memsw_is_minimum = false;
+    mem_cgroup_clear_flag(memcg,
+        MEMCG_MEMSW_IS_MINIMUM);
   }
   mutex_unlock(&set_limit_mutex);

@@ -3475,9 +3498,11 @@ static int mem_cgroup_resize_memsw_limit(struct mem_cgroup
*memcg,
   ret = res_counter_set_limit(&memcg->memsw, val);
   if (!ret) {
    if (memlimit == val)
-    memcg->memsw_is_minimum = true;
+    mem_cgroup_set_flag(memcg,
+        MEMCG_MEMSW_IS_MINIMUM);
    else
-    memcg->memsw_is_minimum = false;
+    mem_cgroup_clear_flag(memcg,
+        MEMCG_MEMSW_IS_MINIMUM);
   }
   mutex_unlock(&set_limit_mutex);

@@ -3745,7 +3770,8 @@ int mem_cgroup_force_empty_write(struct cgroup *cont, unsigned int
event)

 static u64 mem_cgroup_hierarchy_read(struct cgroup *cont, struct cftype *cft)
 {
- return mem_cgroup_from_cont(cont)->use_hierarchy;
+ return mem_cgroup_test_flag(mem_cgroup_from_cont(cont),
+     MEMCG_USE_HIERARCHY);
 }

 static int mem_cgroup_hierarchy_write(struct cgroup *cont, struct cftype *cft,
@@ -3768,10 +3794,14 @@ static int mem_cgroup_hierarchy_write(struct cgroup *cont, struct
cftype *cft,
```

```
   * For the root cgroup, parent_mem is NULL, we allow value to be
   * set if there are no children.
   */
- if ((!parent_memcg || !parent_memcg->use_hierarchy) &&
-    (val == 1 || val == 0)) {
+ if ((!parent_memcg || !mem_cgroup_test_flag(parent_memcg,
+     MEMCG_USE_HIERARCHY)) && (val == 1 || val == 0)) {
   if (list_empty(&cont->children))
-   memcg->use_hierarchy = val;
+   if (val)
+    mem_cgroup_set_flag(memcg, MEMCG_USE_HIERARCHY);
+   else
+    mem_cgroup_clear_flag(memcg,
+        MEMCG_USE_HIERARCHY);
   else
    retval = -EBUSY;
  } else
@@ -3903,13 +3933,13 @@ static void memcg_get_hierarchical_limit(struct mem_cgroup
*memcg,
  min_limit = res_counter_read_u64(&memcg->res, RES_LIMIT);
  min_memsw_limit = res_counter_read_u64(&memcg->memsw, RES_LIMIT);
  cgroup = memcg->css.cgroup;
- if (!memcg->use_hierarchy)
+ if (!mem_cgroup_test_flag(memcg, MEMCG_USE_HIERARCHY))
   goto out;

  while (cgroup->parent) {
   cgroup = cgroup->parent;
   memcg = mem_cgroup_from_cont(cgroup);
-  if (!memcg->use_hierarchy)
+  if (!mem_cgroup_test_flag(memcg, MEMCG_USE_HIERARCHY))
    break;
   tmp = res_counter_read_u64(&memcg->res, RES_LIMIT);
   min_limit = min(min_limit, tmp);
@@ -4206,8 +4236,9 @@ static int mem_cgroup_swappiness_write(struct cgroup *cgrp, struct
cftype *cft,
  cgroup_lock();

  /* If under hierarchy, only empty-root can set this value */
- if ((parent->use_hierarchy) ||
-     (memcg->use_hierarchy && !list_empty(&cgrp->children))) {
+ if (mem_cgroup_test_flag(parent, MEMCG_USE_HIERARCHY) ||
+     (mem_cgroup_test_flag(memcg, MEMCG_USE_HIERARCHY) &&
+     !list_empty(&cgrp->children))) {
   cgroup_unlock();
   return -EINVAL;
  }
@@ -4518,7 +4549,8 @@ static int mem_cgroup_oom_control_read(struct cgroup *cgrp,
```

```
 {
  struct mem_cgroup *memcg = mem_cgroup_from_cont(cgrp);

- cb->fill(cb, "oom_kill_disable", memcg->oom_kill_disable);
+ cb->fill(cb, "oom_kill_disable", mem_cgroup_test_flag(memcg,
+     MEMCG_OOM_KILL_DISABLE));

  if (atomic_read(&memcg->under_oom))
   cb->fill(cb, "under_oom", 1);
@@ -4541,14 +4573,18 @@ static int mem_cgroup_oom_control_write(struct cgroup *cgrp,

  cgroup_lock();
  /* oom-kill-disable is a flag for subhierarchy. */
- if ((parent->use_hierarchy) ||
-     (memcg->use_hierarchy && !list_empty(&cgrp->children))) {
+ if (mem_cgroup_test_flag(parent, MEMCG_USE_HIERARCHY) ||
+     (mem_cgroup_test_flag(memcg, MEMCG_USE_HIERARCHY) &&
+     !list_empty(&cgrp->children))) {
   cgroup_unlock();
   return -EINVAL;
  }
- memcg->oom_kill_disable = val;
- if (!val)
+ if (val)
+  mem_cgroup_set_flag(memcg, MEMCG_OOM_KILL_DISABLE);
+ else {
+  mem_cgroup_clear_flag(memcg, MEMCG_OOM_KILL_DISABLE);
   memcg_oom_recover(memcg);
+ }
  cgroup_unlock();
  return 0;
 }
@@ -4916,11 +4952,19 @@ mem_cgroup_create(struct cgroup_subsys *ss, struct cgroup *cont)
   hotcpu_notifier(memcg_cpu_hotplug_callback, 0);
  } else {
   parent = mem_cgroup_from_cont(cont->parent);
-  memcg->use_hierarchy = parent->use_hierarchy;
-  memcg->oom_kill_disable = parent->oom_kill_disable;
+
+  if (mem_cgroup_test_flag(parent, MEMCG_USE_HIERARCHY))
+   mem_cgroup_set_flag(memcg, MEMCG_USE_HIERARCHY);
+  else
+   mem_cgroup_clear_flag(memcg, MEMCG_USE_HIERARCHY);
+
+  if (mem_cgroup_test_flag(parent, MEMCG_OOM_KILL_DISABLE))
+   mem_cgroup_set_flag(memcg, MEMCG_OOM_KILL_DISABLE);
+  else
+   mem_cgroup_clear_flag(memcg, MEMCG_OOM_KILL_DISABLE);
```

```
 }

- if (parent && parent->use_hierarchy) {
+ if (parent && mem_cgroup_test_flag(parent, MEMCG_USE_HIERARCHY)) {
   res_counter_init(&memcg->res, &parent->res);
   res_counter_init(&memcg->memsw, &parent->memsw);
   /*
--
1.7.7.3
```

Subject: [PATCH v2 02/13] memcg: Kernel memory accounting infrastructure.
Posted by Suleiman Souhlal on Fri, 09 Mar 2012 20:39:05 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 |  136 ++++++++++++++++++++++++++++++++++++++++++++++++++++++-
 1 files changed, 135 insertions(+), 1 deletions(-)

diff --git a/mm/memcontrol.c b/mm/memcontrol.c
index 37ad2cb..e6fd558 100644
--- a/mm/memcontrol.c
+++ b/mm/memcontrol.c
@@ -220,6 +220,10 @@ enum memcg_flags {
      */
   MEMCG_MEMSW_IS_MINIMUM, /* Set when res.limit == memsw.limit */
   MEMCG_OOM_KILL_DISABLE, /* OOM-Killer disable */
+ MEMCG_INDEPENDENT_KMEM_LIMIT, /*
+      * kernel memory is not counted in
+      * memory.usage_in_bytes
+      */
 };

 /*
@@ -244,6 +248,10 @@ struct mem_cgroup {
    */
   struct res_counter memsw;
   /*
+  * 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.
   */
@@ -355,6 +363,7 @@ 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)
@@ -371,6 +380,8 @@ 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);

 static inline bool
 mem_cgroup_test_flag(const struct mem_cgroup *memcg, enum memcg_flags flag)
@@ -1435,6 +1446,10 @@ 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));
 }

 /*
@@ -3868,6 +3883,9 @@ 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, name);
+  break;
  default:
   BUG();
   break;
@@ -3900,8 +3918,15 @@ 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)
```

```
+  else if (type == _MEMSWAP)
    ret = mem_cgroup_resize_memsw_limit(memcg, val);
+  else if (type == _KMEM) {
+   if (!mem_cgroup_test_flag(memcg,
+       MEMCG_INDEPENDENT_KMEM_LIMIT))
+    return -EINVAL;
+   ret = res_counter_set_limit(&memcg->kmem, val);
+  } else
+   return -EINVAL;
   break;
  case RES_SOFT_LIMIT:
   ret = res_counter_memparse_write_strategy(buffer, &val);
@@ -4606,8 +4631,56 @@ 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_test_flag(mem_cgroup_from_cont(cgrp),
+     MEMCG_INDEPENDENT_KMEM_LIMIT);
+}
+
+static int mem_cgroup_independent_kmem_limit_write(struct cgroup *cgrp,
+    struct cftype *cft, u64 val)
+{
+ struct mem_cgroup *memcg;
+
+ memcg = mem_cgroup_from_cont(cgrp);
+ if (val)
+  mem_cgroup_set_flag(memcg, MEMCG_INDEPENDENT_KMEM_LIMIT);
+ else {
+  mem_cgroup_clear_flag(memcg, MEMCG_INDEPENDENT_KMEM_LIMIT);
+  res_counter_set_limit(&memcg->kmem, RESOURCE_MAX);
+ }
+
+ 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.
@@ -4621,6 +4694,10 @@ 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, RES_USAGE) != 0);
  mem_cgroup_sockets_destroy(cont, ss);
 }
 #else
@@ -4980,6 +5057,8 @@ 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 && mem_cgroup_test_flag(parent,
+      MEMCG_USE_HIERARCHY) ? parent : NULL);

  if (parent)
   memcg->swappiness = mem_cgroup_swappiness(parent);
@@ -5561,3 +5640,58 @@ 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;
+
+ _memcg = memcg;
+ if (memcg && !mem_cgroup_test_flag(memcg,
+     MEMCG_INDEPENDENT_KMEM_LIMIT)) {
+  ret = __mem_cgroup_try_charge(NULL, gfp, delta / PAGE_SIZE,
+      &_memcg, may_oom);
+  if (ret == -ENOMEM)
+   return ret;
+ }
+
+ if (memcg && _memcg == memcg)
+  ret = res_counter_charge(&memcg->kmem, delta, &fail_res);
+
+ return ret;
+}
+
+void
+memcg_uncharge_kmem(struct mem_cgroup *memcg, long long delta)
+{
+ if (memcg)
+  res_counter_uncharge(&memcg->kmem, delta);
+
+ if (memcg && !mem_cgroup_test_flag(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;
+ res_counter_init(&memcg->kmem, parent_res);
+}
```

```
+#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 v2 03/13] memcg: Uncharge all kmem when deleting a cgroup.
Posted by Suleiman Souhlal on Fri, 09 Mar 2012 20:39:06 GMT
View Forum Message <> Reply to Message

```
Signed-off-by: Suleiman Souhlal <suleiman@google.com>
---
 mm/memcontrol.c |   31 +++++++++++++++++++++++++++++-
 1 files changed, 30 insertions(+), 1 deletions(-)

diff --git a/mm/memcontrol.c b/mm/memcontrol.c
index e6fd558..6fbb438 100644
--- a/mm/memcontrol.c
+++ b/mm/memcontrol.c
@@ -382,6 +382,7 @@ 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);

 static inline bool
 mem_cgroup_test_flag(const struct mem_cgroup *memcg, enum memcg_flags flag)
@@ -3700,6 +3701,7 @@ 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);
@@ -3719,6 +3721,8 @@ 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) {
@@ -3740,8 +3744,14 @@ 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 && !mem_cgroup_test_flag(memcg,
+      MEMCG_INDEPENDENT_KMEM_LIMIT))
+   usage -= memcg->kmem.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;
@@ -5689,9 +5699,28 @@ memcg_kmem_init(struct mem_cgroup *memcg, struct mem_cgroup
*parent)
   parent_res = &parent->kmem;
  res_counter_init(&memcg->kmem, parent_res);
 }
+
+static void
+memcg_kmem_move(struct mem_cgroup *memcg)
+{
+ unsigned long flags;
+ long kmem;
+
+ spin_lock_irqsave(&memcg->kmem.lock, flags);
+ kmem = memcg->kmem.usage;
+ res_counter_uncharge_locked(&memcg->kmem, kmem);
+ spin_unlock_irqrestore(&memcg->kmem.lock, flags);
+ if (!mem_cgroup_test_flag(memcg, MEMCG_INDEPENDENT_KMEM_LIMIT))
+  res_counter_uncharge(&memcg->res, kmem);
+}
 #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 */
--
```

Subject: [PATCH v2 04/13] memcg: Make it possible to use the stock for more than one page.
Posted by Suleiman Souhlal on Fri, 09 Mar 2012 20:39:07 GMT
View Forum Message <> Reply to Message

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

diff --git a/mm/memcontrol.c b/mm/memcontrol.c
index 6fbb438..f605100 100644
--- a/mm/memcontrol.c
+++ b/mm/memcontrol.c
@@ -1965,19 +1965,19 @@ 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);
@@ -2290,7 +2290,7 @@ 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 {
@@ -2315,7 +2315,7 @@ again:
    rcu_read_unlock();
    goto done;
   }
-  if (nr_pages == 1 && consume_stock(memcg)) {
+  if (consume_stock(memcg, nr_pages)) {
    /*
     * It seems dagerous to access memcg without css_get().
     * But considering how consume_stok works, it's not
--
1.7.7.3
```

Subject: [PATCH v2 06/13] slab: Add kmem_cache_gfp_flags() helper function.
Posted by Suleiman Souhlal on Fri, 09 Mar 2012 20:39:09 GMT
View Forum Message <> Reply to Message

This function returns the gfp flags that are always applied to
allocations of a kmem_cache.

Signed-off-by: Suleiman Souhlal <suleiman@google.com>
---
 include/linux/slab_def.h | 6 ++++++
 include/linux/slob_def.h | 6 ++++++
 include/linux/slub_def.h | 6 ++++++
 3 files changed, 18 insertions(+), 0 deletions(-)

```
diff --git a/include/linux/slab_def.h b/include/linux/slab_def.h
index fbd1117..25f9a6a 100644
--- a/include/linux/slab_def.h
+++ b/include/linux/slab_def.h
@@ -159,6 +159,12 @@ found:
  return __kmalloc(size, flags);
 }

+static inline gfp_t
+kmem_cache_gfp_flags(struct kmem_cache *cachep)
+{
+ return cachep->gfpflags;
+}
+
 #ifdef CONFIG_NUMA
```

```
 extern void *__kmalloc_node(size_t size, gfp_t flags, int node);
 extern void *kmem_cache_alloc_node(struct kmem_cache *, gfp_t flags, int node);
diff --git a/include/linux/slob_def.h b/include/linux/slob_def.h
index 0ec00b3..3fa527d 100644
--- a/include/linux/slob_def.h
+++ b/include/linux/slob_def.h
@@ -34,4 +34,10 @@ static __always_inline void *__kmalloc(size_t size, gfp_t flags)
  return kmalloc(size, flags);
 }

+static inline gfp_t
+kmem_cache_gfp_flags(struct kmem_cache *cachep)
+{
+ return 0;
+}
+
 #endif /* __LINUX_SLOB_DEF_H */
diff --git a/include/linux/slub_def.h b/include/linux/slub_def.h
index a32bcfd..5911d81 100644
--- a/include/linux/slub_def.h
+++ b/include/linux/slub_def.h
@@ -313,4 +313,10 @@ static __always_inline void *kmalloc_node(size_t size, gfp_t flags, int
node)
 }
 #endif

+static inline gfp_t
+kmem_cache_gfp_flags(struct kmem_cache *cachep)
+{
+ return cachep->allocflags;
+}
+
 #endif /* _LINUX_SLUB_DEF_H */
--
1.7.7.3
```

Subject: [PATCH v2 07/13] memcg: Slab accounting.
Posted by Suleiman Souhlal on Fri, 09 Mar 2012 20:39:10 GMT
View Forum Message <> Reply to Message

Introduce per-cgroup kmem_caches for memcg slab accounting, that
get created asynchronously the first time we do an allocation of
that type in the cgroup.
The cgroup cache gets used in subsequent allocations, and permits
accounting of slab on a per-page basis.

For a slab type to getaccounted, the SLAB_MEMCG_ACCT flag has to be

passed to kmem_cache_create().

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.

Only allocations that can be attributed to a cgroup get charged.

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 | 41 ++++++
 include/linux/slab_def.h | 72 ++++++++++-
 include/linux/slub_def.h | 3 +
 init/Kconfig | 2 +-
 mm/memcontrol.c | 290 ++
 mm/slab.c | 248 +++++++++++++++++++++++++++++++++++---
 7 files changed, 663 insertions(+), 23 deletions(-)

diff --git a/include/linux/memcontrol.h b/include/linux/memcontrol.h
index b80de52..b6b8388 100644
--- a/include/linux/memcontrol.h
+++ b/include/linux/memcontrol.h
@@ -416,13 +416,41 @@ 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..bc9f87f 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_ACCT  0x00008000UL /* Accounted by memcg */
 #define SLAB_STORE_USER  0x00010000UL /* DEBUG: Store the last owner for bug hunting */
 #define SLAB_PANIC  0x00040000UL /* Panic if kmem_cache_create() fails */
 /*
@@ -153,6 +154,21 @@ unsigned int kmem_cache_size(struct kmem_cache *);
 #define ARCH_SLAB_MINALIGN __alignof__(unsigned long long)
 #endif

+#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
+struct mem_cgroup_cache_params {
+ struct mem_cgroup *memcg;
+ int id;
+ int obj_size;
+
+ /* Original cache parameters, used when creating a memcg cache */
+ size_t orig_align;
+ unsigned long orig_flags;
```

```
+ struct kmem_cache *orig_cache;
+
+ struct list_head destroyed_list; /* Used when deleting cpuset cache */
+};
+#endif
+
 /*
  * Common kmalloc functions provided by all allocators
  */
@@ -353,4 +369,29 @@ static inline void *kzalloc_node(size_t size, gfp_t flags, int node)

 void __init kmem_cache_init_late(void);

+#if defined(CONFIG_CGROUP_MEM_RES_CTLR_KMEM) && defined(CONFIG_SLAB)
+
+#define MAX_KMEM_CACHE_TYPES 400
+
+struct kmem_cache *kmem_cache_create_memcg(struct kmem_cache *cachep,
+    char *name);
+void kmem_cache_drop_ref(struct kmem_cache *cachep);
+
+#else /* !CONFIG_CGROUP_MEM_RES_CTLR_KMEM || !CONFIG_SLAB */
+
+#define MAX_KMEM_CACHE_TYPES 0
+
+static inline struct kmem_cache *
+kmem_cache_create_memcg(struct kmem_cache *cachep,
+    char *name)
+{
+ return NULL;
+}
+
+static inline void
+kmem_cache_drop_ref(struct kmem_cache *cachep)
+{
+}
+#endif /* CONFIG_CGROUP_MEM_RES_CTLR_KMEM && CONFIG_SLAB */
+
 #endif /* _LINUX_SLAB_H */
diff --git a/include/linux/slab_def.h b/include/linux/slab_def.h
index 25f9a6a..248b8a9 100644
--- a/include/linux/slab_def.h
+++ b/include/linux/slab_def.h
@@ -51,7 +51,7 @@ struct kmem_cache {
  void (*ctor)(void *obj);

 /* 4) cache creation/removal */
- const char *name;
```

```
+ char *name;
  struct list_head next;

 /* 5) statistics */
@@ -81,6 +81,12 @@ struct kmem_cache {
  int obj_size;
 #endif /* CONFIG_DEBUG_SLAB */

+#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
+ struct mem_cgroup_cache_params memcg_params;
+
+ atomic_t refcnt;
+#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
@@ -218,4 +224,68 @@ found:

 #endif /* CONFIG_NUMA */

+#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
+
+void kmem_cache_drop_ref(struct kmem_cache *cachep);
+
+static inline void
+kmem_cache_get_ref(struct kmem_cache *cachep)
+{
+ if (cachep->memcg_params.id == -1 &&
+     unlikely(!atomic_add_unless(&cachep->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/include/linux/slub_def.h b/include/linux/slub_def.h
index 5911d81..09b602d 100644
--- a/include/linux/slub_def.h
+++ b/include/linux/slub_def.h
@@ -99,6 +99,9 @@ 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
  /*
diff --git a/init/Kconfig b/init/Kconfig
index 3f42cd6..e7eb652 100644
--- a/init/Kconfig
+++ b/init/Kconfig
@@ -705,7 +705,7 @@ 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 2576a2b..a5593cf 100644
--- a/mm/memcontrol.c
+++ b/mm/memcontrol.c
@@ -302,6 +302,11 @@ 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
 };

 /* Stuffs for move charges at task migration. */
@@ -5692,6 +5697,287 @@ memcg_uncharge_kmem(struct mem_cgroup *memcg, long long
delta)
   res_counter_uncharge(&memcg->res, delta);
 }

+static struct kmem_cache *
+memcg_create_kmem_cache(struct mem_cgroup *memcg, struct kmem_cache *cachep)
+{
+ struct kmem_cache *new_cachep;
+ struct dentry *dentry;
+ char *name;
+ int idx;
+
+ idx = cachep->memcg_params.id;
+
```

```
+ dentry = memcg->css.cgroup->dentry;
+ BUG_ON(dentry == NULL);
+
+ /* Preallocate the space for "dead" at the end */
+ name = kasprintf(GFP_KERNEL, "%s(%d:%s)dead",
+     cachep->name, css_id(&memcg->css), dentry->d_name.name);
+ if (name == NULL)
+  return cachep;
+ /* Remove "dead" */
+ name[strlen(name) - 4] = '\0';
+
+ 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_params.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);
+  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);
+  /* Drop the reference gotten when we enqueued. */
+  css_put(&cw->memcg->css);
+  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);
+
+/*
+ * 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(&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);
+
+ /* 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(&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 mem_cgroup *memcg;
+ int idx;
+
+ rcu_read_lock();
+
+ if (in_interrupt())
+  return cachep;
+ if (current == NULL)
+  return cachep;
+
+ if (!(cachep->flags & SLAB_MEMCG_ACCT))
+  return cachep;
+
+ gfp |= kmem_cache_gfp_flags(cachep);
+ if (gfp & __GFP_NOFAIL)
+  return cachep;
+
+ if (cachep->memcg_params.memcg)
+  return cachep;
+
+ memcg = mem_cgroup_from_task(current);
+ idx = cachep->memcg_params.id;
+
+ if (memcg == NULL || memcg == root_mem_cgroup)
+  return cachep;
+
+ VM_BUG_ON(idx == -1);
+
+ if (rcu_access_pointer(memcg->slabs[idx]) == NULL) {
+  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_params.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();
+ memcg = cachep->memcg_params.memcg;
+
+ 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();
+ memcg = cachep->memcg_params.memcg;
+
+ 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_params.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);
+  }
+ }
+}
+
 static void
 memcg_kmem_init(struct mem_cgroup *memcg, struct mem_cgroup *parent)
 {
@@ -5701,6 +5987,8 @@ memcg_kmem_init(struct mem_cgroup *memcg, struct mem_cgroup
*parent)
  if (parent && parent != root_mem_cgroup)
   parent_res = &parent->kmem;
  res_counter_init(&memcg->kmem, parent_res);
+
+ memcg_slab_init(memcg);
 }

 static void
@@ -5709,6 +5997,8 @@ memcg_kmem_move(struct mem_cgroup *memcg)
  unsigned long flags;
  long kmem;
```

```
+ memcg_slab_move(memcg);
+
  spin_lock_irqsave(&memcg->kmem.lock, flags);
  kmem = memcg->kmem.usage;
  res_counter_uncharge_locked(&memcg->kmem, kmem);
diff --git a/mm/slab.c b/mm/slab.c
index f0bd785..95b024c 100644
--- a/mm/slab.c
+++ b/mm/slab.c
@@ -159,13 +159,15 @@
     SLAB_STORE_USER | \
     SLAB_RECLAIM_ACCOUNT | SLAB_PANIC | \
     SLAB_DESTROY_BY_RCU | SLAB_MEM_SPREAD | \
-    SLAB_DEBUG_OBJECTS | SLAB_NOLEAKTRACE | SLAB_NOTRACK)
+    SLAB_DEBUG_OBJECTS | SLAB_NOLEAKTRACE | \
+    SLAB_NOTRACK | SLAB_MEMCG_ACCT)
 #else
 # define CREATE_MASK (SLAB_HWCACHE_ALIGN | \
     SLAB_CACHE_DMA | \
     SLAB_RECLAIM_ACCOUNT | SLAB_PANIC | \
     SLAB_DESTROY_BY_RCU | SLAB_MEM_SPREAD | \
-    SLAB_DEBUG_OBJECTS | SLAB_NOLEAKTRACE | SLAB_NOTRACK)
+    SLAB_DEBUG_OBJECTS | SLAB_NOLEAKTRACE | \
+    SLAB_NOTRACK | SLAB_MEMCG_ACCT)
 #endif

 /*
@@ -301,6 +303,8 @@ 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 +330,11 @@ 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 +1765,23 @@ 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 +1814,8 @@ 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 +2241,17 @@ 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 +2268,6 @@ 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 +2288,12 @@ 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;
+   }
   }
  }

@@ -2369,6 +2390,13 @@ kmem_cache_create (const char *name, size_t size, size_t align,
   goto oops;

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

```
  cachep->obj_size = size;

@@ -2477,7 +2505,23 @@ 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->memcg_params.orig_cache = NULL;
+ atomic_set(&cachep->refcnt, 1);
+ INIT_LIST_HEAD(&cachep->memcg_params.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->memcg_params.id = id;
+ } else
+  cachep->memcg_params.id = -1;
+#endif

   if (setup_cpu_cache(cachep, gfp)) {
    __kmem_cache_destroy(cachep);
@@ -2502,13 +2546,53 @@ 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->memcg_params.orig_flags & ~SLAB_PANIC;
+ mutex_lock(&cache_chain_mutex);
+ new = __kmem_cache_create(name, cachep->memcg_params.obj_size,
+     cachep->memcg_params.orig_align, flags, cachep->ctor, 1);
+ if (new == NULL) {
+  mutex_unlock(&cache_chain_mutex);
+  return NULL;
+ }
+ new->memcg_params.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);
+ mutex_unlock(&cache_chain_mutex);
+
+ return new;
+}
+#endif /* CONFIG_CGROUP_MEM_RES_CTLR_KMEM */
+
 #if DEBUG
 static void check_irq_off(void)
 {
@@ -2703,12 +2787,74 @@ void kmem_cache_destroy(struct kmem_cache *cachep)
  if (unlikely(cachep->flags & SLAB_DESTROY_BY_RCU))
   rcu_barrier();

+#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
+ /* Not a memcg cache */
+ if (cachep->memcg_params.id != -1) {
+  __clear_bit(cachep->memcg_params.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 = container_of(list_first_entry(&destroyed_caches,
+      struct mem_cgroup_cache_params, destroyed_list), struct
+      kmem_cache, memcg_params);
+  name = (char *)cachep->name;
+  list_del(&cachep->memcg_params.destroyed_list);
+  spin_unlock_irq(&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->memcg_params.id != -1);
+
+ /*
+  * We have to defer the actual destroying to a workqueue, because
+  * we might currently be in a context that cannot sleep.
+  */
+ spin_lock_irqsave(&destroy_lock, flags);
+ list_add(&cachep->memcg_params.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->memcg_params.id == -1 &&
+     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 +3054,10 @@ static int cache_grow(struct kmem_cache *cachep,

  offset *= cachep->colour_off;

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

  /*
   * The test for missing atomic flag is performed here, rather than
@@ -2938,8 +3086,10 @@ static int cache_grow(struct kmem_cache *cachep,

  cache_init_objs(cachep, slabp);

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

@@ -2952,8 +3102,10 @@ static int cache_grow(struct kmem_cache *cachep,
 opps1:
  kmem_freepages(cachep, objp);
 failed:
- if (local_flags & __GFP_WAIT)
+ if (local_flags & __GFP_WAIT) {
   local_irq_disable();
+  mem_cgroup_kmem_cache_finish_sleep(cachep);
+ }
  return 0;
 }
```

```
@@ -3712,10 +3864,14 @@ 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 +3883,12 @@ 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 +3897,16 @@ 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 +3920,13 @@ 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);
@@ -3866,9 +4030,35 @@ 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);
+   VM_BUG_ON(actual_cachep->memcg_params.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 +4086,19 @@ 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 +4367,13 @@ 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->refcnt, 1, 0))
+   continue;
+#endif
+
   /*
    * We only take the l3 lock if absolutely necessary and we
    * have established with reasonable certainty that
@@ -4199,6 +4406,7 @@ static void cache_reap(struct work_struct *w)
    STATS_ADD_REAPED(searchp, freed);
   }
 next:
+  kmem_cache_drop_ref(searchp);
   cond_resched();
  }
  check_irq_on();
--
1.7.7.3
```

Subject: [PATCH v2 08/13] memcg: Make dentry slab memory accounted in kernel memory accounting.
Posted by Suleiman Souhlal on Fri, 09 Mar 2012 20:39:11 GMT
View Forum Message <> Reply to Message

Signed-off-by: <suleiman@google.com>
---
 fs/dcache.c | 4 ++--
 1 files changed, 2 insertions(+), 2 deletions(-)

diff --git a/fs/dcache.c b/fs/dcache.c
index bcbdb33..cc6dd92 100644
--- a/fs/dcache.c
+++ b/fs/dcache.c
@@ -3022,8 +3022,8 @@ static void __init dcache_init(void)
   * but it is probably not worth it because of the cache nature
   * of the dcache.
   */
- dentry_cache = KMEM_CACHE(dentry,
- SLAB_RECLAIM_ACCOUNT|SLAB_PANIC|SLAB_MEM_SPREAD);
+ dentry_cache = KMEM_CACHE(dentry, SLAB_RECLAIM_ACCOUNT | SLAB_PANIC |
+   SLAB_MEM_SPREAD | SLAB_MEMCG_ACCT);

  /* Hash may have been set up in dcache_init_early */
  if (!hashdist)
--
1.7.7.3

Subject: [PATCH v2 09/13] memcg: Account for kmalloc in kernel memory accounting.
Posted by Suleiman Souhlal on Fri, 09 Mar 2012 20:39:12 GMT
View Forum Message <> Reply to Message

In order to do this, we have to create a kmalloc_no_account()
function that is used for kmalloc allocations that we do not
want to account, because the kernel memory accounting code has
to make some kmalloc allocations and is not allowed to recurse.

Signed-off-by: Suleiman Souhlal <suleiman@google.com>
---
 include/linux/slab.h | 8 ++++++++
 mm/memcontrol.c | 2 +-
 mm/slab.c | 42 ++++++++++++++++++++++++++++++++++++++----
 3 files changed, 48 insertions(+), 4 deletions(-)

diff --git a/include/linux/slab.h b/include/linux/slab.h
index bc9f87f..906a015 100644

```
--- a/include/linux/slab.h
+++ b/include/linux/slab.h
@@ -377,6 +377,8 @@ struct kmem_cache *kmem_cache_create_memcg(struct kmem_cache
*cachep,
     char *name);
 void kmem_cache_drop_ref(struct kmem_cache *cachep);

+void *kmalloc_no_account(size_t size, gfp_t flags);
+
 #else /* !CONFIG_CGROUP_MEM_RES_CTLR_KMEM || !CONFIG_SLAB */

 #define MAX_KMEM_CACHE_TYPES 0
@@ -392,6 +394,12 @@ static inline void
 kmem_cache_drop_ref(struct kmem_cache *cachep)
 {
 }
+
+static inline void *kmalloc_no_account(size_t size, gfp_t flags)
+{
+ return kmalloc(size, flags);
+}
+
 #endif /* CONFIG_CGROUP_MEM_RES_CTLR_KMEM && CONFIG_SLAB */

 #endif /* _LINUX_SLAB_H */
diff --git a/mm/memcontrol.c b/mm/memcontrol.c
index a5593cf..72e83af 100644
--- a/mm/memcontrol.c
+++ b/mm/memcontrol.c
@@ -5824,7 +5824,7 @@ memcg_create_cache_enqueue(struct mem_cgroup *memcg, struct
kmem_cache *cachep)
  if (!css_tryget(&memcg->css))
   return;

- cw = kmalloc(sizeof(struct create_work), GFP_NOWAIT);
+ cw = kmalloc_no_account(sizeof(struct create_work), GFP_NOWAIT);
  if (cw == NULL) {
   css_put(&memcg->css);
   return;
diff --git a/mm/slab.c b/mm/slab.c
index 95b024c..7af7384 100644
--- a/mm/slab.c
+++ b/mm/slab.c
@@ -1623,8 +1623,8 @@ void __init kmem_cache_init(void)
    sizes->cs_cachep = kmem_cache_create(names->name,
      sizes->cs_size,
      ARCH_KMALLOC_MINALIGN,
-     ARCH_KMALLOC_FLAGS|SLAB_PANIC,
```

```
-     NULL);
+     ARCH_KMALLOC_FLAGS | SLAB_PANIC |
+     SLAB_MEMCG_ACCT, NULL);
   }
 #ifdef CONFIG_ZONE_DMA
   sizes->cs_dmacachep = kmem_cache_create(
@@ -3936,11 +3936,16 @@ 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)
@@ -3986,14 +3991,31 @@ 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;
 }

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

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

 #if defined(CONFIG_DEBUG_SLAB) || defined(CONFIG_TRACING)
 void *__kmalloc(size_t size, gfp_t flags)
@@ -4008,12 +4030,26 @@ void *__kmalloc_track_caller(size_t size, gfp_t flags, unsigned long
caller)
 }
 EXPORT_SYMBOL(__kmalloc_track_caller);

+void *
+kmalloc_no_account(size_t size, gfp_t flags)
+{
+ return __do_kmalloc_no_account(size, flags,
+      __builtin_return_address(0));
+}
+EXPORT_SYMBOL(kmalloc_no_account);
 #else
 void *__kmalloc(size_t size, gfp_t flags)
 {
  return __do_kmalloc(size, flags, NULL);
 }
 EXPORT_SYMBOL(__kmalloc);
+
+void *
+kmalloc_no_account(size_t size, gfp_t flags)
+{
+ return __do_kmalloc_no_account(size, flags, NULL);
+}
+EXPORT_SYMBOL(kmalloc_no_account);
 #endif

 /**
--
1.7.7.3
```

Subject: [PATCH v2 10/13] memcg: Track all the memcg children of a kmem_cache.

Posted by Suleiman Souhlal on Fri, 09 Mar 2012 20:39:13 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.h | 1 +
 mm/slab.c | 53 ++---
 2 files changed, 50 insertions(+), 4 deletions(-)

diff --git a/include/linux/slab.h b/include/linux/slab.h
index 906a015..fd1d2ba 100644
--- a/include/linux/slab.h
+++ b/include/linux/slab.h
@@ -166,6 +166,7 @@ struct mem_cgroup_cache_params {
  struct kmem_cache *orig_cache;

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

diff --git a/mm/slab.c b/mm/slab.c
index 7af7384..02239ed 100644
--- a/mm/slab.c
+++ b/mm/slab.c
@@ -2511,6 +2511,7 @@ __kmem_cache_create(const char *name, size_t size, size_t align,
  cachep->memcg_params.orig_cache = NULL;
  atomic_set(&cachep->refcnt, 1);
  INIT_LIST_HEAD(&cachep->memcg_params.destroyed_list);
+ INIT_LIST_HEAD(&cachep->memcg_params.sibling_list);

  if (!memcg) {
   int id;
@@ -2582,6 +2583,8 @@ kmem_cache_create_memcg(struct kmem_cache *cachep, char
*name)
  }
  new->memcg_params.orig_cache = cachep;

+ 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))
@@ -2769,6 +2772,29 @@ void kmem_cache_destroy(struct kmem_cache *cachep)
 {

```
  BUG_ON(!cachep || in_interrupt());

+#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
+ /* Destroy all the children caches if we aren't a memcg cache */
+ if (cachep->memcg_params.id != -1) {
+  struct kmem_cache *c;
+  struct mem_cgroup_cache_params *p, *tmp;
+
+  mutex_lock(&cache_chain_mutex);
+  list_for_each_entry_safe(p, tmp,
+      &cachep->memcg_params.sibling_list, sibling_list) {
+   c = container_of(p, struct kmem_cache, memcg_params);
+   if (c == cachep)
+    continue;
+   mutex_unlock(&cache_chain_mutex);
+   BUG_ON(c->memcg_params.id != -1);
+   mem_cgroup_remove_child_kmem_cache(c,
+       cachep->memcg_params.id);
+   kmem_cache_destroy(c);
+   mutex_lock(&cache_chain_mutex);
+  }
+  mutex_unlock(&cache_chain_mutex);
+ }
+#endif /* CONFIG_CGROUP_MEM_RES_CTLR_KMEM */
+
  /* Find the cache in the chain of caches. */
  get_online_cpus();
  mutex_lock(&cache_chain_mutex);
@@ -2776,6 +2802,9 @@ void kmem_cache_destroy(struct kmem_cache *cachep)
   * the chain is never empty, cache_cache is never destroyed
   */
  list_del(&cachep->next);
+#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
+ list_del(&cachep->memcg_params.sibling_list);
+#endif
  if (__cache_shrink(cachep)) {
   slab_error(cachep, "Can't free all objects");
   list_add(&cachep->next, &cache_chain);
@@ -4654,11 +4683,27 @@ static ssize_t slabinfo_write(struct file *file, const char __user
*buffer,
    if (limit < 1 || batchcount < 1 ||
      batchcount > limit || shared < 0) {
     res = 0;
-   } else {
-    res = do_tune_cpucache(cachep, limit,
-            batchcount, shared,
-            GFP_KERNEL);
+    break;
```

```
   }
+
+   res = do_tune_cpucache(cachep, limit, batchcount,
+       shared, GFP_KERNEL);
+
+#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
+   {
+    struct kmem_cache *c;
+    struct mem_cgroup_cache_params *p;
+
+    list_for_each_entry(p,
+        &cachep->memcg_params.sibling_list,
+        sibling_list) {
+     c = container_of(p, struct kmem_cache,
+         memcg_params);
+     do_tune_cpucache(c, limit, batchcount,
+         shared, GFP_KERNEL);
+    }
+   }
+#endif
    break;
  }
 }
--
1.7.7.3
```

Subject: [PATCH v2 11/13] memcg: Handle bypassed kernel memory charges.
Posted by Suleiman Souhlal on Fri, 09 Mar 2012 20:39:14 GMT
View Forum Message <> Reply to Message

When __mem_cgroup_try_charge() decides to bypass a slab charge
(because we are getting OOM killed or have a fatal signal pending),
we may end up with a slab that belongs to a memcg, but wasn't
charged to it. When we free such a slab page, we end up uncharging
it from the memcg, even though it was never charged, which may
lead to res_counter underflows.

To avoid this, when a charge is bypassed, we force the charge,
without checking for the bypass conditions or doing any reclaim.
This may cause the cgroup's usage to temporarily go above its limit.

Signed-off-by: Suleiman Souhlal <suleiman@google.com>
---
 mm/memcontrol.c | 15 +++++++++++++--
 1 files changed, 13 insertions(+), 2 deletions(-)

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

```
index 72e83af..9f5e9d8 100644
--- a/mm/memcontrol.c
+++ b/mm/memcontrol.c
@@ -5672,16 +5672,27 @@ memcg_charge_kmem(struct mem_cgroup *memcg, gfp_t gfp, long
long delta)

  ret = 0;

- _memcg = memcg;
  if (memcg && !mem_cgroup_test_flag(memcg,
      MEMCG_INDEPENDENT_KMEM_LIMIT)) {
+  _memcg = memcg;
   ret = __mem_cgroup_try_charge(NULL, gfp, delta / PAGE_SIZE,
       &_memcg, may_oom);
   if (ret == -ENOMEM)
    return ret;
+  else if (ret == -EINTR) {
+   /*
+    * __mem_cgroup_try_charge() chose to bypass to root due
+    * to OOM kill or fatal signal.
+    * Since our only options are to either fail the
+    * allocation or charge it to this cgroup, force the
+    * change, going above the limit if needed.
+    */
+   ret = res_counter_charge_nofail(&memcg->res, delta,
+       &fail_res);
+  }
  }

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

  return ret;
--
1.7.7.3
```

Subject: [PATCH v2 12/13] memcg: Per-memcg memory.kmem.slabinfo file.
Posted by Suleiman Souhlal on Fri, 09 Mar 2012 20:39:15 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.h | 6 +++
 include/linux/slab_def.h | 1 +

```
 mm/memcontrol.c         |   18 +++++++++
 mm/slab.c               |   88 ++++++++++++++++++++++++++++++++++------------
 4 files changed, 90 insertions(+), 23 deletions(-)

diff --git a/include/linux/slab.h b/include/linux/slab.h
index fd1d2ba..0ff5ee2 100644
--- a/include/linux/slab.h
+++ b/include/linux/slab.h
@@ -401,6 +401,12 @@ static inline void *kmalloc_no_account(size_t size, gfp_t flags)
  return kmalloc(size, flags);
 }

+static inline int
+mem_cgroup_slabinfo(struct mem_cgroup *mem, struct seq_file *m)
+{
+ return 0;
+}
+
 #endif /* CONFIG_CGROUP_MEM_RES_CTLR_KMEM && CONFIG_SLAB */

 #endif /* _LINUX_SLAB_H */
diff --git a/include/linux/slab_def.h b/include/linux/slab_def.h
index 248b8a9..fa6b272 100644
--- a/include/linux/slab_def.h
+++ b/include/linux/slab_def.h
@@ -227,6 +227,7 @@ found:
 #ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM

 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)
diff --git a/mm/memcontrol.c b/mm/memcontrol.c
index 9f5e9d8..4a4fa48 100644
--- a/mm/memcontrol.c
+++ b/mm/memcontrol.c
@@ -4672,6 +4672,20 @@ static int mem_cgroup_independent_kmem_limit_write(struct cgroup
*cgrp,
  return 0;
 }

+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.independent_kmem_limit",
@@ -4689,6 +4703,10 @@ static struct cftype kmem_cgroup_files[] = {
   .private = MEMFILE_PRIVATE(_KMEM, RES_USAGE),
   .read_u64 = mem_cgroup_read,
  },
+ {
+  .name = "kmem.slabinfo",
+  .read_seq_string = mem_cgroup_slabinfo_show,
+ },
 };

 static int register_kmem_files(struct cgroup *cont, struct cgroup_subsys *ss)
diff --git a/mm/slab.c b/mm/slab.c
index 02239ed..1b35799 100644
--- a/mm/slab.c
+++ b/mm/slab.c
@@ -4528,21 +4528,26 @@ 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)
@@ -4554,31 +4559,43 @@ 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;
@@ -4586,12 +4603,12 @@ 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;
@@ -4726,6 +4743,31 @@ 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_params.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 v2 13/13] memcg: Document kernel memory accounting.
Posted by Suleiman Souhlal on Fri, 09 Mar 2012 20:39:16 GMT
View Forum Message <> Reply to Message

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

 Documentation/cgroups/memory.txt | 44 ++++++++++++++++++++++++++++++++++++---
 1 files changed, 40 insertions(+), 4 deletions(-)

```
diff --git a/Documentation/cgroups/memory.txt b/Documentation/cgroups/memory.txt
index 4c95c00..73f2e38 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

@@ -265,11 +270,19 @@ the amount of kernel memory used by the system. Kernel memory is fundamentally
 different than user memory, since it can't be swapped out, which makes it
 possible to DoS the system by consuming too much of this precious resource.

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

-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 becomes unaccounted.
+
+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 +292,29 @@ per cgroup, instead of globally.

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

+* slab memory.
+
+2.7.1.1 Slab memory accounting
+
+Any slab type created with the SLAB_MEMCG_ACCT kmem_cache_create() flag
+is accounted.
+
+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.
+
+Only slab memory that can be attributed to a cgroup gets accounted in this
+fashion.
+
+A per-cgroup kmem_cache is named like the original, with the cgroup's name
+in parentheses.
+

+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
--
1.7.7.3

Subject: Re: [PATCH v2 00/13] Memcg Kernel Memory Tracking.
Posted by Suleiman Souhlal on Sat, 10 Mar 2012 06:25:58 GMT
View Forum Message <> Reply to Message

On Fri, Mar 9, 2012 at 12:39 PM, Suleiman Souhlal <ssouhlal@freebsd.org> wrote:
> This is v2 of my kernel memory tracking patchset for memcg.

I just realized that I forgot to test without the config option
enabled, so that doesn't compile. :-(

I will make sure to fix this in v3 and test more thoroughly.

Hopefully this shouldn't impede discussion on the patchset.

Sorry about that.
-- Suleiman
