Subject: [PATCH v2 20/29] slub: charge allocation to a memcg
Posted by Glauber Costa on Fri, 11 May 2012 17:44:22 GMT
View Forum Message <> Reply to Message

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

The cache is selected with mem_cgroup_get_kmem_cache(),
which is the biggest overhead we pay here, because
it happens at all allocations. However, other than forcing
a function call, this function is not very expensive, and
try to return as soon as we realize we are not a memcg cache.

The charge/uncharge functions are heavier, but are only called
for new page allocations.

The kmalloc_no_account variant is patched so the base
function is used and we don't even try to do cache
selection.

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

```
---
 include/linux/slub_def.h |   39 ++++++++++++++--
 mm/slub.c                |  113 ++++++++++++++++++++++++++++++++++++++++++++++-----
 2 files changed, 135 insertions(+), 17 deletions(-)

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

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

 enum stat_item {
  ALLOC_FASTPATH,  /* Allocation from cpu slab */
@@ -210,27 +212,54 @@ static __always_inline int kmalloc_index(size_t size)
  * This ought to end up with a global pointer to the right cache
  * in kmalloc_caches.
```

```
 */
-static __always_inline struct kmem_cache *kmalloc_slab(size_t size)
+static __always_inline struct kmem_cache *kmalloc_slab(gfp_t flags, size_t size)
 {
+ struct kmem_cache *s;
  int index = kmalloc_index(size);

  if (index == 0)
   return NULL;

- return kmalloc_caches[index];
+ s = kmalloc_caches[index];
+
+ rcu_read_lock();
+ s = mem_cgroup_get_kmem_cache(s, flags);
+ rcu_read_unlock();
+
+ return s;
 }

 void *kmem_cache_alloc(struct kmem_cache *, gfp_t);
 void *__kmalloc(size_t size, gfp_t flags);

 static __always_inline void *
-kmalloc_order(size_t size, gfp_t flags, unsigned int order)
+kmalloc_order_base(size_t size, gfp_t flags, unsigned int order)
 {
  void *ret = (void *) __get_free_pages(flags | __GFP_COMP, order);
  kmemleak_alloc(ret, size, 1, flags);
  return ret;
 }

+static __always_inline void *
+kmalloc_order(size_t size, gfp_t flags, unsigned int order)
+{
+ void *ret = NULL;
+ struct page *page;
+
+ ret = kmalloc_order_base(size, flags, order);
+ if (!ret)
+  return ret;
+
+ page = virt_to_head_page(ret);
+
+ if (!mem_cgroup_new_kmem_page(page, flags)) {
+  put_page(page);
+  return NULL;
+ }
```

```
+
+ return ret;
+}
+
 /**
  * Calling this on allocated memory will check that the memory
  * is expected to be in use, and print warnings if not.
@@ -275,7 +304,7 @@ static __always_inline void *kmalloc(size_t size, gfp_t flags)
    return kmalloc_large(size, flags);

   if (!(flags & SLUB_DMA)) {
-   struct kmem_cache *s = kmalloc_slab(size);
+   struct kmem_cache *s = kmalloc_slab(flags, size);

    if (!s)
     return ZERO_SIZE_PTR;
@@ -308,7 +337,7 @@ static __always_inline void *kmalloc_node(size_t size, gfp_t flags, int
node)
 {
  if (__builtin_constant_p(size) &&
   size <= SLUB_MAX_SIZE && !(flags & SLUB_DMA)) {
-   struct kmem_cache *s = kmalloc_slab(size);
+   struct kmem_cache *s = kmalloc_slab(flags, size);

   if (!s)
    return ZERO_SIZE_PTR;
diff --git a/mm/slub.c b/mm/slub.c
index 9b21b38..c70db56 100644
--- a/mm/slub.c
+++ b/mm/slub.c
@@ -1284,11 +1284,39 @@ static inline struct page *alloc_slab_page(gfp_t flags, int node,
   return alloc_pages_exact_node(node, flags, order);
 }

+static inline unsigned long size_in_bytes(unsigned int order)
+{
+ return (1 << order) << PAGE_SHIFT;
+}
+
+#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
+static void kmem_cache_inc_ref(struct kmem_cache *s)
+{
+ if (s->memcg_params.memcg)
+  atomic_inc(&s->memcg_params.refcnt);
+}
+static void kmem_cache_drop_ref(struct kmem_cache *s)
+{
+ if (s->memcg_params.memcg)
```

```
+  atomic_dec(&s->memcg_params.refcnt);
+}
+#else
+static inline void kmem_cache_inc_ref(struct kmem_cache *s)
+{
+}
+static inline void kmem_cache_drop_ref(struct kmem_cache *s)
+{
+}
+#endif
+
+
+
 static struct page *allocate_slab(struct kmem_cache *s, gfp_t flags, int node)
 {
- struct page *page;
+ struct page *page = NULL;
  struct kmem_cache_order_objects oo = s->oo;
  gfp_t alloc_gfp;
+ unsigned int memcg_allowed = oo_order(oo);

  flags &= gfp_allowed_mask;

@@ -1297,13 +1325,29 @@ static struct page *allocate_slab(struct kmem_cache *s, gfp_t flags,
int node)

  flags |= s->allocflags;

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

```
+   * pressure so we fall-back to the minimum order allocation.
+   */
+  alloc_gfp = (flags | __GFP_NOWARN | __GFP_NORETRY) &
+        ~__GFP_NOFAIL;
+
+  page = alloc_slab_page(alloc_gfp, node, oo);
+ }

- page = alloc_slab_page(alloc_gfp, node, oo);
  if (unlikely(!page)) {
   oo = s->min;
   /*
@@ -1314,13 +1358,25 @@ static struct page *allocate_slab(struct kmem_cache *s, gfp_t flags,
int node)

   if (page)
    stat(s, ORDER_FALLBACK);
+  /*
+   * We reserved more than we used, time to give it back
+   */
+  if (page && memcg_allowed != oo_order(oo)) {
+   unsigned long delta;
+   delta = memcg_allowed - oo_order(oo);
+   mem_cgroup_uncharge_slab(s, size_in_bytes(delta));
+  }
  }

  if (flags & __GFP_WAIT)
   local_irq_disable();

- if (!page)
+ if (!page) {
+  mem_cgroup_uncharge_slab(s, size_in_bytes(memcg_allowed));
   return NULL;
+ }
+
+ kmem_cache_inc_ref(s);

  if (kmemcheck_enabled
   && !(s->flags & (SLAB_NOTRACK | DEBUG_DEFAULT_FLAGS))) {
@@ -1420,6 +1476,9 @@ static void __free_slab(struct kmem_cache *s, struct page *page)
  if (current->reclaim_state)
   current->reclaim_state->reclaimed_slab += pages;
  __free_pages(page, order);
+
+ mem_cgroup_uncharge_slab(s, (1 << order) << PAGE_SHIFT);
+ kmem_cache_drop_ref(s);
 }
```

```
 #define need_reserve_slab_rcu      \
@@ -2301,8 +2360,9 @@ new_slab:
  *
  * Otherwise we can simply pick the next object from the lockless free list.
  */
-static __always_inline void *slab_alloc(struct kmem_cache *s,
-  gfp_t gfpflags, int node, unsigned long addr)
+static __always_inline void *slab_alloc_base(struct kmem_cache *s,
+          gfp_t gfpflags, int node,
+          unsigned long addr)
 {
  void **object;
  struct kmem_cache_cpu *c;
@@ -2370,6 +2430,24 @@ redo:
  return object;
 }

+static __always_inline void *slab_alloc(struct kmem_cache *s,
+  gfp_t gfpflags, int node, unsigned long addr)
+{
+
+ if (slab_pre_alloc_hook(s, gfpflags))
+  return NULL;
+
+ if (in_interrupt() || (current == NULL) || (gfpflags & __GFP_NOFAIL))
+  goto kernel_alloc;
+
+ rcu_read_lock();
+ s = mem_cgroup_get_kmem_cache(s, gfpflags);
+ rcu_read_unlock();
+
+kernel_alloc:
+ return slab_alloc_base(s, gfpflags, node, addr);
+}
+
 void *kmem_cache_alloc(struct kmem_cache *s, gfp_t gfpflags)
 {
  void *ret = slab_alloc(s, gfpflags, NUMA_NO_NODE, _RET_IP_);
@@ -3197,8 +3275,10 @@ void kmem_cache_destroy(struct kmem_cache *s)
   list_del(&s->list);
 #ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
   /* Not a memcg cache */
-  if (s->memcg_params.id != -1)
+  if (s->memcg_params.id != -1) {
    mem_cgroup_release_cache(s);
+   mem_cgroup_flush_cache_create_queue();
+  }
```

```
 #endif
   up_write(&slub_lock);
   if (kmem_cache_close(s)) {
@@ -3372,10 +3452,18 @@ static void *kmalloc_large_node(size_t size, gfp_t flags, int node)
  void *ptr = NULL;

  flags |= __GFP_COMP | __GFP_NOTRACK;
+
+
  page = alloc_pages_node(node, flags, get_order(size));
- if (page)
+ if (!page)
+  goto out;
+
+ if (!mem_cgroup_new_kmem_page(page, flags))
+  put_page(page);
+ else
   ptr = page_address(page);

+out:
  kmemleak_alloc(ptr, size, 1, flags);
  return ptr;
 }
@@ -3477,6 +3565,7 @@ void kfree(const void *x)
  if (unlikely(!PageSlab(page))) {
   BUG_ON(!PageCompound(page));
   kmemleak_free(x);
+  mem_cgroup_free_kmem_page(page);
   put_page(page);
   return;
  }
--
1.7.7.6
```