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

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

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

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

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

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

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

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

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

Signed-off-by: Glauber Costa <glommer@parallels.com>
CC: Christoph Lameter <cl@linux.com>
CC: Pekka Enberg <penberg@cs.helsinki.fi>
CC: Michal Hocko <mhocko@suse.cz>
CC: Kamezawa Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com>
CC: Johannes Weiner <hannes@cmpxchg.org>
CC: Suleiman Souhlal <suleiman@google.com>
---
 include/linux/memcontrol.h | 68 ++++++++
 init/Kconfig | 2 +-
 mm/memcontrol.c | 373 ++-
 3 files changed, 441 insertions(+), 2 deletions(-)

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

+void mem_cgroup_flush_cache_create_queue(void);
+void mem_cgroup_remove_child_kmem_cache(struct kmem_cache *cachep, int id);
+bool __mem_cgroup_charge_slab(struct kmem_cache *cachep, gfp_t gfp,
+         size_t size);
+void __mem_cgroup_uncharge_slab(struct kmem_cache *cachep, size_t size);
+
+bool __mem_cgroup_charge_kmem(gfp_t gfp, size_t size);
+void __mem_cgroup_uncharge_kmem(size_t size);
+
+struct kmem_cache *
+__mem_cgroup_get_kmem_cache(struct kmem_cache *cachep, gfp_t gfp);
+
+#define mem_cgroup_kmem_on 1
+
+void mem_cgroup_destroy_cache(struct kmem_cache *cachep);
 #else
 static inline void mem_cgroup_register_cache(struct mem_cgroup *memcg,
          struct kmem_cache *s)
@@ -464,6 +479,59 @@ static inline void sock_update_memcg(struct sock *sk)
 static inline void sock_release_memcg(struct sock *sk)
 {
 }
+
+static inline void
+mem_cgroup_flush_cache_create_queue(void)
+{
+}
+
+static inline void mem_cgroup_destroy_cache(struct kmem_cache *cachep)
+{
+}
+
+#define mem_cgroup_kmem_on 0
+#define __mem_cgroup_get_kmem_cache(a, b) a
+#define __mem_cgroup_charge_slab(a, b, c) false
+#define __mem_cgroup_charge_kmem(a, b) false
+#define __mem_cgroup_uncharge_slab(a, b)
+#define __mem_cgroup_uncharge_kmem(b)
 #endif /* CONFIG_CGROUP_MEM_RES_CTLR_KMEM */
+static __always_inline struct kmem_cache *
```

```
+mem_cgroup_get_kmem_cache(struct kmem_cache *cachep, gfp_t gfp)
+{
+ if (mem_cgroup_kmem_on)
+  return __mem_cgroup_get_kmem_cache(cachep, gfp);
+ return cachep;
+}
+
+static __always_inline bool
+mem_cgroup_charge_slab(struct kmem_cache *cachep, gfp_t gfp, size_t size)
+{
+ if (mem_cgroup_kmem_on)
+  return __mem_cgroup_charge_slab(cachep, gfp, size);
+ return true;
+}
+
+static __always_inline void
+mem_cgroup_uncharge_slab(struct kmem_cache *cachep, size_t size)
+{
+ if (mem_cgroup_kmem_on)
+  __mem_cgroup_uncharge_slab(cachep, size);
+}
+
+static __always_inline
+bool mem_cgroup_charge_kmem(gfp_t gfp, size_t size)
+{
+ if (mem_cgroup_kmem_on)
+  return __mem_cgroup_charge_kmem(gfp, size);
+ return true;
+}
+
+static __always_inline
+void mem_cgroup_uncharge_kmem(size_t size)
+{
+ if (mem_cgroup_kmem_on)
+  __mem_cgroup_uncharge_kmem(size);
+}
 #endif /* _LINUX_MEMCONTROL_H */

diff --git a/init/Kconfig b/init/Kconfig
index 72f33fa..071b7e3 100644
--- a/init/Kconfig
+++ b/init/Kconfig
@@ -696,7 +696,7 @@ config CGROUP_MEM_RES_CTLR_SWAP_ENABLED
    then swapaccount=0 does the trick).
 config CGROUP_MEM_RES_CTLR_KMEM
  bool "Memory Resource Controller Kernel Memory accounting (EXPERIMENTAL)"
- depends on CGROUP_MEM_RES_CTLR && EXPERIMENTAL
+ depends on CGROUP_MEM_RES_CTLR && EXPERIMENTAL && !SLOB
```

```
  default n
  help
    The Kernel Memory extension for Memory Resource Controller can limit
diff --git a/mm/memcontrol.c b/mm/memcontrol.c
index e881d83..ae61e99 100644
--- a/mm/memcontrol.c
+++ b/mm/memcontrol.c
@@ -10,6 +10,10 @@
  * Copyright (C) 2009 Nokia Corporation
  * Author: Kirill A. Shutemov
  *
+ * Kernel Memory Controller
+ * Copyright (C) 2012 Parallels Inc. and Google Inc.
+ * Authors: Glauber Costa and Suleiman Souhlal
+ *
  * This program is free software; you can redistribute it and/or modify
  * it under the terms of the GNU General Public License as published by
  * the Free Software Foundation; either version 2 of the License, or
@@ -321,6 +325,11 @@ struct mem_cgroup {
 #ifdef CONFIG_INET
  struct tcp_memcontrol tcp_mem;
 #endif
+
+#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
+ /* Slab accounting */
+ struct kmem_cache *slabs[MAX_KMEM_CACHE_TYPES];
+#endif
 };

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

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

 /* Bitmap used for allocating the cache id numbers. */
 static DECLARE_BITMAP(cache_types, MAX_KMEM_CACHE_TYPES);
+static DEFINE_MUTEX(memcg_cache_mutex);
+
+static inline bool mem_cgroup_kmem_enabled(struct mem_cgroup *memcg)
```

```
+{
+ return !mem_cgroup_disabled() && memcg &&
+        !mem_cgroup_is_root(memcg) && memcg->kmem_accounted;
+}

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

```
+/* Use a single spinlock for destruction and creation, not a frequent op */
+static DEFINE_SPINLOCK(cache_queue_lock);
+static LIST_HEAD(create_queue);
+static LIST_HEAD(destroyed_caches);
+
+static void kmem_cache_destroy_work_func(struct work_struct *w)
+{
+ struct kmem_cache *cachep;
+ char *name;
+
+ spin_lock_irq(&cache_queue_lock);
+ while (!list_empty(&destroyed_caches)) {
+  cachep = container_of(list_first_entry(&destroyed_caches,
+      struct mem_cgroup_cache_params, destroyed_list), struct
+      kmem_cache, memcg_params);
+  name = (char *)cachep->name;
+  list_del(&cachep->memcg_params.destroyed_list);
+  spin_unlock_irq(&cache_queue_lock);
+  synchronize_rcu();
+  kmem_cache_destroy(cachep);
+  kfree(name);
+  spin_lock_irq(&cache_queue_lock);
+ }
+ spin_unlock_irq(&cache_queue_lock);
+}
+static DECLARE_WORK(kmem_cache_destroy_work, kmem_cache_destroy_work_func);
+
+void mem_cgroup_destroy_cache(struct kmem_cache *cachep)
+{
+ unsigned long flags;
+
+ BUG_ON(cachep->memcg_params.id != -1);
+
+ /*
+  * We have to defer the actual destroying to a workqueue, because
+  * we might currently be in a context that cannot sleep.
+  */
+ spin_lock_irqsave(&cache_queue_lock, flags);
+ list_add(&cachep->memcg_params.destroyed_list, &destroyed_caches);
+ spin_unlock_irqrestore(&cache_queue_lock, flags);
+
+ schedule_work(&kmem_cache_destroy_work);
+}
+
+
+/*
+ * Flush the queue of kmem_caches to create, because we're creating a cgroup.
+ *
```

```
+ * We might end up flushing other cgroups' creation requests as well, but
+ * they will just get queued again next time someone tries to make a slab
+ * allocation for them.
+ */
+void mem_cgroup_flush_cache_create_queue(void)
+{
+ struct create_work *cw, *tmp;
+ unsigned long flags;
+
+ spin_lock_irqsave(&cache_queue_lock, flags);
+ list_for_each_entry_safe(cw, tmp, &create_queue, list) {
+  list_del(&cw->list);
+  kfree(cw);
+ }
+ spin_unlock_irqrestore(&cache_queue_lock, flags);
+}
+
+static void memcg_create_cache_work_func(struct work_struct *w)
+{
+ struct kmem_cache *cachep;
+ struct create_work *cw;
+
+ spin_lock_irq(&cache_queue_lock);
+ while (!list_empty(&create_queue)) {
+  cw = list_first_entry(&create_queue, struct create_work, list);
+  list_del(&cw->list);
+  spin_unlock_irq(&cache_queue_lock);
+  cachep = memcg_create_kmem_cache(cw->memcg, cw->cachep);
+  if (cachep == NULL)
+   printk(KERN_ALERT
+   "%s: Couldn't create memcg-cache for %s memcg %s\n",
+   __func__, cw->cachep->name,
+   cw->memcg->css.cgroup->dentry->d_name.name);
+  /* Drop the reference gotten when we enqueued. */
+  css_put(&cw->memcg->css);
+  kfree(cw);
+  spin_lock_irq(&cache_queue_lock);
+ }
+ spin_unlock_irq(&cache_queue_lock);
+}
+
+static DECLARE_WORK(memcg_create_cache_work, memcg_create_cache_work_func);
+
+/*
+ * Enqueue the creation of a per-memcg kmem_cache.
+ * Called with rcu_read_lock.
+ */
+static void memcg_create_cache_enqueue(struct mem_cgroup *memcg,
```

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

```
+          gfp_t gfp)
+{
+ struct mem_cgroup *memcg;
+ int idx;
+
+ gfp |=  cachep->allocflags;
+
+ if ((current->mm == NULL))
+  return cachep;
+
+ if (cachep->memcg_params.memcg)
+  return cachep;
+
+ idx = cachep->memcg_params.id;
+ VM_BUG_ON(idx == -1);
+
+ memcg = mem_cgroup_from_task(current);
+ if (!mem_cgroup_kmem_enabled(memcg))
+  return cachep;
+
+ if (rcu_access_pointer(memcg->slabs[idx]) == NULL) {
+  memcg_create_cache_enqueue(memcg, cachep);
+  return cachep;
+ }
+
+ return rcu_dereference(memcg->slabs[idx]);
+}
+EXPORT_SYMBOL(__mem_cgroup_get_kmem_cache);
+
+void mem_cgroup_remove_child_kmem_cache(struct kmem_cache *cachep, int id)
+{
+ rcu_assign_pointer(cachep->memcg_params.memcg->slabs[id], NULL);
+}
+
+bool __mem_cgroup_charge_kmem(gfp_t gfp, size_t size)
+{
+ struct mem_cgroup *memcg;
+ bool ret = true;
+
+ rcu_read_lock();
+ memcg = mem_cgroup_from_task(current);
+
+ if (!mem_cgroup_kmem_enabled(memcg))
+  goto out;
+
+ mem_cgroup_get(memcg);
+ ret = memcg_charge_kmem(memcg, gfp, size) == 0;
+ if (ret)
```

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

```
+ if (!mem_cgroup_kmem_enabled(memcg)) {
+  rcu_read_unlock();
+  return;
+ }
+ rcu_read_unlock();
+
+ memcg_uncharge_kmem(memcg, size);
+}
+EXPORT_SYMBOL(__mem_cgroup_uncharge_slab);
+
+static void memcg_slab_init(struct mem_cgroup *memcg)
+{
+ int i;
+
+ for (i = 0; i < MAX_KMEM_CACHE_TYPES; i++)
+  rcu_assign_pointer(memcg->slabs[i], NULL);
+}
 #endif /* CONFIG_CGROUP_MEM_RES_CTLR_KMEM */

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

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

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

 #endif
+
+#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
+int memcg_charge_kmem(struct mem_cgroup *memcg, gfp_t gfp, s64 delta)
+{
+ struct res_counter *fail_res;
+ struct mem_cgroup *_memcg;
+ int may_oom, ret;
+ bool nofail = false;
+
+ may_oom = (gfp & __GFP_WAIT) && (gfp & __GFP_FS) &&
+     !(gfp & __GFP_NORETRY);
```

```
+
+ ret = 0;
+
+ if (!memcg)
+  return ret;
+
+ _memcg = memcg;
+ ret = __mem_cgroup_try_charge(NULL, gfp, delta / PAGE_SIZE,
+     &_memcg, may_oom);
+ if (ret == -ENOMEM)
+  return ret;
+ else if ((ret == -EINTR) || (ret && (gfp & __GFP_NOFAIL)))  {
+  nofail = true;
+  /*
+   * __mem_cgroup_try_charge() chose to bypass to root due
+   * to OOM kill or fatal signal.
+   * Since our only options are to either fail the
+   * allocation or charge it to this cgroup, force the
+   * change, going above the limit if needed.
+   */
+  res_counter_charge_nofail(&memcg->res, delta, &fail_res);
+ }
+
+ if (nofail)
+  res_counter_charge_nofail(&memcg->kmem, delta, &fail_res);
+ else
+  ret = res_counter_charge(&memcg->kmem, delta, &fail_res);
+
+ if (ret)
+  res_counter_uncharge(&memcg->res, delta);
+
+ return ret;
+}
+
+void memcg_uncharge_kmem(struct mem_cgroup *memcg, s64 delta)
+{
+ if (!memcg)
+  return;
+
+ res_counter_uncharge(&memcg->kmem, delta);
+ res_counter_uncharge(&memcg->res, delta);
+}
+#endif /* CONFIG_CGROUP_MEM_RES_CTLR_KMEM */
--
1.7.7.6
```