Subject: [PATCH 0/7] memcg kernel memory tracking
Posted by Glauber Costa on Tue, 21 Feb 2012 11:34:32 GMT
View Forum Message <> Reply to Message

This is a first structured approach to tracking general kernel
memory within the memory controller. Please tell me what you think.

As previously proposed, one has the option of keeping kernel memory
accounted separatedly, or together with the normal userspace memory.
However, this time I made the option to, in this later case, bill
the memory directly to memcg->res. It has the disadvantage that it becomes
complicated to know which memory came from user or kernel, but OTOH,
it does not create any overhead of drawing from multiple res_counters
at read time. (and if you want them to be joined, you probably don't care)

Kernel memory is never tracked for the root memory cgroup. This means
that a system where no memory cgroups exists other than the root, the
time cost of this implementation is a couple of branches in the slub
code - none of them in fast paths. At the moment, this works only
with the slub.

At cgroup destruction, memory is billed to the parent. With no hierarchy,
this would mean the root memcg. But since we are not billing to that,
it simply ceases to be tracked.

The caches that we want to be tracked need to explicit register into
the infrastructure.

If you would like to give it a try, you'll need one of Frederic's patches
that is used as a basis for this
(cgroups: ability to stop res charge propagation on bounded ancestor)

Glauber Costa (7):
  small cleanup for memcontrol.c
  Basic kernel memory functionality for the Memory Controller
  per-cgroup slab caches
  chained slab caches: move pages to a different cache when a cache is
    destroyed.
  shrink support for memcg kmem controller
  track dcache per-memcg
  example shrinker for memcg-aware dcache

 fs/dcache.c | 136 ++++++++++++++++++-
 include/linux/dcache.h | 4 +
 include/linux/memcontrol.h | 35 +++++
 include/linux/shrinker.h | 4 +
 include/linux/slab.h | 12 ++
 include/linux/slub_def.h | 3 +

```
 mm/memcontrol.c          |  344 ++++++++++++++++++++++++++++++++++++++++++++++++-
 mm/slub.c                |  237 ++++++++++++++++++++++++++++++++++++++++++++++---
 mm/vmscan.c              |   60 ++++++++-
 9 files changed, 806 insertions(+), 29 deletions(-)

--
1.7.7.6
```

Subject: [PATCH 1/7] small cleanup for memcontrol.c
Posted by Glauber Costa on Tue, 21 Feb 2012 11:34:33 GMT
View Forum Message <> Reply to Message

Move some hardcoded definitions to an enum type.

Signed-off-by: Glauber Costa <glommer@parallels.com>
CC: Kirill A. Shutemov <kirill@shutemov.name>
CC: Greg Thelen <gthelen@google.com>
CC: Johannes Weiner <jweiner@redhat.com>
CC: Michal Hocko <mhocko@suse.cz>
CC: Hiroyouki Kamezawa <kamezawa.hiroyu@jp.fujitsu.com>
CC: Paul Turner <pjt@google.com>
CC: Frederic Weisbecker <fweisbec@gmail.com>
---
 mm/memcontrol.c | 10 +++++++---
 1 files changed, 7 insertions(+), 3 deletions(-)

```
diff --git a/mm/memcontrol.c b/mm/memcontrol.c
index 6728a7a..b15a693 100644
--- a/mm/memcontrol.c
+++ b/mm/memcontrol.c
@@ -351,9 +351,13 @@ enum charge_type {
 };

 /* for encoding cft->private value on file */
-#define _MEM   (0)
-#define _MEMSWAP  (1)
-#define _OOM_TYPE  (2)
+
+enum mem_type {
+ _MEM = 0,
+ _MEMSWAP,
+ _OOM_TYPE,
+};
+
 #define MEMFILE_PRIVATE(x, val) (((x) << 16) | (val))
 #define MEMFILE_TYPE(val) (((val) >> 16) & 0xffff)
 #define MEMFILE_ATTR(val) ((val) & 0xffff)
```

--
1.7.7.6

Subject: [PATCH 2/7] Basic kernel memory functionality for the Memory Controller
Posted by Glauber Costa on Tue, 21 Feb 2012 11:34:34 GMT
View Forum Message <> Reply to Message

This patch lays down the foundation for the kernel memory component
of the Memory Controller.

As of today, I am only laying down the following files:

 * memory.independent_kmem_limit
 * memory.kmem.limit_in_bytes
 * memory.kmem.soft_limit_in_bytes
 * memory.kmem.usage_in_bytes

I am omitting the Documentation files in this version, at least
in the first cycle. But they should not differ much from what
I posted previously. The patch itself is not much different
than the previous versions I posted.

Signed-off-by: Glauber Costa <glommer@parallels.com>
CC: Kirill A. Shutemov <kirill@shutemov.name>
CC: Greg Thelen <gthelen@google.com>
CC: Johannes Weiner <jweiner@redhat.com>
CC: Michal Hocko <mhocko@suse.cz>
CC: Hiroyouki Kamezawa <kamezawa.hiroyu@jp.fujitsu.com>
CC: Paul Turner <pjt@google.com>
CC: Frederic Weisbecker <fweisbec@gmail.com>
---
 mm/memcontrol.c | 98 ++-
 1 files changed, 97 insertions(+), 1 deletions(-)

diff --git a/mm/memcontrol.c b/mm/memcontrol.c
index b15a693..26fda11 100644
--- a/mm/memcontrol.c
+++ b/mm/memcontrol.c
@@ -235,6 +235,10 @@ struct mem_cgroup {
   */
  struct res_counter memsw;
  /*
+ * the counter to account for kmem usage.
+ */
+ struct res_counter kmem;
+ /*
   * Per cgroup active and inactive list, similar to the

```
   * per zone LRU lists.
   */
@@ -280,6 +284,11 @@ struct mem_cgroup {
   */
  unsigned long  move_charge_at_immigrate;
  /*
+  * Should kernel memory limits be stabilished independently
+  * from user memory ?
+  */
+ int  kmem_independent_accounting;
+ /*
   * percpu counter.
   */
  struct mem_cgroup_stat_cpu *stat;
@@ -356,6 +365,7 @@ enum mem_type {
  _MEM = 0,
  _MEMSWAP,
  _OOM_TYPE,
+ _KMEM,
 };

 #define MEMFILE_PRIVATE(x, val) (((x) << 16) | (val))
@@ -3844,6 +3854,11 @@ static u64 mem_cgroup_read(struct cgroup *cont, struct cftype *cft)
   else
    val = res_counter_read_u64(&memcg->memsw, name);
   break;
+#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
+ case _KMEM:
+  val = res_counter_read_u64(&memcg->kmem, name);
+  break;
+#endif
  default:
   BUG();
   break;
@@ -3876,7 +3891,13 @@ 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 == _KMEM) {
+   if (!memcg->kmem_independent_accounting) {
+    ret = -EINVAL;
+    break;
+   }
+   ret = res_counter_set_limit(&memcg->kmem, val);
+  } else
    ret = mem_cgroup_resize_memsw_limit(memcg, val);
   break;
```

```
  case RES_SOFT_LIMIT:
@@ -3890,6 +3911,16 @@ static int mem_cgroup_write(struct cgroup *cont, struct cftype *cft,
     */
    if (type == _MEM)
     ret = res_counter_set_soft_limit(&memcg->res, val);
+#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
+   else if (type == _KMEM) {
+    if (!memcg->kmem_independent_accounting) {
+     ret = -EINVAL;
+     break;
+    }
+    ret = res_counter_set_soft_limit(&memcg->kmem, val);
+    break;
+   }
+#endif
    else
     ret = -EINVAL;
    break;
@@ -4573,8 +4604,69 @@ 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 kmem_limit_independent_read(struct cgroup *cgroup, struct cftype *cft)
+{
+ return mem_cgroup_from_cont(cgroup)->kmem_independent_accounting;
+}
+
+static int kmem_limit_independent_write(struct cgroup *cgroup, struct cftype *cft,
+      u64 val)
+{
+ struct mem_cgroup *memcg = mem_cgroup_from_cont(cgroup);
+ struct mem_cgroup *parent = parent_mem_cgroup(memcg);
+
+ val = !!val;
+ /*
+  * This follows the same hierarchy restrictions than
+  * mem_cgroup_hierarchy_write().
+  *
+  * TODO: We also shouldn't allow cgroups
+  * with tasks in it to change this value. Otherwise it is impossible
+  * to track the kernel memory that is already in memcg->res.
+  */
+ if (!parent || !parent->use_hierarchy || mem_cgroup_is_root(parent)) {
+  if (list_empty(&cgroup->children))
+   memcg->kmem_independent_accounting = val;
+  else
+   return -EBUSY;
```

```
+ } else
+  return -EINVAL;
+
+ return 0;
+}
+static struct cftype kmem_cgroup_files[] = {
+ {
+  .name = "independent_kmem_limit",
+  .read_u64 = kmem_limit_independent_read,
+  .write_u64 = kmem_limit_independent_write,
+ },
+ {
+  .name = "kmem.usage_in_bytes",
+  .private = MEMFILE_PRIVATE(_KMEM, RES_USAGE),
+  .read_u64 = mem_cgroup_read,
+ },
+ {
+  .name = "kmem.limit_in_bytes",
+  .private = MEMFILE_PRIVATE(_KMEM, RES_LIMIT),
+  .read_u64 = mem_cgroup_read,
+  .write_string = mem_cgroup_write,
+ },
+ {
+  .name = "kmem.soft_limit_in_bytes",
+  .private = MEMFILE_PRIVATE(_KMEM, RES_SOFT_LIMIT),
+  .write_string = mem_cgroup_write,
+  .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.
@@ -4926,6 +5018,9 @@ mem_cgroup_create(struct cgroup_subsys *ss, struct cgroup *cont)
  if (parent && parent->use_hierarchy) {
   res_counter_init(&memcg->res, &parent->res);
   res_counter_init(&memcg->memsw, &parent->memsw);
+  res_counter_init(&memcg->kmem, &parent->kmem);
+  memcg->kmem_independent_accounting =
+     parent->kmem_independent_accounting;
```

```
   /*
    * We increment refcnt of the parent to ensure that we can
    * safely access it on res_counter_charge/uncharge.
@@ -4936,6 +5031,7 @@ mem_cgroup_create(struct cgroup_subsys *ss, struct cgroup *cont)
  } else {
   res_counter_init(&memcg->res, NULL);
   res_counter_init(&memcg->memsw, NULL);
+  res_counter_init(&memcg->kmem, NULL);
  }
  memcg->last_scanned_node = MAX_NUMNODES;
  INIT_LIST_HEAD(&memcg->oom_notify);
--
1.7.7.6
```

Subject: [PATCH 3/7] per-cgroup slab caches
Posted by Glauber Costa on Tue, 21 Feb 2012 11:34:35 GMT
View Forum Message <> Reply to Message

This patch creates the infrastructure to allow us to register
per-memcg slab caches. As an example implementation, I am tracking
the dentry cache, but others will follow.

I am using an opt-in istead of opt-out system here: this means the
cache needs to explicitly register itself to be tracked by memcg.
I prefer this approach since:

1) not all caches are big enough to be relevant,
2) most of the relevant ones will involve shrinking in some way,
   and it would be better be sure they are shrinker-aware
3) some objects, like network sockets, have their very own idea
   of memory control, that goes beyond the allocation itself.

Once registered, allocations made on behalf of a task living
on a cgroup will be billed to it. It is a first-touch mechanism,
but if follows what we have today, and the cgroup infrastructure
itself. No overhead is expected in object allocation: only when
slab pages are allocated and freed, any form of billing occurs.

The allocation stays billed to a cgroup until it is destroyed.
It is kept if a task leaves the cgroup, since it is close to
impossible to efficiently map an object to a task - and the tracking
is done by pages, which contain multiple objects.

Signed-off-by: Glauber Costa <glommer@parallels.com>
CC: Kirill A. Shutemov <kirill@shutemov.name>
CC: Greg Thelen <gthelen@google.com>
CC: Johannes Weiner <jweiner@redhat.com>

CC: Michal Hocko <mhocko@suse.cz>
CC: Hiroyouki Kamezawa <kamezawa.hiroyu@jp.fujitsu.com>
CC: Paul Turner <pjt@google.com>
CC: Frederic Weisbecker <fweisbec@gmail.com>
CC: Pekka Enberg <penberg@kernel.org>
CC: Christoph Lameter <cl@linux.com>
---
 include/linux/memcontrol.h | 29 +++++++++++++
 include/linux/slab.h | 8 ++++
 include/linux/slub_def.h | 2 +
 mm/memcontrol.c | 93 ++
 mm/slub.c | 68 +++++++++++++++++++++++++++++++--
 5 files changed, 195 insertions(+), 5 deletions(-)

diff --git a/include/linux/memcontrol.h b/include/linux/memcontrol.h
index 4d34356..95e7e19 100644
--- a/include/linux/memcontrol.h
+++ b/include/linux/memcontrol.h
@@ -21,12 +21,41 @@
 #define _LINUX_MEMCONTROL_H
 #include <linux/cgroup.h>
 #include <linux/vm_event_item.h>
+#include <linux/slab.h>
+#include <linux/workqueue.h>
+#include <linux/shrinker.h>

 struct mem_cgroup;
 struct page_cgroup;
 struct page;
 struct mm_struct;

+struct memcg_kmem_cache {
+ struct kmem_cache *cache;
+ struct mem_cgroup *memcg; /* Should be able to do without this */
+};
+
+struct memcg_cache_struct {
+ int index;
+ struct kmem_cache *cache;
+};
+
+enum memcg_cache_indexes {
+ CACHE_DENTRY,
+ NR_CACHES,
+};
+
+int memcg_kmem_newpage(struct mem_cgroup *memcg, struct page *page, unsigned long
pages);

```
+void memcg_kmem_freepage(struct mem_cgroup *memcg, struct page *page, unsigned long
pages);
+struct mem_cgroup *memcg_from_shrinker(struct shrinker *s);
+
+struct memcg_kmem_cache *memcg_cache_get(struct mem_cgroup *memcg, int index);
+void register_memcg_cache(struct memcg_cache_struct *cache);
+void memcg_slab_destroy(struct kmem_cache *cache, struct mem_cgroup *memcg);
+
+struct kmem_cache *
+kmem_cache_dup(struct mem_cgroup *memcg, struct kmem_cache *base);
+
 /* Stats that can be updated by kernel. */
 enum mem_cgroup_page_stat_item {
  MEMCG_NR_FILE_MAPPED, /* # of pages charged as file rss */
diff --git a/include/linux/slab.h b/include/linux/slab.h
index 573c809..8a372cd 100644
--- a/include/linux/slab.h
+++ b/include/linux/slab.h
@@ -98,6 +98,14 @@
 void __init kmem_cache_init(void);
 int slab_is_available(void);

+struct mem_cgroup;
+
+unsigned long slab_nr_pages(struct kmem_cache *s);
+
+struct kmem_cache *kmem_cache_create_cg(struct mem_cgroup *memcg,
+   const char *, size_t, size_t,
+   unsigned long,
+   void (*)(void *));
 struct kmem_cache *kmem_cache_create(const char *, size_t, size_t,
    unsigned long,
    void (*)(void *));
diff --git a/include/linux/slub_def.h b/include/linux/slub_def.h
index a32bcfd..4f39fff 100644
--- a/include/linux/slub_def.h
+++ b/include/linux/slub_def.h
@@ -100,6 +100,8 @@ struct kmem_cache {
  struct kobject kobj; /* For sysfs */
 #endif

+ struct mem_cgroup *memcg;
+ struct kmem_cache *parent_slab; /* can be moved out of here as well */
 #ifdef CONFIG_NUMA
  /*
   * Defragmentation by allocating from a remote node.
diff --git a/mm/memcontrol.c b/mm/memcontrol.c
index 26fda11..2aa35b0 100644
```

```
--- a/mm/memcontrol.c
+++ b/mm/memcontrol.c
@@ -49,6 +49,7 @@
 #include <linux/page_cgroup.h>
 #include <linux/cpu.h>
 #include <linux/oom.h>
+#include <linux/slab.h>
 #include "internal.h"
 #include <net/sock.h>
 #include <net/tcp_memcontrol.h>
@@ -302,8 +303,11 @@ struct mem_cgroup {
 #ifdef CONFIG_INET
  struct tcp_memcontrol tcp_mem;
 #endif
+ struct memcg_kmem_cache kmem_cache[NR_CACHES];
 };

+struct memcg_cache_struct *kmem_avail_caches[NR_CACHES];
+
 /* Stuffs for move charges at task migration. */
 /*
  * Types of charges to be moved. "move_charge_at_immitgrate" is treated as a
@@ -4980,6 +4984,93 @@ err_cleanup:

 }

+struct memcg_kmem_cache *memcg_cache_get(struct mem_cgroup *memcg, int index)
+{
+ return &memcg->kmem_cache[index];
+}
+
+void register_memcg_cache(struct memcg_cache_struct *cache)
+{
+ BUG_ON(kmem_avail_caches[cache->index]);
+
+ kmem_avail_caches[cache->index] = cache;
+}
+
+#define memcg_kmem(memcg) \
+ (memcg->kmem_independent_accounting ?  &memcg->kmem : &memcg->res)
+
+struct kmem_cache *
+kmem_cache_dup(struct mem_cgroup *memcg, struct kmem_cache *base)
+{
+ struct kmem_cache *s;
+ unsigned long pages;
+ struct res_counter *fail;
+ /*
```

```
+  * TODO: We should use an ida-like index here, instead
+  * of the kernel address
+  */
+ char *kname = kasprintf(GFP_KERNEL, "%s-%p", base->name, memcg);
+
+ WARN_ON(mem_cgroup_is_root(memcg));
+
+ if (!kname)
+  return NULL;
+
+ s = kmem_cache_create_cg(memcg, kname, base->size,
+      base->align, base->flags, base->ctor);
+ if (WARN_ON(!s))
+  goto out;
+
+
+ pages = slab_nr_pages(s);
+
+ if (res_counter_charge(memcg_kmem(memcg), pages << PAGE_SHIFT, &fail)) {
+  kmem_cache_destroy(s);
+  s = NULL;
+ }
+
+ mem_cgroup_get(memcg);
+out:
+ kfree(kname);
+ return s;
+}
+
+int memcg_kmem_newpage(struct mem_cgroup *memcg, struct page *page, unsigned long
pages)
+{
+ unsigned long size = pages << PAGE_SHIFT;
+ struct res_counter *fail;
+
+ return res_counter_charge(memcg_kmem(memcg), size, &fail);
+}
+
+void memcg_kmem_freepage(struct mem_cgroup *memcg, struct page *page, unsigned long
pages)
+{
+ unsigned long size = pages << PAGE_SHIFT;
+
+ res_counter_uncharge(memcg_kmem(memcg), size);
+}
+
+void memcg_create_kmem_caches(struct mem_cgroup *memcg)
+{
```

```
+ int i;
+
+ for (i = 0; i < NR_CACHES; i++) {
+  struct kmem_cache *cache;
+
+  if (!kmem_avail_caches[i] || !kmem_avail_caches[i]->cache)
+   continue;
+
+  cache = kmem_avail_caches[i]->cache;
+
+  if (mem_cgroup_is_root(memcg))
+   memcg->kmem_cache[i].cache = cache;
+  else
+   memcg->kmem_cache[i].cache = kmem_cache_dup(memcg, cache);
+  memcg->kmem_cache[i].memcg = memcg;
+ }
+}
+
+
 static struct cgroup_subsys_state * __ref
 mem_cgroup_create(struct cgroup_subsys *ss, struct cgroup *cont)
 {
@@ -5039,6 +5130,8 @@ mem_cgroup_create(struct cgroup_subsys *ss, struct cgroup *cont)
  if (parent)
   memcg->swappiness = mem_cgroup_swappiness(parent);
  atomic_set(&memcg->refcnt, 1);
+
+ memcg_create_kmem_caches(memcg);
  memcg->move_charge_at_immigrate = 0;
  mutex_init(&memcg->thresholds_lock);
  return &memcg->css;
diff --git a/mm/slub.c b/mm/slub.c
index 4907563..f3815ec 100644
--- a/mm/slub.c
+++ b/mm/slub.c
@@ -31,6 +31,8 @@
 #include <linux/stacktrace.h>

 #include <trace/events/kmem.h>
+#include <linux/cgroup.h>
+#include <linux/memcontrol.h>

 /*
  * Lock order:
@@ -1281,6 +1283,7 @@ static struct page *allocate_slab(struct kmem_cache *s, gfp_t flags, int
node)
 struct page *page;
 struct kmem_cache_order_objects oo = s->oo;
```

```
  gfp_t alloc_gfp;
+ int pages;

  flags &= gfp_allowed_mask;

@@ -1314,9 +1317,17 @@ static struct page *allocate_slab(struct kmem_cache *s, gfp_t flags,
int node)
  if (!page)
   return NULL;

+ pages = 1 << oo_order(oo);
+
+#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
+ if (s->memcg && memcg_kmem_newpage(s->memcg, page, pages) < 0) {
+  __free_pages(page, oo_order(oo));
+  return NULL;
+ }
+#endif
+
  if (kmemcheck_enabled
   && !(s->flags & (SLAB_NOTRACK | DEBUG_DEFAULT_FLAGS))) {
-  int pages = 1 << oo_order(oo);

   kmemcheck_alloc_shadow(page, oo_order(oo), flags, node);

@@ -1412,6 +1423,12 @@ 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);
+
+#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
+ if (s->memcg)
+  memcg_kmem_freepage(s->memcg, page, 1 << order);
+#endif
+
 }

 #define need_reserve_slab_rcu      \
@@ -3851,7 +3868,7 @@ static int slab_unmergeable(struct kmem_cache *s)
  return 0;
 }

-static struct kmem_cache *find_mergeable(size_t size,
+static struct kmem_cache *find_mergeable(struct mem_cgroup *memcg, size_t size,
   size_t align, unsigned long flags, const char *name,
   void (*ctor)(void *))
 {
@@ -3887,13 +3904,34 @@ static struct kmem_cache *find_mergeable(size_t size,
```

```
   if (s->size - size >= sizeof(void *))
    continue;

+  if (memcg && s->memcg != memcg)
+   continue;
+
   return s;
  }
  return NULL;
 }

-struct kmem_cache *kmem_cache_create(const char *name, size_t size,
-  size_t align, unsigned long flags, void (*ctor)(void *))
+unsigned long slab_nr_pages(struct kmem_cache *s)
+{
+ int node;
+ unsigned long nr_slabs = 0;
+
+        for_each_online_node(node) {
+                struct kmem_cache_node *n = get_node(s, node);
+
+                if (!n)
+                        continue;
+
+                nr_slabs += atomic_long_read(&n->nr_slabs);
+        }
+
+ return nr_slabs << oo_order(s->oo);
+}
+
+struct kmem_cache *
+kmem_cache_create_cg(struct mem_cgroup *memcg, const char *name, size_t size,
+       size_t align, unsigned long flags, void (*ctor)(void *))
 {
  struct kmem_cache *s;
  char *n;
@@ -3901,8 +3939,12 @@ struct kmem_cache *kmem_cache_create(const char *name, size_t
size,
  if (WARN_ON(!name))
   return NULL;

+#ifndef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
+ WARN_ON(memcg != NULL);
+#endif
+
  down_write(&slub_lock);
- s = find_mergeable(size, align, flags, name, ctor);
+ s = find_mergeable(memcg, size, align, flags, name, ctor);
```

```
  if (s) {
   s->refcount++;
   /*
@@ -3929,6 +3971,7 @@ struct kmem_cache *kmem_cache_create(const char *name, size_t
size,
   if (kmem_cache_open(s, n,
     size, align, flags, ctor)) {
    list_add(&s->list, &slab_caches);
+   s->memcg = memcg;
    if (sysfs_slab_add(s)) {
     list_del(&s->list);
     kfree(n);
@@ -3950,6 +3993,12 @@ err:
   s = NULL;
  return s;
 }
+
+struct kmem_cache *kmem_cache_create(const char *name, size_t size,
+  size_t align, unsigned long flags, void (*ctor)(void *))
+{
+ return kmem_cache_create_cg(NULL, name, size, align, flags, ctor);
+}
 EXPORT_SYMBOL(kmem_cache_create);

 #ifdef CONFIG_SMP
@@ -5239,6 +5288,15 @@ static char *create_unique_id(struct kmem_cache *s)
  if (p != name + 1)
   *p++ = '-';
  p += sprintf(p, "%07d", s->size);
+
+#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
+ /*
+  * TODO: We should use an ida-like index here, instead
+  * of the kernel address
+  */
+ if (s->memcg)
+  p += sprintf(p, "-%08x", (u32)((u64)(s->memcg)));
+#endif
  BUG_ON(p > name + ID_STR_LENGTH - 1);
  return name;
 }
--
1.7.7.6
```

Subject: [PATCH 4/7] chained slab caches: move pages to a different cache when a cache is destroyed.

Posted by Glauber Costa on Tue, 21 Feb 2012 11:34:36 GMT
View Forum Message <> Reply to Message

In the context of tracking kernel memory objects to a cgroup, the following problem appears: we may need to destroy a cgroup, but this does not guarantee that all objects inside the cache are dead. This can't be guaranteed even if we shrink the cache beforehand.

The simple option is to simply leave the cache around. However, intensive workloads may have generated a lot of objects and thus the dead cache will live in memory for a long while.

Scanning the list of objects in the dead cache takes time, and would probably require us to lock the free path of every objects to make sure we're not racing against the update.

I decided to give a try to a different idea then - but I'd be happy to pursue something else if you believe it would be better.

Upon memcg destruction, all the pages on the partial list are moved to the new slab (usually the parent memcg, or root memcg) When an object is freed, there are high stakes that no list locks are needed - so this case poses no overhead. If list manipulation is indeed needed, we can detect this case, and perform it in the right slab.

If all pages were residing in the partial list, we can free the cache right away. Otherwise, we do it when the last cache leaves the full list.

Signed-off-by: Glauber Costa <glommer@parallels.com>
CC: Kirill A. Shutemov <kirill@shutemov.name>
CC: Greg Thelen <gthelen@google.com>
CC: Johannes Weiner <jweiner@redhat.com>
CC: Michal Hocko <mhocko@suse.cz>
CC: Hiroyouki Kamezawa <kamezawa.hiroyu@jp.fujitsu.com>
CC: Paul Turner <pjt@google.com>
CC: Frederic Weisbecker <fweisbec@gmail.com>
CC: Pekka Enberg <penberg@kernel.org>
CC: Christoph Lameter <cl@linux.com>

```
---
 include/linux/memcontrol.h |    1 +
 include/linux/slab.h       |    4 +
 include/linux/slub_def.h   |    1 +
 mm/memcontrol.c            |   64 ++++++++++++++++-
 mm/slub.c                  |  171 +++++++++++++++++++++++++++++++++++++++++++----
 5 files changed, 227 insertions(+), 14 deletions(-)

diff --git a/include/linux/memcontrol.h b/include/linux/memcontrol.h
```

```
index 95e7e19..6138d10 100644
--- a/include/linux/memcontrol.h
+++ b/include/linux/memcontrol.h
@@ -32,6 +32,7 @@ struct mm_struct;

 struct memcg_kmem_cache {
  struct kmem_cache *cache;
+ struct work_struct destroy;
  struct mem_cgroup *memcg; /* Should be able to do without this */
 };

diff --git a/include/linux/slab.h b/include/linux/slab.h
index 8a372cd..c181927 100644
--- a/include/linux/slab.h
+++ b/include/linux/slab.h
@@ -79,6 +79,9 @@
 /* The following flags affect the page allocator grouping pages by mobility */
 #define SLAB_RECLAIM_ACCOUNT 0x00020000UL  /* Objects are reclaimable */
 #define SLAB_TEMPORARY  SLAB_RECLAIM_ACCOUNT /* Objects are short-lived */
+
+#define SLAB_CHAINED  0x04000000UL /* Slab is dead, but some objects still points
+         to it.*/
 /*
  * ZERO_SIZE_PTR will be returned for zero sized kmalloc requests.
  *
@@ -113,6 +116,7 @@ void kmem_cache_destroy(struct kmem_cache *);
 int kmem_cache_shrink(struct kmem_cache *);
 void kmem_cache_free(struct kmem_cache *, void *);
 unsigned int kmem_cache_size(struct kmem_cache *);
+void kmem_switch_slab(struct kmem_cache *old, struct kmem_cache *new);

 /*
  * Please use this macro to create slab caches. Simply specify the
diff --git a/include/linux/slub_def.h b/include/linux/slub_def.h
index 4f39fff..e20da0e 100644
--- a/include/linux/slub_def.h
+++ b/include/linux/slub_def.h
@@ -38,6 +38,7 @@ enum stat_item {
  CMPXCHG_DOUBLE_FAIL, /* Number of times that cmpxchg double did not match */
  CPU_PARTIAL_ALLOC, /* Used cpu partial on alloc */
  CPU_PARTIAL_FREE, /* USed cpu partial on free */
+ CPU_CHAINED_FREE, /* Chained to a parent slab during free*/
  NR_SLUB_STAT_ITEMS };

 struct kmem_cache_cpu {
diff --git a/mm/memcontrol.c b/mm/memcontrol.c
index 2aa35b0..1b1db88 100644
--- a/mm/memcontrol.c
```

```
+++ b/mm/memcontrol.c
@@ -4984,6 +4984,22 @@ err_cleanup:

 }

+static void __memcg_cache_destroy(struct memcg_kmem_cache *cache)
+{
+ if (!slab_nr_pages(cache->cache)) {
+  kmem_cache_destroy(cache->cache);
+  mem_cgroup_put(cache->memcg);
+ }
+}
+
+static void memcg_cache_destroy(struct work_struct *work)
+{
+ struct memcg_kmem_cache *cache;
+
+ cache = container_of(work, struct memcg_kmem_cache, destroy);
+ __memcg_cache_destroy(cache);
+}
+
 struct memcg_kmem_cache *memcg_cache_get(struct mem_cgroup *memcg, int index)
 {
  return &memcg->kmem_cache[index];
@@ -5066,6 +5082,7 @@ void memcg_create_kmem_caches(struct mem_cgroup *memcg)
    memcg->kmem_cache[i].cache = cache;
   else
    memcg->kmem_cache[i].cache = kmem_cache_dup(memcg, cache);
+  INIT_WORK(&memcg->kmem_cache[i].destroy, memcg_cache_destroy);
   memcg->kmem_cache[i].memcg = memcg;
  }
 }
@@ -5140,12 +5157,57 @@ free_out:
  return ERR_PTR(error);
 }

+void memcg_slab_destroy(struct kmem_cache *cache, struct mem_cgroup *memcg)
+{
+ int i;
+
+ for (i = 0; i < NR_CACHES; i++) {
+  if (memcg->kmem_cache[i].cache != cache)
+   continue;
+  schedule_work(&memcg->kmem_cache[i].destroy);
+ }
+}
+
 static int mem_cgroup_pre_destroy(struct cgroup_subsys *ss,
```

```
      struct cgroup *cont)
 {
  struct mem_cgroup *memcg = mem_cgroup_from_cont(cont);
+ struct mem_cgroup *parent = parent_mem_cgroup(memcg);
+ struct kmem_cache *newcache, *oldcache;
+ int i;
+ int ret;
+
+ if (WARN_ON(mem_cgroup_is_root(memcg)))
+  goto out;
+
+ for (i = 0; i < NR_CACHES; i++) {
+  unsigned long pages;
+
+  if (!memcg->kmem_cache[i].cache)
+   continue;
+
+  if (!parent)
+   parent = root_mem_cgroup;

- return mem_cgroup_force_empty(memcg, false);
+  if (parent->use_hierarchy)
+   newcache = parent->kmem_cache[i].cache;
+  else
+   newcache = root_mem_cgroup->kmem_cache[i].cache;
+
+
+  oldcache = memcg->kmem_cache[i].cache;
+  pages = slab_nr_pages(oldcache);
+
+  if (pages) {
+   kmem_switch_slab(oldcache, newcache);
+   res_counter_uncharge_until(memcg_kmem(memcg), memcg_kmem(memcg)->parent,
+         pages << PAGE_SHIFT);
+  }
+  __memcg_cache_destroy(&memcg->kmem_cache[i]);
+ }
+out:
+ ret = mem_cgroup_force_empty(memcg, false);
+ return ret;
 }

 static void mem_cgroup_destroy(struct cgroup_subsys *ss,
diff --git a/mm/slub.c b/mm/slub.c
index f3815ec..22851d7 100644
--- a/mm/slub.c
+++ b/mm/slub.c
@@ -1425,7 +1425,7 @@ static void __free_slab(struct kmem_cache *s, struct page *page)
```

```
  __free_pages(page, order);

 #ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
- if (s->memcg)
+ if (s->memcg && !(s->flags & SLAB_CHAINED))
   memcg_kmem_freepage(s->memcg, page, 1 << order);
 #endif

@@ -1471,7 +1471,10 @@ static void free_slab(struct kmem_cache *s, struct page *page)

 static void discard_slab(struct kmem_cache *s, struct page *page)
 {
- dec_slabs_node(s, page_to_nid(page), page->objects);
+ if (s->flags & SLAB_CHAINED)
+  dec_slabs_node(s->parent_slab, page_to_nid(page), page->objects);
+ else
+  dec_slabs_node(s, page_to_nid(page), page->objects);
  free_slab(s, page);
 }

@@ -2412,6 +2415,40 @@ EXPORT_SYMBOL(kmem_cache_alloc_node_trace);
 #endif
 #endif

+static void move_previous_full(struct kmem_cache *s, struct page *page)
+{
+ struct kmem_cache *target;
+ struct kmem_cache_node *n;
+ struct kmem_cache_node *tnode;
+ unsigned long uninitialized_var(flags);
+ int node = page_to_nid(page);
+
+ if (WARN_ON(s->flags & SLAB_STORE_USER))
+  return;
+
+ target = s->parent_slab;
+ if (WARN_ON(!target))
+  goto out;
+
+ n = get_node(s, node);
+ tnode = get_node(target, node);
+
+ dec_slabs_node(s, node, page->objects);
+ inc_slabs_node(target, node, page->objects);
+
+ page->slab = target;
+ add_partial(tnode, page, DEACTIVATE_TO_TAIL);
+out:
```

```
+ up_write(&slub_lock);
+}
+
+void destroy_chained_slab(struct kmem_cache *s, struct page *page)
+{
+ struct kmem_cache_node *n = get_node(s, page_to_nid(page));
+ if (atomic_long_read(&n->nr_slabs) == 0)
+  memcg_slab_destroy(s, s->memcg);
+}
+
 /*
  * Slow patch handling. This may still be called frequently since objects
  * have a longer lifetime than the cpu slabs in most processing loads.
@@ -2431,12 +2468,19 @@ static void __slab_free(struct kmem_cache *s, struct page *page,
  unsigned long counters;
  struct kmem_cache_node *n = NULL;
  unsigned long uninitialized_var(flags);
+ bool chained = s->flags & SLAB_CHAINED;

  stat(s, FREE_SLOWPATH);

  if (kmem_cache_debug(s) && !free_debug_processing(s, page, x, addr))
   return;

+ if (chained) {
+  /* Will only stall during the chaining, very unlikely */
+  do {} while (unlikely(!s->parent_slab));
+  stat(s->parent_slab, CPU_CHAINED_FREE);
+ }
+
  do {
   prior = page->freelist;
   counters = page->counters;
@@ -2455,8 +2499,10 @@ static void __slab_free(struct kmem_cache *s, struct page *page,
     new.frozen = 1;

    else { /* Needs to be taken off a list */
-
-                        n = get_node(s, page_to_nid(page));
+    if (chained)
+     n = get_node(s->parent_slab, page_to_nid(page));
+    else
+     n = get_node(s, page_to_nid(page));
     /*
      * Speculatively acquire the list_lock.
      * If the cmpxchg does not succeed then we may
@@ -2482,8 +2528,19 @@ static void __slab_free(struct kmem_cache *s, struct page *page,
    * If we just froze the page then put it onto the
```

```
     * per cpu partial list.
     */
-  if (new.frozen && !was_frozen)
-   put_cpu_partial(s, page, 1);
+  if (new.frozen && !was_frozen) {
+   if (!(chained))
+    put_cpu_partial(s, page, 1);
+   else  {
+    if (unlikely(page->slab == s)) {
+     n = get_node(s, page_to_nid(page));
+     spin_lock_irqsave(&n->list_lock, flags);
+     move_previous_full(s, page);
+     spin_unlock_irqrestore(&n->list_lock, flags);
+    } else
+     destroy_chained_slab(s, page);
+   }
+  }

    /*
     * The list lock was not taken therefore no list
@@ -2501,7 +2558,7 @@ static void __slab_free(struct kmem_cache *s, struct page *page,
   if (was_frozen)
    stat(s, FREE_FROZEN);
   else {
-   if (unlikely(!inuse && n->nr_partial > s->min_partial))
+   if (unlikely(!inuse && ((n->nr_partial > s->min_partial) || (chained))))
                        goto slab_empty;

    /*
@@ -2509,10 +2566,18 @@ static void __slab_free(struct kmem_cache *s, struct page *page,
     * then add it.
     */
    if (unlikely(!prior)) {
+    struct kmem_cache *target = s;
     remove_full(s, page);
-    add_partial(n, page, DEACTIVATE_TO_TAIL);
-    stat(s, FREE_ADD_PARTIAL);
+
+    if ((page->slab == s) && (chained))
+     move_previous_full(s, page);
+    else
+     add_partial(n, page, DEACTIVATE_TO_TAIL);
+    stat(target, FREE_ADD_PARTIAL);
    }
+
+   if (page->slab != s)
+    destroy_chained_slab(s, page);
   }
```

```
  spin_unlock_irqrestore(&n->list_lock, flags);
  return;
@@ -2522,13 +2587,26 @@ slab_empty:
    /*
     * Slab on the partial list.
     */
-  remove_partial(n, page);
-  stat(s, FREE_REMOVE_PARTIAL);
- } else
+  if (likely(!chained)) {
+   remove_partial(n, page);
+   stat(s, FREE_REMOVE_PARTIAL);
+  }
+ } else {
    /* Slab must be on the full list */
    remove_full(s, page);
+  /*
+   * In chaining, an empty slab can't be in the
+   * full list. We should have removed it when the first
+   * object was freed from it
+   */
+  WARN_ON((page->slab == s) && chained);
+ }

  spin_unlock_irqrestore(&n->list_lock, flags);
+
+ if (page->slab != s)
+  destroy_chained_slab(s, page);
+
  stat(s, FREE_SLAB);
  discard_slab(s, page);
 }
@@ -2577,8 +2655,11 @@ redo:
     goto redo;
    }
    stat(s, FREE_FASTPATH);
- } else
+ } else {
+  rcu_read_lock();
    __slab_free(s, page, x, addr);
+  rcu_read_unlock();
+ }

 }

@@ -3150,6 +3231,66 @@ static void free_partial(struct kmem_cache *s, struct
kmem_cache_node *n)
  }
```

```
 }

+#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
+void kmem_switch_slab(struct kmem_cache *old, struct kmem_cache *new)
+{
+ int node;
+ unsigned long flags;
+ struct page *page, *spage;
+
+ /*
+  * We need to make sure nobody is at slab_free's slow path.
+  * everybody before that will see the slab as non-chained.
+  *
+  * After the flag is set, we don't care if new freers appear.
+  * The node list lock should be able to protect any list operation
+  * being made. And if we're chained, we'll make sure we're grabbing
+  * the lock to the target cache.
+  *
+  * We also need to make sure nobody puts a page back to the per-cpu
+  * list.
+  */
+ synchronize_rcu();
+ old->flags |= SLAB_CHAINED;
+
+ flush_all(old);
+ for_each_online_node(node) {
+  struct kmem_cache_node *nnew = get_node(new, node);
+  struct kmem_cache_node *nold = get_node(old, node);
+
+  if (!nnew)
+   continue;
+
+  WARN_ON(!nold);
+  spin_lock_irqsave(&nnew->list_lock, flags);
+
+  list_for_each_entry_safe(page, spage, &nold->partial, lru) {
+   dec_slabs_node(old, node, page->objects);
+   page->slab = new;
+   inc_slabs_node(new, node, page->objects);
+   remove_partial(nold, page);
+   add_partial(nnew, page, DEACTIVATE_TO_TAIL);
+  }
+
+  if (!(old->flags & SLAB_STORE_USER)) {
+   spin_unlock_irqrestore(&nnew->list_lock, flags);
+   continue;
+  }
+
```

```
+  list_for_each_entry_safe(page, spage, &nold->full, lru) {
+   dec_slabs_node(old, node, page->objects);
+   page->slab = new;
+   inc_slabs_node(new, node, page->objects);
+   remove_full(old, page);
+   add_full(new, nnew, page);
+  }
+  spin_unlock_irqrestore(&nnew->list_lock, flags);
+
+ }
+ old->parent_slab = new;
+}
+#endif
+
 /*
  * Release all resources used by a slab cache.
  */
@@ -5108,6 +5249,7 @@ STAT_ATTR(CMPXCHG_DOUBLE_CPU_FAIL,
cmpxchg_double_cpu_fail);
 STAT_ATTR(CMPXCHG_DOUBLE_FAIL, cmpxchg_double_fail);
 STAT_ATTR(CPU_PARTIAL_ALLOC, cpu_partial_alloc);
 STAT_ATTR(CPU_PARTIAL_FREE, cpu_partial_free);
+STAT_ATTR(CPU_CHAINED_FREE, cpu_chained_free);
 #endif

 static struct attribute *slab_attrs[] = {
@@ -5173,6 +5315,7 @@ static struct attribute *slab_attrs[] = {
  &cmpxchg_double_cpu_fail_attr.attr,
  &cpu_partial_alloc_attr.attr,
  &cpu_partial_free_attr.attr,
+ &cpu_chained_free_attr.attr,
 #endif
 #ifdef CONFIG_FAILSLAB
  &failslab_attr.attr,
@@ -5487,6 +5630,8 @@ static int s_show(struct seq_file *m, void *p)
  int node;

  s = list_entry(p, struct kmem_cache, list);
+ if (s->flags & SLAB_CHAINED)
+  return 0;

  for_each_online_node(node) {
   struct kmem_cache_node *n = get_node(s, node);
--
1.7.7.6
```

Subject: [PATCH 5/7] shrink support for memcg kmem controller
Posted by Glauber Costa on Tue, 21 Feb 2012 11:34:37 GMT
View Forum Message <> Reply to Message

This patch adds the shrinker interface to memcg proposed kmem
controller. With this, softlimits starts being meaningful. I didn't
played to much with softlimits itself, since it is a bit in progress
for the general case as well. But this patch at least makes vmscan.c
no longer skip shrink_slab for the memcg case.

It also allows us to set the hard limit to a lower value than
current usage, as it is possible for the current memcg: a reclaim
is carried on, and if we succeed in freeing enough of kernel memory,
we can lower the limit.

Signed-off-by: Glauber Costa <glommer@parallels.com>
CC: Kirill A. Shutemov <kirill@shutemov.name>
CC: Greg Thelen <gthelen@google.com>
CC: Johannes Weiner <jweiner@redhat.com>
CC: Michal Hocko <mhocko@suse.cz>
CC: Hiroyouki Kamezawa <kamezawa.hiroyu@jp.fujitsu.com>
CC: Paul Turner <pjt@google.com>
CC: Frederic Weisbecker <fweisbec@gmail.com>
CC: Pekka Enberg <penberg@kernel.org>
CC: Christoph Lameter <cl@linux.com>
---
 include/linux/memcontrol.h | 5 +++
 include/linux/shrinker.h | 4 ++
 mm/memcontrol.c | 87 ++--
 mm/vmscan.c | 60 ++++++++++++++++++++++++++++++--
 4 files changed, 150 insertions(+), 6 deletions(-)

diff --git a/include/linux/memcontrol.h b/include/linux/memcontrol.h
index 6138d10..246b2d4 100644
--- a/include/linux/memcontrol.h
+++ b/include/linux/memcontrol.h
@@ -33,12 +33,16 @@ struct mm_struct;
 struct memcg_kmem_cache {
  struct kmem_cache *cache;
  struct work_struct destroy;
+ struct list_head lru;
+ u32 nr_objects;
  struct mem_cgroup *memcg; /* Should be able to do without this */
 };

 struct memcg_cache_struct {
  int index;
  struct kmem_cache *cache;
+ int (*shrink_fn)(struct shrinker *shrink, struct shrink_control *sc);

```
+ struct shrinker shrink;
 };

 enum memcg_cache_indexes {
@@ -53,6 +57,7 @@ struct mem_cgroup *memcg_from_shrinker(struct shrinker *s);
 struct memcg_kmem_cache *memcg_cache_get(struct mem_cgroup *memcg, int index);
 void register_memcg_cache(struct memcg_cache_struct *cache);
 void memcg_slab_destroy(struct kmem_cache *cache, struct mem_cgroup *memcg);
+bool memcg_slab_reclaim(struct mem_cgroup *memcg);

 struct kmem_cache *
 kmem_cache_dup(struct mem_cgroup *memcg, struct kmem_cache *base);
diff --git a/include/linux/shrinker.h b/include/linux/shrinker.h
index 07ceb97..11efdba 100644
--- a/include/linux/shrinker.h
+++ b/include/linux/shrinker.h
@@ -1,6 +1,7 @@
 #ifndef _LINUX_SHRINKER_H
 #define _LINUX_SHRINKER_H

+struct mem_cgroup;
 /*
  * This struct is used to pass information from page reclaim to the shrinkers.
  * We consolidate the values for easier extention later.
@@ -10,6 +11,7 @@ struct shrink_control {

  /* How many slab objects shrinker() should scan and try to reclaim */
  unsigned long nr_to_scan;
+ struct mem_cgroup *memcg;
 };

 /*
@@ -40,4 +42,6 @@ struct shrinker {
 #define DEFAULT_SEEKS 2 /* A good number if you don't know better. */
 extern void register_shrinker(struct shrinker *);
 extern void unregister_shrinker(struct shrinker *);
+
+extern void register_shrinker_memcg(struct shrinker *);
 #endif
diff --git a/mm/memcontrol.c b/mm/memcontrol.c
index 1b1db88..9c89a3c 100644
--- a/mm/memcontrol.c
+++ b/mm/memcontrol.c
@@ -3460,6 +3460,54 @@ static int mem_cgroup_resize_limit(struct mem_cgroup *memcg,
  return ret;
 }

+static int mem_cgroup_resize_kmem_limit(struct mem_cgroup *memcg,
```

```
+     unsigned long long val)
+{
+
+ int retry_count;
+ int ret = 0;
+ int children = mem_cgroup_count_children(memcg);
+ u64 curusage, oldusage;
+
+ struct shrink_control shrink = {
+  .gfp_mask = GFP_KERNEL,
+  .memcg = memcg,
+ };
+
+ /*
+  * For keeping hierarchical_reclaim simple, how long we should retry
+  * is depends on callers. We set our retry-count to be function
+  * of # of children which we should visit in this loop.
+  */
+ retry_count = MEM_CGROUP_RECLAIM_RETRIES * children;
+
+ oldusage = res_counter_read_u64(&memcg->kmem, RES_USAGE);
+
+ while (retry_count) {
+  if (signal_pending(current)) {
+   ret = -EINTR;
+   break;
+  }
+  mutex_lock(&set_limit_mutex);
+  ret = res_counter_set_limit(&memcg->kmem, val);
+  mutex_unlock(&set_limit_mutex);
+  if (!ret)
+   break;
+
+  shrink_slab(&shrink, 0, 0);
+
+  curusage = res_counter_read_u64(&memcg->kmem, RES_USAGE);
+
+  /* Usage is reduced ? */
+  if (curusage >= oldusage)
+   retry_count--;
+  else
+   oldusage = curusage;
+ }
+ return ret;
+
+}
+
 static int mem_cgroup_resize_memsw_limit(struct mem_cgroup *memcg,
```

```
      unsigned long long val)
 {
@@ -3895,13 +3943,17 @@ static int mem_cgroup_write(struct cgroup *cont, struct cftype *cft,
     break;
    if (type == _MEM)
     ret = mem_cgroup_resize_limit(memcg, val);
+#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
    else if (type == _KMEM) {
     if (!memcg->kmem_independent_accounting) {
      ret = -EINVAL;
      break;
     }
-    ret = res_counter_set_limit(&memcg->kmem, val);
-  } else
+
+    ret = mem_cgroup_resize_kmem_limit(memcg, val);
+  }
+#endif
+  else
     ret = mem_cgroup_resize_memsw_limit(memcg, val);
    break;
   case RES_SOFT_LIMIT:
@@ -5007,9 +5059,19 @@ struct memcg_kmem_cache *memcg_cache_get(struct mem_cgroup
*memcg, int index)

 void register_memcg_cache(struct memcg_cache_struct *cache)
 {
+ struct shrinker *shrink;
+
   BUG_ON(kmem_avail_caches[cache->index]);

   kmem_avail_caches[cache->index] = cache;
+ if (!kmem_avail_caches[cache->index]->shrink_fn)
+  return;
+
+ shrink = &kmem_avail_caches[cache->index]->shrink;
+ shrink->seeks = DEFAULT_SEEKS;
+ shrink->shrink = kmem_avail_caches[cache->index]->shrink_fn;
+ shrink->batch = 1024;
+ register_shrinker_memcg(shrink);
 }

 #define memcg_kmem(memcg) \
@@ -5055,8 +5117,21 @@ int memcg_kmem_newpage(struct mem_cgroup *memcg, struct
page *page, unsigned lon
 {
   unsigned long size = pages << PAGE_SHIFT;
   struct res_counter *fail;
```

```
+ int ret;
+ bool do_softlimit;
+
+ ret = res_counter_charge(memcg_kmem(memcg), size, &fail);
+ if (unlikely(mem_cgroup_event_ratelimit(memcg,
+      MEM_CGROUP_TARGET_THRESH))) {
+
+  do_softlimit = mem_cgroup_event_ratelimit(memcg,
+      MEM_CGROUP_TARGET_SOFTLIMIT);
+  mem_cgroup_threshold(memcg);
+  if (unlikely(do_softlimit))
+   mem_cgroup_update_tree(memcg, page);
+ }

- return res_counter_charge(memcg_kmem(memcg), size, &fail);
+ return ret;
 }

 void memcg_kmem_freepage(struct mem_cgroup *memcg, struct page *page, unsigned long
pages)
@@ -5083,6 +5158,7 @@ void memcg_create_kmem_caches(struct mem_cgroup *memcg)
   else
    memcg->kmem_cache[i].cache = kmem_cache_dup(memcg, cache);
   INIT_WORK(&memcg->kmem_cache[i].destroy, memcg_cache_destroy);
+  INIT_LIST_HEAD(&memcg->kmem_cache[i].lru);
   memcg->kmem_cache[i].memcg = memcg;
  }
 }
@@ -5157,6 +5233,11 @@ free_out:
  return ERR_PTR(error);
 }

+bool memcg_slab_reclaim(struct mem_cgroup *memcg)
+{
+ return !memcg->kmem_independent_accounting;
+}
+
 void memcg_slab_destroy(struct kmem_cache *cache, struct mem_cgroup *memcg)
 {
  int i;
diff --git a/mm/vmscan.c b/mm/vmscan.c
index c52b235..b9bceb6 100644
--- a/mm/vmscan.c
+++ b/mm/vmscan.c
@@ -159,6 +159,23 @@ long vm_total_pages; /* The total number of pages which the VM
controls */
 static LIST_HEAD(shrinker_list);
 static DECLARE_RWSEM(shrinker_rwsem);
```

```
+#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
+/*
+ * If we could guarantee the root mem cgroup will always exist, we could just
+ * use the normal shrinker_list, and assume that the root memcg is passed
+ * as a parameter. But we're not quite there yet. Because of that, the shinkers
+ * from the memcg case can be different from the normal shrinker for the same
+ * object. This is not the ideal situation but is a step towards that.
+ *
+ * Also, not all caches will have their memcg version (also likely to change),
+ * so scanning the whole list is a waste.
+ *
+ * I am using, however, the same lock for both lists. Updates to it should
+ * be unfrequent, so I don't expect that to generate contention
+ */
+static LIST_HEAD(shrinker_memcg_list);
+#endif
+
 #ifdef CONFIG_CGROUP_MEM_RES_CTLR
 static bool global_reclaim(struct scan_control *sc)
 {
@@ -169,6 +186,11 @@ static bool scanning_global_lru(struct mem_cgroup_zone *mz)
 {
  return !mz->mem_cgroup;
 }
+
+static bool global_slab_reclaim(struct scan_control *sc)
+{
+ return !memcg_slab_reclaim(sc->target_mem_cgroup);
+}
 #else
 static bool global_reclaim(struct scan_control *sc)
 {
@@ -179,6 +201,11 @@ static bool scanning_global_lru(struct mem_cgroup_zone *mz)
 {
  return true;
 }
+
+static bool global_slab_reclaim(struct scan_control *sc)
+{
+ return true;
+}
 #endif

 static struct zone_reclaim_stat *get_reclaim_stat(struct mem_cgroup_zone *mz)
@@ -225,6 +252,16 @@ void unregister_shrinker(struct shrinker *shrinker)
 }
 EXPORT_SYMBOL(unregister_shrinker);
```

```
+#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
+void register_shrinker_memcg(struct shrinker *shrinker)
+{
+ atomic_long_set(&shrinker->nr_in_batch, 0);
+ down_write(&shrinker_rwsem);
+ list_add_tail(&shrinker->list, &shrinker_memcg_list);
+ up_write(&shrinker_rwsem);
+}
+#endif
+
 static inline int do_shrinker_shrink(struct shrinker *shrinker,
         struct shrink_control *sc,
         unsigned long nr_to_scan)
@@ -234,6 +271,18 @@ static inline int do_shrinker_shrink(struct shrinker *shrinker,
 }

 #define SHRINK_BATCH 128
+
+static inline struct list_head
+*get_shrinker_list(struct shrink_control *shrink)
+{
+#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
+ if (shrink->memcg)
+  return &shrinker_memcg_list;
+ else
+#endif
+  return &shrinker_list;
+}
+
 /*
  * Call the shrink functions to age shrinkable caches
  *
@@ -259,6 +308,9 @@ unsigned long shrink_slab(struct shrink_control *shrink,
 {
  struct shrinker *shrinker;
  unsigned long ret = 0;
+ struct list_head *slist;
+
+ slist = get_shrinker_list(shrink);

  if (nr_pages_scanned == 0)
   nr_pages_scanned = SWAP_CLUSTER_MAX;
@@ -269,7 +321,7 @@ unsigned long shrink_slab(struct shrink_control *shrink,
   goto out;
  }

- list_for_each_entry(shrinker, &shrinker_list, list) {
```

```
+ list_for_each_entry(shrinker, slist, list) {
  unsigned long long delta;
  long total_scan;
  long max_pass;
@@ -2351,9 +2403,9 @@ static unsigned long do_try_to_free_pages(struct zonelist *zonelist,

  /*
   * Don't shrink slabs when reclaiming memory from
-  * over limit cgroups
+  * over limit cgroups, if kernel memory is controlled independently
   */
- if (global_reclaim(sc)) {
+ if (!global_slab_reclaim(sc)) {
   unsigned long lru_pages = 0;
   for_each_zone_zonelist(zone, z, zonelist,
     gfp_zone(sc->gfp_mask)) {
@@ -2362,8 +2414,10 @@ static unsigned long do_try_to_free_pages(struct zonelist *zonelist,

    lru_pages += zone_reclaimable_pages(zone);
   }
+  shrink->memcg = sc->target_mem_cgroup;

   shrink_slab(shrink, sc->nr_scanned, lru_pages);
+
   if (reclaim_state) {
    sc->nr_reclaimed += reclaim_state->reclaimed_slab;
    reclaim_state->reclaimed_slab = 0;
--
1.7.7.6
```

Subject: [PATCH 6/7] track dcache per-memcg
Posted by Glauber Costa on Tue, 21 Feb 2012 11:34:38 GMT
View Forum Message <> Reply to Message

This patch allows to track kernel memory used by dentry caches
in the memory controller. It uses the infrastructure already laid
down, and register the dcache as the first users of it.

A new cache is created for that purpose, and new allocations
coming from tasks belonging to that cgroup will be serviced from
the new cache.

Signed-off-by: Glauber Costa <glommer@parallels.com>
CC: Kirill A. Shutemov <kirill@shutemov.name>
CC: Greg Thelen <gthelen@google.com>
CC: Johannes Weiner <jweiner@redhat.com>
CC: Michal Hocko <mhocko@suse.cz>

```
CC: Hiroyouki Kamezawa <kamezawa.hiroyu@jp.fujitsu.com>
CC: Paul Turner <pjt@google.com>
CC: Frederic Weisbecker <fweisbec@gmail.com>
CC: Dave Chinner <david@fromorbit.com>
---
 fs/dcache.c            |   38 ++++++++++++++++++++++++++++++++++-----
 include/linux/dcache.h |    3 +++
 2 files changed, 36 insertions(+), 5 deletions(-)

diff --git a/fs/dcache.c b/fs/dcache.c
index 16a53cc..a452c19 100644
--- a/fs/dcache.c
+++ b/fs/dcache.c
@@ -86,7 +86,7 @@ __cacheline_aligned_in_smp DEFINE_SEQLOCK(rename_lock);

 EXPORT_SYMBOL(rename_lock);

-static struct kmem_cache *dentry_cache __read_mostly;
+static struct memcg_kmem_cache dentry_cache __read_mostly;

 /*
  * This is the single most critical data structure when it comes
@@ -144,7 +144,7 @@ static void __d_free(struct rcu_head *head)
  WARN_ON(!list_empty(&dentry->d_alias));
  if (dname_external(dentry))
   kfree(dentry->d_name.name);
- kmem_cache_free(dentry_cache, dentry);
+ kmem_cache_free(dentry->d_cache->cache, dentry);
 }

 /*
@@ -234,6 +234,7 @@ static void dentry_lru_add(struct dentry *dentry)
  if (list_empty(&dentry->d_lru)) {
   spin_lock(&dcache_lru_lock);
   list_add(&dentry->d_lru, &dentry->d_sb->s_dentry_lru);
+  dentry->d_cache->nr_objects++;
   dentry->d_sb->s_nr_dentry_unused++;
   dentry_stat.nr_unused++;
   spin_unlock(&dcache_lru_lock);
@@ -1178,6 +1179,21 @@ void shrink_dcache_parent(struct dentry * parent)
 }
 EXPORT_SYMBOL(shrink_dcache_parent);

+static struct memcg_kmem_cache *dcache_pick_cache(void)
+{
+ struct mem_cgroup *memcg;
+ struct memcg_kmem_cache *kmem = &dentry_cache;
+
```

```
+ rcu_read_lock();
+ memcg = mem_cgroup_from_task(current);
+ rcu_read_unlock();
+
+ if (memcg)
+  kmem = memcg_cache_get(memcg, CACHE_DENTRY);
+
+ return kmem;
+}
+
 /**
  * __d_alloc - allocate a dcache entry
  * @sb: filesystem it will belong to
@@ -1192,15 +1208,18 @@ struct dentry *__d_alloc(struct super_block *sb, const struct qstr
*name)
 {
  struct dentry *dentry;
  char *dname;
+ struct memcg_kmem_cache *cache;
+
+ cache = dcache_pick_cache();

- dentry = kmem_cache_alloc(dentry_cache, GFP_KERNEL);
+ dentry = kmem_cache_alloc(cache->cache, GFP_KERNEL);
  if (!dentry)
   return NULL;

  if (name->len > DNAME_INLINE_LEN-1) {
   dname = kmalloc(name->len + 1, GFP_KERNEL);
   if (!dname) {
-   kmem_cache_free(dentry_cache, dentry);
+   kmem_cache_free(cache->cache, dentry);
    return NULL;
   }
  } else  {
@@ -1222,6 +1241,7 @@ struct dentry *__d_alloc(struct super_block *sb, const struct qstr
*name)
  dentry->d_sb = sb;
  dentry->d_op = NULL;
  dentry->d_fsdata = NULL;
+ dentry->d_cache = cache;
  INIT_HLIST_BL_NODE(&dentry->d_hash);
  INIT_LIST_HEAD(&dentry->d_lru);
  INIT_LIST_HEAD(&dentry->d_subdirs);
@@ -2990,6 +3010,11 @@ static void __init dcache_init_early(void)
   INIT_HLIST_BL_HEAD(dentry_hashtable + loop);
 }
```

```
+struct memcg_cache_struct memcg_dcache = {
+ .index = CACHE_DENTRY,
+ .shrink_fn = dcache_shrink_memcg,
+};
+
 static void __init dcache_init(void)
 {
  int loop;
@@ -2999,7 +3024,7 @@ 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,
+ dentry_cache.cache = KMEM_CACHE(dentry,
   SLAB_RECLAIM_ACCOUNT|SLAB_PANIC|SLAB_MEM_SPREAD);

  /* Hash may have been set up in dcache_init_early */
@@ -3018,6 +3043,9 @@ static void __init dcache_init(void)

  for (loop = 0; loop < (1 << d_hash_shift); loop++)
   INIT_HLIST_BL_HEAD(dentry_hashtable + loop);
+
+ memcg_dcache.cache = dentry_cache.cache;
+ register_memcg_cache(&memcg_dcache);
 }

 /* SLAB cache for __getname() consumers */
diff --git a/include/linux/dcache.h b/include/linux/dcache.h
index d64a55b..4d94b657 100644
--- a/include/linux/dcache.h
+++ b/include/linux/dcache.h
@@ -113,6 +113,8 @@ full_name_hash(const unsigned char *name, unsigned int len)
 # endif
 #endif

+struct mem_cgroup;
+
 struct dentry {
  /* RCU lookup touched fields */
  unsigned int d_flags;  /* protected by d_lock */
@@ -142,6 +144,7 @@ struct dentry {
  } d_u;
  struct list_head d_subdirs; /* our children */
  struct list_head d_alias; /* inode alias list */
+ struct memcg_kmem_cache *d_cache;
 };

 /*
```

--
1.7.7.6

Subject: [PATCH 7/7] example shrinker for memcg-aware dcache
Posted by Glauber Costa on Tue, 21 Feb 2012 11:34:39 GMT
View Forum Message <> Reply to Message

This is an example of a memcg-aware shrinker for the dcache.
In the way it works today, it tries to not interfere much with
the existing shrinker. We can probably make a list of sb's per-memcg -
having the root memcg to hold them all, but this adds a lot
of complications along the way.

This version is simpler, and serves to pave the way to future
work.

Signed-off-by: Glauber Costa <glommer@parallels.com>
CC: Kirill A. Shutemov <kirill@shutemov.name>
CC: Greg Thelen <gthelen@google.com>
CC: Johannes Weiner <jweiner@redhat.com>
CC: Michal Hocko <mhocko@suse.cz>
CC: Hiroyouki Kamezawa <kamezawa.hiroyu@jp.fujitsu.com>
CC: Paul Turner <pjt@google.com>
CC: Frederic Weisbecker <fweisbec@gmail.com>
CC: Dave Chinner <david@fromorbit.com>

```
---
 fs/dcache.c            |   98 +++++++++++++++++++++++++++++++++++++++++++++++++
 include/linux/dcache.h |    1 +
 2 files changed, 99 insertions(+), 0 deletions(-)

diff --git a/fs/dcache.c b/fs/dcache.c
index a452c19..98a0490 100644
--- a/fs/dcache.c
+++ b/fs/dcache.c
@@ -234,6 +234,7 @@ static void dentry_lru_add(struct dentry *dentry)
  if (list_empty(&dentry->d_lru)) {
   spin_lock(&dcache_lru_lock);
   list_add(&dentry->d_lru, &dentry->d_sb->s_dentry_lru);
+  list_add(&dentry->d_memcg_lru, &dentry->d_cache->lru);
   dentry->d_cache->nr_objects++;
   dentry->d_sb->s_nr_dentry_unused++;
   dentry_stat.nr_unused++;
@@ -243,9 +244,12 @@ static void dentry_lru_add(struct dentry *dentry)

 static void __dentry_lru_del(struct dentry *dentry)
 {
+ WARN_ON(list_empty(&dentry->d_memcg_lru));
```

```
  list_del_init(&dentry->d_lru);
+ list_del_init(&dentry->d_memcg_lru);
  dentry->d_flags &= ~DCACHE_SHRINK_LIST;
  dentry->d_sb->s_nr_dentry_unused--;
+ dentry->d_cache->nr_objects--;
  dentry_stat.nr_unused--;
 }

@@ -283,10 +287,13 @@ static void dentry_lru_move_list(struct dentry *dentry, struct list_head
*list)
  spin_lock(&dcache_lru_lock);
  if (list_empty(&dentry->d_lru)) {
   list_add_tail(&dentry->d_lru, list);
+  list_add_tail(&dentry->d_memcg_lru, &dentry->d_cache->lru);
   dentry->d_sb->s_nr_dentry_unused++;
+  dentry->d_cache->nr_objects++;
   dentry_stat.nr_unused++;
  } else {
   list_move_tail(&dentry->d_lru, list);
+  list_move_tail(&dentry->d_memcg_lru, &dentry->d_cache->lru);
  }
  spin_unlock(&dcache_lru_lock);
 }
@@ -771,6 +778,96 @@ static void shrink_dentry_list(struct list_head *list)
  rcu_read_unlock();
 }

+static void dcache_shrink_dentry_list(struct list_head *list)
+{
+ struct dentry *dentry;
+
+ rcu_read_lock();
+ for (;;) {
+  dentry = list_entry_rcu(list->prev, struct dentry, d_memcg_lru);
+  if (&dentry->d_memcg_lru == list)
+   break; /* empty */
+
+  spin_lock(&dentry->d_lock);
+
+  WARN_ON(dentry->d_cache == &dentry_cache);
+
+  if (dentry != list_entry(list->prev, struct dentry, d_memcg_lru)) {
+   spin_unlock(&dentry->d_lock);
+   continue;
+  }
+
+  /*
+   * We found an inuse dentry which was not removed from
```

```
+   * the LRU because of laziness during lookup.  Do not free
+   * it - just keep it off the LRU list.
+   */
+  if (dentry->d_count) {
+   dentry_lru_del(dentry);
+   spin_unlock(&dentry->d_lock);
+   continue;
+  }
+
+  rcu_read_unlock();
+
+  try_prune_one_dentry(dentry);
+
+  rcu_read_lock();
+ }
+ rcu_read_unlock();
+}
+
+static int dcache_shrink_memcg(struct shrinker *shrink, struct shrink_control *sc)
+{
+ struct memcg_kmem_cache *kcache;
+ int count = sc->nr_to_scan;
+ struct dentry *dentry;
+ struct mem_cgroup *memcg;
+ LIST_HEAD(referenced);
+ LIST_HEAD(tmp);
+
+ memcg = sc->memcg;
+ kcache = memcg_cache_get(memcg, CACHE_DENTRY);
+ if (!count) {
+  return kcache->nr_objects;
+ }
+relock:
+ spin_lock(&dcache_lru_lock);
+ while (!list_empty(&kcache->lru)) {
+
+  dentry = list_entry(kcache->lru.prev, struct dentry, d_memcg_lru);
+
+  BUG_ON(dentry->d_cache != kcache);
+
+  if (!spin_trylock(&dentry->d_lock)) {
+   spin_unlock(&dcache_lru_lock);
+   cpu_relax();
+   goto relock;
+  }
+
+  if (dentry->d_flags & DCACHE_REFERENCED) {
+   dentry->d_flags &= ~DCACHE_REFERENCED;
```

```
+   list_move(&dentry->d_memcg_lru, &referenced);
+   spin_unlock(&dentry->d_lock);
+  } else {
+   list_move_tail(&dentry->d_memcg_lru, &tmp);
+   spin_unlock(&dentry->d_lock);
+   if (!--count)
+    break;
+  }
+
+  cond_resched_lock(&dcache_lru_lock);
+ }
+ if (!list_empty(&referenced))
+  list_splice(&referenced, &kcache->lru);
+
+ spin_unlock(&dcache_lru_lock);
+
+ dcache_shrink_dentry_list(&tmp);
+
+ return sc->nr_to_scan - count;
+}
+
 /**
  * prune_dcache_sb - shrink the dcache
  * @sb: superblock
@@ -1244,6 +1341,7 @@ struct dentry *__d_alloc(struct super_block *sb, const struct qstr
*name)
  dentry->d_cache = cache;
  INIT_HLIST_BL_NODE(&dentry->d_hash);
  INIT_LIST_HEAD(&dentry->d_lru);
+ INIT_LIST_HEAD(&dentry->d_memcg_lru);
  INIT_LIST_HEAD(&dentry->d_subdirs);
  INIT_LIST_HEAD(&dentry->d_alias);
  INIT_LIST_HEAD(&dentry->d_u.d_child);
diff --git a/include/linux/dcache.h b/include/linux/dcache.h
index 4d94b657..4af7be3 100644
--- a/include/linux/dcache.h
+++ b/include/linux/dcache.h
@@ -135,6 +135,7 @@ struct dentry {
  void *d_fsdata;   /* fs-specific data */

  struct list_head d_lru;  /* LRU list */
+ struct list_head d_memcg_lru; /* per-memcg LRU list */
  /*
   * d_child and d_rcu can share memory
   */
--
1.7.7.6
```

Subject: Re: [PATCH 0/7] memcg kernel memory tracking
Posted by Suleiman Souhlal on Tue, 21 Feb 2012 23:25:18 GMT
View Forum Message <> Reply to Message

Hi Glauber,

On Tue, Feb 21, 2012 at 3:34 AM, Glauber Costa <glommer@parallels.com> wrote:
> This is a first structured approach to tracking general kernel
> memory within the memory controller. Please tell me what you think.

Thanks for posting these.

> As previously proposed, one has the option of keeping kernel memory
> accounted separatedly, or together with the normal userspace memory.
> However, this time I made the option to, in this later case, bill
> the memory directly to memcg->res. It has the disadvantage that it becomes
> complicated to know which memory came from user or kernel, but OTOH,
> it does not create any overhead of drawing from multiple res_counters
> at read time. (and if you want them to be joined, you probably don't care)

It would be nice to still keep a kernel memory counter (that gets
updated at the same time as memcg->res) even when the limits are not
independent, because sometimes it's important to know how much kernel
memory is being used by a cgroup.

> Kernel memory is never tracked for the root memory cgroup. This means
> that a system where no memory cgroups exists other than the root, the
> time cost of this implementation is a couple of branches in the slub
> code - none of them in fast paths. At the moment, this works only
> with the slub.
>
> At cgroup destruction, memory is billed to the parent. With no hierarchy,
> this would mean the root memcg. But since we are not billing to that,
> it simply ceases to be tracked.
>
> The caches that we want to be tracked need to explicit register into
> the infrastructure.

Why not track every cache unless otherwise specified? If you don't,
you might end up polluting code all around the kernel to create
per-cgroup caches.
>From what I've seen, there are a fair amount of different caches that
can end up using a significant amount of memory, and having to go
around and explicitly mark each one doesn't seem like the right thing
to do.

-- Suleiman

Subject: Re: [PATCH 5/7] shrink support for memcg kmem controller
Posted by Suleiman Souhlal on Tue, 21 Feb 2012 23:35:10 GMT
View Forum Message <> Reply to Message

On Tue, Feb 21, 2012 at 3:34 AM, Glauber Costa <glommer@parallels.com> wrote:

> @@ -5055,8 +5117,21 @@ int memcg_kmem_newpage(struct mem_cgroup *memcg, struct page *page, unsigned lon
> {
> unsigned long size = pages << PAGE_SHIFT;
> struct res_counter *fail;
> + int ret;
> + bool do_softlimit;
> +
> + ret = res_counter_charge(memcg_kmem(memcg), size, &fail);
> + if (unlikely(mem_cgroup_event_ratelimit(memcg,
> + MEM_CGROUP_TARGET_THRESH))) {
> +
> + do_softlimit = mem_cgroup_event_ratelimit(memcg,
> + MEM_CGROUP_TARGET_SOFTLIMIT);
> + mem_cgroup_threshold(memcg);
> + if (unlikely(do_softlimit))
> + mem_cgroup_update_tree(memcg, page);
> + }
>
> - return res_counter_charge(memcg_kmem(memcg), size, &fail);
> + return ret;
> }

It seems like this might cause a lot of kernel memory allocations to fail whenever we are at the limit, even if we have a lot of reclaimable memory, when we don't have independent accounting.

Would it be better to use __mem_cgroup_try_charge() here, when we don't have independent accounting, in order to deal with this situation?

-- Suleiman

Subject: Re: [PATCH 4/7] chained slab caches: move pages to a different cache when a cache is destroyed.
Posted by Suleiman Souhlal on Tue, 21 Feb 2012 23:40:46 GMT
View Forum Message <> Reply to Message

On Tue, Feb 21, 2012 at 3:34 AM, Glauber Costa <glommer@parallels.com> wrote:

> In the context of tracking kernel memory objects to a cgroup, the
> following problem appears: we may need to destroy a cgroup, but
> this does not guarantee that all objects inside the cache are dead.
> This can't be guaranteed even if we shrink the cache beforehand.
>
> The simple option is to simply leave the cache around. However,
> intensive workloads may have generated a lot of objects and thus
> the dead cache will live in memory for a long while.

Why is this a problem?

Leaving the cache around while there are still active objects in it
would certainly be a lot simpler to understand and implement.

-- Suleiman

Subject: Re: [PATCH 3/7] per-cgroup slab caches
Posted by Suleiman Souhlal on Tue, 21 Feb 2012 23:50:32 GMT
View Forum Message <> Reply to Message

On Tue, Feb 21, 2012 at 3:34 AM, Glauber Costa <glommer@parallels.com> wrote:
> diff --git a/mm/memcontrol.c b/mm/memcontrol.c
> index 26fda11..2aa35b0 100644
> --- a/mm/memcontrol.c
> +++ b/mm/memcontrol.c
> +struct kmem_cache *
> +kmem_cache_dup(struct mem_cgroup *memcg, struct kmem_cache *base)
> +{
> +	struct kmem_cache *s;
> +	unsigned long pages;
> +	struct res_counter *fail;
> +	/*
> +	 * TODO: We should use an ida-like index here, instead
> +	 * of the kernel address
> +	 */
> +	char *kname = kasprintf(GFP_KERNEL, "%s-%p", base->name, memcg);

Would it make more sense to use the memcg name instead of the pointer?

> +
> +	WARN_ON(mem_cgroup_is_root(memcg));
> +
> +	if (!kname)
> +		return NULL;
> +
> +	s = kmem_cache_create_cg(memcg, kname, base->size,
> +				 base->align, base->flags, base->ctor);

```
> +       if (WARN_ON(!s))
> +               goto out;
> +
> +
> +       pages = slab_nr_pages(s);
> +
> +       if (res_counter_charge(memcg_kmem(memcg), pages << PAGE_SHIFT, &fail)) {
> +               kmem_cache_destroy(s);
> +               s = NULL;
> +       }
```

What are we charging here? Does it ever get uncharged?

-- Suleiman

Subject: Re: [PATCH 1/7] small cleanup for memcontrol.c
Posted by KAMEZAWA Hiroyuki on Wed, 22 Feb 2012 00:46:19 GMT
View Forum Message <> Reply to Message

On Tue, 21 Feb 2012 15:34:33 +0400
Glauber Costa <glommer@parallels.com> wrote:

> Move some hardcoded definitions to an enum type.
>
> Signed-off-by: Glauber Costa <glommer@parallels.com>
> CC: Kirill A. Shutemov <kirill@shutemov.name>
> CC: Greg Thelen <gthelen@google.com>
> CC: Johannes Weiner <jweiner@redhat.com>
> CC: Michal Hocko <mhocko@suse.cz>
> CC: Hiroyouki Kamezawa <kamezawa.hiroyu@jp.fujitsu.com>
> CC: Paul Turner <pjt@google.com>
> CC: Frederic Weisbecker <fweisbec@gmail.com>

seems ok to me.

Acked-by: KAMEZAWA Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com>

a nitpick..

```
> ---
>  mm/memcontrol.c |   10 +++++++---
>  1 files changed, 7 insertions(+), 3 deletions(-)
>
> diff --git a/mm/memcontrol.c b/mm/memcontrol.c
> index 6728a7a..b15a693 100644
> --- a/mm/memcontrol.c
> +++ b/mm/memcontrol.c
```

> @@ -351,9 +351,13 @@ enum charge_type {
> };
>
> /* for encoding cft->private value on file */
> -#define _MEM   (0)
> -#define _MEMSWAP (1)
> -#define _OOM_TYPE (2)
> +
> +enum mem_type {
> + _MEM = 0,

=0 is required ?

> + _MEMSWAP,
> + _OOM_TYPE,
> +};
> +
> #define MEMFILE_PRIVATE(x, val) (((x) << 16) | (val))
> #define MEMFILE_TYPE(val) (((val) >> 16) & 0xffff)
> #define MEMFILE_ATTR(val) ((val) & 0xffff)
> --
> 1.7.7.6
>
>

Subject: Re: [PATCH 3/7] per-cgroup slab caches
Posted by KAMEZAWA Hiroyuki on Wed, 22 Feb 2012 01:21:38 GMT
View Forum Message <> Reply to Message

On Tue, 21 Feb 2012 15:34:35 +0400
Glauber Costa <glommer@parallels.com> wrote:

> This patch creates the infrastructure to allow us to register
> per-memcg slab caches. As an example implementation, I am tracking
> the dentry cache, but others will follow.
>
> I am using an opt-in istead of opt-out system here: this means the
> cache needs to explicitly register itself to be tracked by memcg.
> I prefer this approach since:
>
> 1) not all caches are big enough to be relevant,
> 2) most of the relevant ones will involve shrinking in some way,
>   and it would be better be sure they are shrinker-aware
> 3) some objects, like network sockets, have their very own idea
>   of memory control, that goes beyond the allocation itself.
>
> Once registered, allocations made on behalf of a task living

> on a cgroup will be billed to it. It is a first-touch mechanism,
> but if follows what we have today, and the cgroup infrastructure
> itself. No overhead is expected in object allocation: only when
> slab pages are allocated and freed, any form of billing occurs.
>
> The allocation stays billed to a cgroup until it is destroyed.
> It is kept if a task leaves the cgroup, since it is close to
> impossible to efficiently map an object to a task - and the tracking
> is done by pages, which contain multiple objects.
>

Hmm....can't we do this by

 kmem_cache = kmem_cache_create(...., ..., SLAB_XXX_XXX | SLAB_MEMCG_AWARE)

 kmem_cache_alloc(kmem_cache, flags)
 => find a memcg_kmem_cache for the thread. if it doesn't exist, create it.

BTW, comparing ANON/FILE caches, we'll have many kinds of gfp_t flags.

Do we handle it correctly ?

Maybe I don't fully understand your implemenatation...but try to comment.

> Signed-off-by: Glauber Costa <glommer@parallels.com>
> CC: Kirill A. Shutemov <kirill@shutemov.name>
> CC: Greg Thelen <gthelen@google.com>
> CC: Johannes Weiner <jweiner@redhat.com>
> CC: Michal Hocko <mhocko@suse.cz>
> CC: Hiroyouki Kamezawa <kamezawa.hiroyu@jp.fujitsu.com>
> CC: Paul Turner <pjt@google.com>
> CC: Frederic Weisbecker <fweisbec@gmail.com>
> CC: Pekka Enberg <penberg@kernel.org>
> CC: Christoph Lameter <cl@linux.com>
> ---
> include/linux/memcontrol.h | 29 +++++++++++++
> include/linux/slab.h | 8 ++++
> include/linux/slub_def.h | 2 +
> mm/memcontrol.c | 93 ++
> mm/slub.c | 68 ++++++++++++++++++++++++++++++++--
> 5 files changed, 195 insertions(+), 5 deletions(-)
>
> diff --git a/include/linux/memcontrol.h b/include/linux/memcontrol.h
> index 4d34356..95e7e19 100644
> --- a/include/linux/memcontrol.h
> +++ b/include/linux/memcontrol.h

> @@ -21,12 +21,41 @@
> #define _LINUX_MEMCONTROL_H
> #include <linux/cgroup.h>
> #include <linux/vm_event_item.h>
> +#include <linux/slab.h>
> +#include <linux/workqueue.h>
> +#include <linux/shrinker.h>
>
> struct mem_cgroup;
> struct page_cgroup;
> struct page;
> struct mm_struct;
>
> +struct memcg_kmem_cache {
> + struct kmem_cache *cache;
> + struct mem_cgroup *memcg; /* Should be able to do without this */
> +};
> +
> +struct memcg_cache_struct {
> + int index;
> + struct kmem_cache *cache;
> +};
> +
> +enum memcg_cache_indexes {
> + CACHE_DENTRY,
> + NR_CACHES,
> +};
> +
> +int memcg_kmem_newpage(struct mem_cgroup *memcg, struct page *page, unsigned long
pages);
> +void memcg_kmem_freepage(struct mem_cgroup *memcg, struct page *page, unsigned long
pages);
> +struct mem_cgroup *memcg_from_shrinker(struct shrinker *s);
> +
> +struct memcg_kmem_cache *memcg_cache_get(struct mem_cgroup *memcg, int index);
> +void register_memcg_cache(struct memcg_cache_struct *cache);
> +void memcg_slab_destroy(struct kmem_cache *cache, struct mem_cgroup *memcg);
> +
> +struct kmem_cache *
> +kmem_cache_dup(struct mem_cgroup *memcg, struct kmem_cache *base);
> +
> /* Stats that can be updated by kernel. */
> enum mem_cgroup_page_stat_item {
> MEMCG_NR_FILE_MAPPED, /* # of pages charged as file rss */
> diff --git a/include/linux/slab.h b/include/linux/slab.h
> index 573c809..8a372cd 100644
> --- a/include/linux/slab.h
> +++ b/include/linux/slab.h

> @@ -98,6 +98,14 @@
> void __init kmem_cache_init(void);
> int slab_is_available(void);
>
> +struct mem_cgroup;
> +
> +unsigned long slab_nr_pages(struct kmem_cache *s);
> +
> +struct kmem_cache *kmem_cache_create_cg(struct mem_cgroup *memcg,
> +   const char *, size_t, size_t,
> +   unsigned long,
> +   void (*)(void *));
> struct kmem_cache *kmem_cache_create(const char *, size_t, size_t,
>   unsigned long,
>   void (*)(void *));
> diff --git a/include/linux/slub_def.h b/include/linux/slub_def.h
> index a32bcfd..4f39fff 100644
> --- a/include/linux/slub_def.h
> +++ b/include/linux/slub_def.h
> @@ -100,6 +100,8 @@ struct kmem_cache {
>   struct kobject kobj; /* For sysfs */
> #endif
>
> + struct mem_cgroup *memcg;
> + struct kmem_cache *parent_slab; /* can be moved out of here as well */
> #ifdef CONFIG_NUMA
>   /*
>   * Defragmentation by allocating from a remote node.
> diff --git a/mm/memcontrol.c b/mm/memcontrol.c
> index 26fda11..2aa35b0 100644
> --- a/mm/memcontrol.c
> +++ b/mm/memcontrol.c
> @@ -49,6 +49,7 @@
> #include <linux/page_cgroup.h>
> #include <linux/cpu.h>
> #include <linux/oom.h>
> +#include <linux/slab.h>
> #include "internal.h"
> #include <net/sock.h>
> #include <net/tcp_memcontrol.h>
> @@ -302,8 +303,11 @@ struct mem_cgroup {
> #ifdef CONFIG_INET
>   struct tcp_memcontrol tcp_mem;
> #endif
> + struct memcg_kmem_cache kmem_cache[NR_CACHES];
> };
>
> +struct memcg_cache_struct *kmem_avail_caches[NR_CACHES];

> +

What this pointer array holds ? This can be accessed without any locks ?

> /* Stuffs for move charges at task migration. */
> /*
>   * Types of charges to be moved. "move_charge_at_immitgrate" is treated as a
> @@ -4980,6 +4984,93 @@ err_cleanup:
>
> }
>
> +struct memcg_kmem_cache *memcg_cache_get(struct mem_cgroup *memcg, int index)
> +{
> + return &memcg->kmem_cache[index];
> +}
> +
> +void register_memcg_cache(struct memcg_cache_struct *cache)
> +{
> + BUG_ON(kmem_avail_caches[cache->index]);
> +
> + kmem_avail_caches[cache->index] = cache;
> +}
> +
> +#define memcg_kmem(memcg) \
> + (memcg->kmem_independent_accounting ? &memcg->kmem : &memcg->res)
> +
> +struct kmem_cache *
> +kmem_cache_dup(struct mem_cgroup *memcg, struct kmem_cache *base)
> +{
> + struct kmem_cache *s;
> + unsigned long pages;
> + struct res_counter *fail;
> + /*
> + * TODO: We should use an ida-like index here, instead
> + * of the kernel address
> + */
> + char *kname = kasprintf(GFP_KERNEL, "%s-%p", base->name, memcg);
> +
> + WARN_ON(mem_cgroup_is_root(memcg));
> +
> + if (!kname)
> + return NULL;
> +
> + s = kmem_cache_create_cg(memcg, kname, base->size,
> +   base->align, base->flags, base->ctor);
> + if (WARN_ON(!s))
> + goto out;

> +
> +
> + pages = slab_nr_pages(s);
> +
> + if (res_counter_charge(memcg_kmem(memcg), pages << PAGE_SHIFT, &fail)) {
> + kmem_cache_destroy(s);
> + s = NULL;
> + }

Why 'pages' should be charged to a new memcg ?
A newly created memcg starts with res_counter.usage != 0 ??

> +
> + mem_cgroup_get(memcg);

get even if allocation failure ? (and for what updating refcnt ?)

> +out:
> + kfree(kname);
> + return s;
> +}

> +
> +int memcg_kmem_newpage(struct mem_cgroup *memcg, struct page *page, unsigned long pages)
> +{
> + unsigned long size = pages << PAGE_SHIFT;
> + struct res_counter *fail;
> +
> + return res_counter_charge(memcg_kmem(memcg), size, &fail);
> +}
> +
> +void memcg_kmem_freepage(struct mem_cgroup *memcg, struct page *page, unsigned long pages)
> +{
> + unsigned long size = pages << PAGE_SHIFT;
> +
> + res_counter_uncharge(memcg_kmem(memcg), size);
> +}
> +
> +void memcg_create_kmem_caches(struct mem_cgroup *memcg)
> +{
> + int i;
> +
> + for (i = 0; i < NR_CACHES; i++) {
> + struct kmem_cache *cache;

> +
> + if (!kmem_avail_caches[i] || !kmem_avail_caches[i]->cache)
> +   continue;
> +
> + cache = kmem_avail_caches[i]->cache;
> +
> + if (mem_cgroup_is_root(memcg))
> +   memcg->kmem_cache[i].cache = cache;
> + else
> +   memcg->kmem_cache[i].cache = kmem_cache_dup(memcg, cache);
> + memcg->kmem_cache[i].memcg = memcg;
> + }
> +}

Hmm... memcg should know _all_ kmem_caches when it's created. Right ?
Then, modules will not use memcg aware kmem_cache...

> +
> +
> static struct cgroup_subsys_state * __ref
> mem_cgroup_create(struct cgroup_subsys *ss, struct cgroup *cont)
> {
> @@ -5039,6 +5130,8 @@ mem_cgroup_create(struct cgroup_subsys *ss, struct cgroup *cont)
>   if (parent)
>   memcg->swappiness = mem_cgroup_swappiness(parent);
>   atomic_set(&memcg->refcnt, 1);
> +
> + memcg_create_kmem_caches(memcg);
>   memcg->move_charge_at_immigrate = 0;
>   mutex_init(&memcg->thresholds_lock);
>   return &memcg->css;
> diff --git a/mm/slub.c b/mm/slub.c
> index 4907563..f3815ec 100644
> --- a/mm/slub.c
> +++ b/mm/slub.c
> @@ -31,6 +31,8 @@
> #include <linux/stacktrace.h>
>
> #include <trace/events/kmem.h>
> +#include <linux/cgroup.h>
> +#include <linux/memcontrol.h>
>
> /*
>   * Lock order:
> @@ -1281,6 +1283,7 @@ static struct page *allocate_slab(struct kmem_cache *s, gfp_t flags,

int node)
>   struct page *page;
>   struct kmem_cache_order_objects oo = s->oo;
>   gfp_t alloc_gfp;
> + int pages;
>
>   flags &= gfp_allowed_mask;
>
> @@ -1314,9 +1317,17 @@ static struct page *allocate_slab(struct kmem_cache *s, gfp_t flags,
int node)
>   if (!page)
>   return NULL;
>
> + pages = 1 << oo_order(oo);
> +
> +#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
> + if (s->memcg && memcg_kmem_newpage(s->memcg, page, pages) < 0) {
> + __free_pages(page, oo_order(oo));
> + return NULL;
> + }
> +#endif

Hmm. no reclaim happens at allocation failure ?
Doesn't it turn to be very bad user experience ?

Thanks,
-Kame

Subject: Re: [PATCH 4/7] chained slab caches: move pages to a different cache when a cache is destroyed.
Posted by KAMEZAWA Hiroyuki on Wed, 22 Feb 2012 01:25:12 GMT
View Forum Message <> Reply to Message

On Tue, 21 Feb 2012 15:34:36 +0400
Glauber Costa <glommer@parallels.com> wrote:

> In the context of tracking kernel memory objects to a cgroup, the
> following problem appears: we may need to destroy a cgroup, but
> this does not guarantee that all objects inside the cache are dead.
> This can't be guaranteed even if we shrink the cache beforehand.
>
> The simple option is to simply leave the cache around. However,
> intensive workloads may have generated a lot of objects and thus
> the dead cache will live in memory for a long while.
>
> Scanning the list of objects in the dead cache takes time, and

> would probably require us to lock the free path of every objects
> to make sure we're not racing against the update.
>
> I decided to give a try to a different idea then - but I'd be
> happy to pursue something else if you believe it would be better.
>
> Upon memcg destruction, all the pages on the partial list
> are moved to the new slab (usually the parent memcg, or root memcg)
> When an object is freed, there are high stakes that no list locks
> are needed - so this case poses no overhead. If list manipulation
> is indeed needed, we can detect this case, and perform it
> in the right slab.
>
> If all pages were residing in the partial list, we can free
> the cache right away. Otherwise, we do it when the last cache
> leaves the full list.
>

How about starting from 'don't handle slabs on dead memcg'
if shrink_slab() can find them....

This "move" complicates all implementation, I think...

Thanks,
-Kame

Subject: Re: [PATCH 5/7] shrink support for memcg kmem controller
Posted by KAMEZAWA Hiroyuki on Wed, 22 Feb 2012 01:42:56 GMT
View Forum Message <> Reply to Message

On Tue, 21 Feb 2012 15:34:37 +0400
Glauber Costa <glommer@parallels.com> wrote:

> This patch adds the shrinker interface to memcg proposed kmem
> controller. With this, softlimits starts being meaningful. I didn't
> played to much with softlimits itself, since it is a bit in progress
> for the general case as well. But this patch at least makes vmscan.c
> no longer skip shrink_slab for the memcg case.
>
> It also allows us to set the hard limit to a lower value than
> current usage, as it is possible for the current memcg: a reclaim
> is carried on, and if we succeed in freeing enough of kernel memory,
> we can lower the limit.
>
> Signed-off-by: Glauber Costa <glommer@parallels.com>
> CC: Kirill A. Shutemov <kirill@shutemov.name>
> CC: Greg Thelen <gthelen@google.com>

> CC: Johannes Weiner <jweiner@redhat.com>
> CC: Michal Hocko <mhocko@suse.cz>
> CC: Hiroyouki Kamezawa <kamezawa.hiroyu@jp.fujitsu.com>
> CC: Paul Turner <pjt@google.com>
> CC: Frederic Weisbecker <fweisbec@gmail.com>
> CC: Pekka Enberg <penberg@kernel.org>
> CC: Christoph Lameter <cl@linux.com>
> ---
> include/linux/memcontrol.h | 5 +++
> include/linux/shrinker.h | 4 ++
> mm/memcontrol.c | 87 ++--
> mm/vmscan.c | 60 ++++++++++++++++++++++++++++--
> 4 files changed, 150 insertions(+), 6 deletions(-)
>
> diff --git a/include/linux/memcontrol.h b/include/linux/memcontrol.h
> index 6138d10..246b2d4 100644
> --- a/include/linux/memcontrol.h
> +++ b/include/linux/memcontrol.h
> @@ -33,12 +33,16 @@ struct mm_struct;
> struct memcg_kmem_cache {
>   struct kmem_cache *cache;
>   struct work_struct destroy;
> + struct list_head lru;
> + u32 nr_objects;
>   struct mem_cgroup *memcg; /* Should be able to do without this */
> };
>
> struct memcg_cache_struct {
>   int index;
>   struct kmem_cache *cache;
> + int (*shrink_fn)(struct shrinker *shrink, struct shrink_control *sc);
> + struct shrinker shrink;
> };
>
> enum memcg_cache_indexes {
> @@ -53,6 +57,7 @@ struct mem_cgroup *memcg_from_shrinker(struct shrinker *s);
> struct memcg_kmem_cache *memcg_cache_get(struct mem_cgroup *memcg, int index);
> void register_memcg_cache(struct memcg_cache_struct *cache);
> void memcg_slab_destroy(struct kmem_cache *cache, struct mem_cgroup *memcg);
> +bool memcg_slab_reclaim(struct mem_cgroup *memcg);
>
> struct kmem_cache *
> kmem_cache_dup(struct mem_cgroup *memcg, struct kmem_cache *base);
> diff --git a/include/linux/shrinker.h b/include/linux/shrinker.h
> index 07ceb97..11efdba 100644
> --- a/include/linux/shrinker.h
> +++ b/include/linux/shrinker.h
> @@ -1,6 +1,7 @@

```
>  #ifndef _LINUX_SHRINKER_H
>  #define _LINUX_SHRINKER_H
>
> +struct mem_cgroup;
>  /*
>   * This struct is used to pass information from page reclaim to the shrinkers.
>   * We consolidate the values for easier extention later.
> @@ -10,6 +11,7 @@ struct shrink_control {
>
>   /* How many slab objects shrinker() should scan and try to reclaim */
>   unsigned long nr_to_scan;
> + struct mem_cgroup *memcg;
>  };
>
>  /*
> @@ -40,4 +42,6 @@ struct shrinker {
>  #define DEFAULT_SEEKS 2 /* A good number if you don't know better. */
>  extern void register_shrinker(struct shrinker *);
>  extern void unregister_shrinker(struct shrinker *);
> +
> +extern void register_shrinker_memcg(struct shrinker *);
>  #endif
> diff --git a/mm/memcontrol.c b/mm/memcontrol.c
> index 1b1db88..9c89a3c 100644
> --- a/mm/memcontrol.c
> +++ b/mm/memcontrol.c
> @@ -3460,6 +3460,54 @@ static int mem_cgroup_resize_limit(struct mem_cgroup *memcg,
>   return ret;
>  }
>
> +static int mem_cgroup_resize_kmem_limit(struct mem_cgroup *memcg,
> +     unsigned long long val)
> +{
> +
> + int retry_count;
> + int ret = 0;
> + int children = mem_cgroup_count_children(memcg);
> + u64 curusage, oldusage;
> +
> + struct shrink_control shrink = {
> +  .gfp_mask = GFP_KERNEL,
> +  .memcg = memcg,
> + };
> +
> + /*
> +  * For keeping hierarchical_reclaim simple, how long we should retry
> +  * is depends on callers. We set our retry-count to be function
> +  * of # of children which we should visit in this loop.
```

```
> +  */
> + retry_count = MEM_CGROUP_RECLAIM_RETRIES * children;
> +
> + oldusage = res_counter_read_u64(&memcg->kmem, RES_USAGE);
> +
> + while (retry_count) {
> +  if (signal_pending(current)) {
> +   ret = -EINTR;
> +   break;
> +  }
> +  mutex_lock(&set_limit_mutex);
> +  ret = res_counter_set_limit(&memcg->kmem, val);
> +  mutex_unlock(&set_limit_mutex);
> +  if (!ret)
> +   break;
> +
> +  shrink_slab(&shrink, 0, 0);
> +
> +  curusage = res_counter_read_u64(&memcg->kmem, RES_USAGE);
> +
> +  /* Usage is reduced ? */
> +  if (curusage >= oldusage)
> +   retry_count--;
> +  else
> +   oldusage = curusage;
> + }
> + return ret;
> +
> +}
> +
>  static int mem_cgroup_resize_memsw_limit(struct mem_cgroup *memcg,
>       unsigned long long val)
>  {
> @@ -3895,13 +3943,17 @@ static int mem_cgroup_write(struct cgroup *cont, struct cftype *cft,
>     break;
>    if (type == _MEM)
>     ret = mem_cgroup_resize_limit(memcg, val);
> +#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
>    else if (type == _KMEM) {
>     if (!memcg->kmem_independent_accounting) {
>      ret = -EINVAL;
>      break;
>     }
> -   ret = res_counter_set_limit(&memcg->kmem, val);
> -  } else
> +
> +   ret = mem_cgroup_resize_kmem_limit(memcg, val);
> +  }
```

> +#endif
> + else
>   ret = mem_cgroup_resize_memsw_limit(memcg, val);
>   break;
>   case RES_SOFT_LIMIT:
> @@ -5007,9 +5059,19 @@ struct memcg_kmem_cache *memcg_cache_get(struct
mem_cgroup *memcg, int index)
>
> void register_memcg_cache(struct memcg_cache_struct *cache)
> {
> + struct shrinker *shrink;
> +
>   BUG_ON(kmem_avail_caches[cache->index]);
>
>   kmem_avail_caches[cache->index] = cache;
> + if (!kmem_avail_caches[cache->index]->shrink_fn)
> + return;
> +
> + shrink = &kmem_avail_caches[cache->index]->shrink;
> + shrink->seeks = DEFAULT_SEEKS;
> + shrink->shrink = kmem_avail_caches[cache->index]->shrink_fn;
> + shrink->batch = 1024;
> + register_shrinker_memcg(shrink);
> }
>
> #define memcg_kmem(memcg) \
> @@ -5055,8 +5117,21 @@ int memcg_kmem_newpage(struct mem_cgroup *memcg, struct
page *page, unsigned lon
> {
>   unsigned long size = pages << PAGE_SHIFT;
>   struct res_counter *fail;
> + int ret;
> + bool do_softlimit;
> +
> + ret = res_counter_charge(memcg_kmem(memcg), size, &fail);
> + if (unlikely(mem_cgroup_event_ratelimit(memcg,
> +   MEM_CGROUP_TARGET_THRESH))) {
> +
> + do_softlimit = mem_cgroup_event_ratelimit(memcg,
> +   MEM_CGROUP_TARGET_SOFTLIMIT);
> + mem_cgroup_threshold(memcg);
> + if (unlikely(do_softlimit))
> +   mem_cgroup_update_tree(memcg, page);
> + }

Do we need to have this hook here ?
(BTW, please don't duplicate...)

>
> - return res_counter_charge(memcg_kmem(memcg), size, &fail);
> + return ret;
> }
>
> void memcg_kmem_freepage(struct mem_cgroup *memcg, struct page *page, unsigned long
pages)
> @@ -5083,6 +5158,7 @@ void memcg_create_kmem_caches(struct mem_cgroup *memcg)
>   else
>   memcg->kmem_cache[i].cache = kmem_cache_dup(memcg, cache);
>   INIT_WORK(&memcg->kmem_cache[i].destroy, memcg_cache_destroy);
> + INIT_LIST_HEAD(&memcg->kmem_cache[i].lru);
>   memcg->kmem_cache[i].memcg = memcg;
>   }
> }
> @@ -5157,6 +5233,11 @@ free_out:
>   return ERR_PTR(error);
> }
>
> +bool memcg_slab_reclaim(struct mem_cgroup *memcg)
> +{
> + return !memcg->kmem_independent_accounting;
> +}
> +
> void memcg_slab_destroy(struct kmem_cache *cache, struct mem_cgroup *memcg)
> {
>   int i;
> diff --git a/mm/vmscan.c b/mm/vmscan.c
> index c52b235..b9bceb6 100644
> --- a/mm/vmscan.c
> +++ b/mm/vmscan.c
> @@ -159,6 +159,23 @@ long vm_total_pages; /* The total number of pages which the VM
controls */
> static LIST_HEAD(shrinker_list);
> static DECLARE_RWSEM(shrinker_rwsem);
>
> +#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
> +/*
> + * If we could guarantee the root mem cgroup will always exist, we could just
> + * use the normal shrinker_list, and assume that the root memcg is passed
> + * as a parameter. But we're not quite there yet. Because of that, the shinkers
> + * from the memcg case can be different from the normal shrinker for the same
> + * object. This is not the ideal situation but is a step towards that.
> + *
> + * Also, not all caches will have their memcg version (also likely to change),
> + * so scanning the whole list is a waste.
> + *

> + * I am using, however, the same lock for both lists. Updates to it should
> + * be unfrequent, so I don't expect that to generate contention
> + */
> +static LIST_HEAD(shrinker_memcg_list);
> +#endif
> +
> #ifdef CONFIG_CGROUP_MEM_RES_CTLR
> static bool global_reclaim(struct scan_control *sc)
> {
> @@ -169,6 +186,11 @@ static bool scanning_global_lru(struct mem_cgroup_zone *mz)
> {
>   return !mz->mem_cgroup;
> }
> +
> +static bool global_slab_reclaim(struct scan_control *sc)
> +{
> + return !memcg_slab_reclaim(sc->target_mem_cgroup);
> +}

Do we need this new function ? global_reclaim() isn't enough ?

Thanks,
-Kame

> #else
> static bool global_reclaim(struct scan_control *sc)
> {
> @@ -179,6 +201,11 @@ static bool scanning_global_lru(struct mem_cgroup_zone *mz)
> {
>   return true;
> }
> +
> +static bool global_slab_reclaim(struct scan_control *sc)
> +{
> + return true;
> +}
> #endif
>
> static struct zone_reclaim_stat *get_reclaim_stat(struct mem_cgroup_zone *mz)
> @@ -225,6 +252,16 @@ void unregister_shrinker(struct shrinker *shrinker)
> }
> EXPORT_SYMBOL(unregister_shrinker);
>
> +#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
> +void register_shrinker_memcg(struct shrinker *shrinker)
> +{
> + atomic_long_set(&shrinker->nr_in_batch, 0);

> + down_write(&shrinker_rwsem);
> + list_add_tail(&shrinker->list, &shrinker_memcg_list);
> + up_write(&shrinker_rwsem);
> +}
> +#endif
> +
> static inline int do_shrinker_shrink(struct shrinker *shrinker,
> struct shrink_control *sc,
> unsigned long nr_to_scan)
> @@ -234,6 +271,18 @@ static inline int do_shrinker_shrink(struct shrinker *shrinker,
> }
>
> #define SHRINK_BATCH 128
> +
> +static inline struct list_head
> +*get_shrinker_list(struct shrink_control *shrink)
> +{
> +#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
> + if (shrink->memcg)
> + return &shrinker_memcg_list;
> + else
> +#endif
> + return &shrinker_list;
> +}
> +
> /*
> * Call the shrink functions to age shrinkable caches
> *
> @@ -259,6 +308,9 @@ unsigned long shrink_slab(struct shrink_control *shrink,
> {
> struct shrinker *shrinker;
> unsigned long ret = 0;
> + struct list_head *slist;
> +
> + slist = get_shrinker_list(shrink);
>
> if (nr_pages_scanned == 0)
> nr_pages_scanned = SWAP_CLUSTER_MAX;
> @@ -269,7 +321,7 @@ unsigned long shrink_slab(struct shrink_control *shrink,
> goto out;
> }
>
> - list_for_each_entry(shrinker, &shrinker_list, list) {
> + list_for_each_entry(shrinker, slist, list) {
> unsigned long long delta;
> long total_scan;
> long max_pass;
> @@ -2351,9 +2403,9 @@ static unsigned long do_try_to_free_pages(struct zonelist *zonelist,

```
>
>    /*
>     * Don't shrink slabs when reclaiming memory from
> -   * over limit cgroups
> +   * over limit cgroups, if kernel memory is controlled independently
>     */
> -  if (global_reclaim(sc)) {
> +  if (!global_slab_reclaim(sc)) {
>     unsigned long lru_pages = 0;
>     for_each_zone_zonelist(zone, z, zonelist,
>       gfp_zone(sc->gfp_mask)) {
> @@ -2362,8 +2414,10 @@ static unsigned long do_try_to_free_pages(struct zonelist *zonelist,
>
>      lru_pages += zone_reclaimable_pages(zone);
>     }
> +   shrink->memcg = sc->target_mem_cgroup;
>
>     shrink_slab(shrink, sc->nr_scanned, lru_pages);
> +
>     if (reclaim_state) {
>      sc->nr_reclaimed += reclaim_state->reclaimed_slab;
>      reclaim_state->reclaimed_slab = 0;
> --
> 1.7.7.6
>
> --
> To unsubscribe from this list: send the line "unsubscribe cgroups" in
> the body of a message to majordomo@vger.kernel.org
> More majordomo info at  http://vger.kernel.org/majordomo-info.html
>
```

Subject: Re: [PATCH 0/7] memcg kernel memory tracking
Posted by Pekka Enberg on Wed, 22 Feb 2012 07:08:26 GMT
View Forum Message <> Reply to Message

Hi Glauber,

On Tue, Feb 21, 2012 at 1:34 PM, Glauber Costa <glommer@parallels.com> wrote:
> This is a first structured approach to tracking general kernel
> memory within the memory controller. Please tell me what you think.

I like it! I only skimmed through the SLUB changes but they seemed
reasonable enough. What kind of performance hit are we taking when
memcg configuration option is enabled but the feature is disabled?

Pekka

Subject: Re: [PATCH 0/7] memcg kernel memory tracking
Posted by Glauber Costa on Wed, 22 Feb 2012 13:58:58 GMT
View Forum Message <> Reply to Message

>> As previously proposed, one has the option of keeping kernel memory
>> accounted separatedly, or together with the normal userspace memory.
>> However, this time I made the option to, in this later case, bill
>> the memory directly to memcg->res. It has the disadvantage that it becomes
>> complicated to know which memory came from user or kernel, but OTOH,
>> it does not create any overhead of drawing from multiple res_counters
>> at read time. (and if you want them to be joined, you probably don't care)
>
> It would be nice to still keep a kernel memory counter (that gets
> updated at the same time as memcg->res) even when the limits are not
> independent, because sometimes it's important to know how much kernel
> memory is being used by a cgroup.

Can you clarify in this "sometimes" ? The way I see it, we either always
use two counters - as did in my original proposal - or use a single
counter for this case. Keeping a separated counter and still billing to
the user memory is the worst of both worlds to me, since you get the
performance hit of updating two resource counters.

>> Kernel memory is never tracked for the root memory cgroup. This means
>> that a system where no memory cgroups exists other than the root, the
>> time cost of this implementation is a couple of branches in the slub
>> code - none of them in fast paths. At the moment, this works only
>> with the slub.
>>
>> At cgroup destruction, memory is billed to the parent. With no hierarchy,
>> this would mean the root memcg. But since we are not billing to that,
>> it simply ceases to be tracked.
>>
>> The caches that we want to be tracked need to explicit register into
>> the infrastructure.
>
> Why not track every cache unless otherwise specified? If you don't,
> you might end up polluting code all around the kernel to create
> per-cgroup caches.
> From what I've seen, there are a fair amount of different caches that
> can end up using a significant amount of memory, and having to go
> around and explicitly mark each one doesn't seem like the right thing
> to do.
>
The registration code is quite simple, so I don't agree this is
polluting code all around the kernel. It is just a couple of lines.

Of course, in an opt-out system, this count would be zero. So is it better?

Let's divide the caches in two groups: Ones that use shrinkers, and simple ones that won't do. I am assuming most of the ones we need to track use shrinkers somehow.

So if they do use a shrinker, it is very unlikely that the normal shrinkers will work without being memcg-aware. We then end up in a scenario in which we track memory, we create a bunch of new caches, but we can't really force reclaim on that cache. We then depend on luck to have the objects reclaimed from the root shrinker. Note that this is a problem that did not exist before: a dcache shrinker would shrink dcache objects and that's it, but we didn't have more than one cache with those objects.

So in this context, registering a cache explicitly is better IMHO, because what you are doing is telling: "I examined this cache, and I believe it will work okay with the memcg. It either does not need changes to the shrinker, or I made them already"

Also, everytime we create a new cache, we're wasting some memory, as we duplicate state. That is fine, since we're doing this to prevent the usage to explode.

But I am not sure it pays of in a lot of caches, even if they use a lot of pages: Like, quickly scanning slabinfo:

task_struct 512 570 5920 5 8 : tunables 0 0
0 : slabdata 114 114 0

Can only grow if # of processes grow. Likely to hit a limit on that first.

Acpi-Namespace 4348 5304 40 102 1 : tunables 0 0
0 : slabdata 52 52 0

I doubt we can take down a sane system by using this cache...

and so on and so forth.

What do you think?

Subject: Re: [PATCH 5/7] shrink support for memcg kmem controller
Posted by Glauber Costa on Wed, 22 Feb 2012 14:00:00 GMT
View Forum Message <> Reply to Message

On 02/22/2012 03:35 AM, Suleiman Souhlal wrote:
> On Tue, Feb 21, 2012 at 3:34 AM, Glauber Costa<glommer@parallels.com> wrote:
>
>> @@ -5055,8 +5117,21 @@ int memcg_kmem_newpage(struct mem_cgroup *memcg, struct

```
page *page, unsigned lon
>>  {
>>       unsigned long size = pages<<  PAGE_SHIFT;
>>       struct res_counter *fail;
>> +      int ret;
>> +      bool do_softlimit;
>> +
>> +      ret = res_counter_charge(memcg_kmem(memcg), size,&fail);
>> +      if (unlikely(mem_cgroup_event_ratelimit(memcg,
>> +                                      MEM_CGROUP_TARGET_THRESH))) {
>> +
>> +            do_softlimit = mem_cgroup_event_ratelimit(memcg,
>> +                                      MEM_CGROUP_TARGET_SOFTLIMIT);
>> +            mem_cgroup_threshold(memcg);
>> +            if (unlikely(do_softlimit))
>> +                  mem_cgroup_update_tree(memcg, page);
>> +      }
>>
>> -      return res_counter_charge(memcg_kmem(memcg), size,&fail);
>> +      return ret;
>>  }
>
> It seems like this might cause a lot of kernel memory allocations to
> fail whenever we are at the limit, even if we have a lot of
> reclaimable memory, when we don't have independent accounting.
>
> Would it be better to use __mem_cgroup_try_charge() here, when we
> don't have independent accounting, in order to deal with this
> situation?
>

Yes, it would.
I'll work on that.
```

Subject: Re: [PATCH 1/7] small cleanup for memcontrol.c
Posted by Glauber Costa on Wed, 22 Feb 2012 14:01:46 GMT
View Forum Message <> Reply to Message

```
On 02/22/2012 04:46 AM, KAMEZAWA Hiroyuki wrote:
> On Tue, 21 Feb 2012 15:34:33 +0400
> Glauber Costa<glommer@parallels.com>  wrote:
>
>> Move some hardcoded definitions to an enum type.
>>
>> Signed-off-by: Glauber Costa<glommer@parallels.com>
>> CC: Kirill A. Shutemov<kirill@shutemov.name>
>> CC: Greg Thelen<gthelen@google.com>
```

>> CC: Johannes Weiner<jweiner@redhat.com>
>> CC: Michal Hocko<mhocko@suse.cz>
>> CC: Hiroyouki Kamezawa<kamezawa.hiroyu@jp.fujitsu.com>
>> CC: Paul Turner<pjt@google.com>
>> CC: Frederic Weisbecker<fweisbec@gmail.com>
>
> seems ok to me.
>
> Acked-by: KAMEZAWA Hiroyuki<kamezawa.hiroyu@jp.fujitsu.com>

BTW, this series is likely to go through many rounds of discussion.
This patch can be probably picked separately, if you want to.

> a nitpick..
>
>> ---
>> mm/memcontrol.c | 10 +++++++---
>> 1 files changed, 7 insertions(+), 3 deletions(-)
>>
>> diff --git a/mm/memcontrol.c b/mm/memcontrol.c
>> index 6728a7a..b15a693 100644
>> --- a/mm/memcontrol.c
>> +++ b/mm/memcontrol.c
>> @@ -351,9 +351,13 @@ enum charge_type {
>> };
>>
>> /* for encoding cft->private value on file */
>> -#define _MEM (0)
>> -#define _MEMSWAP (1)
>> -#define _OOM_TYPE (2)
>> +
>> +enum mem_type {
>> + _MEM = 0,
>
> =0 is required ?
I believe not, but I always liked to use it to be 100 % explicit.
Personal taste... Can change it, if this is a big deal.

Subject: Re: [PATCH 3/7] per-cgroup slab caches
Posted by Glauber Costa on Wed, 22 Feb 2012 14:08:06 GMT
View Forum Message <> Reply to Message

On 02/22/2012 03:50 AM, Suleiman Souhlal wrote:
> On Tue, Feb 21, 2012 at 3:34 AM, Glauber Costa<glommer@parallels.com> wrote:
>> diff --git a/mm/memcontrol.c b/mm/memcontrol.c
>> index 26fda11..2aa35b0 100644
>> --- a/mm/memcontrol.c

```
>> +++ b/mm/memcontrol.c
>> +struct kmem_cache *
>> +kmem_cache_dup(struct mem_cgroup *memcg, struct kmem_cache *base)
>> +{
>> +	struct kmem_cache *s;
>> +	unsigned long pages;
>> +	struct res_counter *fail;
>> +	/*
>> +	 * TODO: We should use an ida-like index here, instead
>> +	 * of the kernel address
>> +	 */
>> +	char *kname = kasprintf(GFP_KERNEL, "%s-%p", base->name, memcg);
>
> Would it make more sense to use the memcg name instead of the pointer?

Well, yes. But at this point in creation time, we still don't have this
all setup. The css pointer is NULL, so I could not derive the name from
it. Instead of keep fighting what seemed to be a minor issue, I opted to
kick the patches out and be clear with a comment that this is not what I
intend in the way.

Do you know about any good way to grab the cgroup name at create() time ?

>> +
>> +	WARN_ON(mem_cgroup_is_root(memcg));
>> +
>> +	if (!kname)
>> +		return NULL;
>> +
>> +	s = kmem_cache_create_cg(memcg, kname, base->size,
>> +				 base->align, base->flags, base->ctor);
>> +	if (WARN_ON(!s))
>> +		goto out;
>> +
>> +
>> +	pages = slab_nr_pages(s);
>> +
>> +	if (res_counter_charge(memcg_kmem(memcg), pages<<  PAGE_SHIFT,&fail)) {
>> +		kmem_cache_destroy(s);
>> +		s = NULL;
>> +	}
>
> What are we charging here? Does it ever get uncharged?

We're charging the slab initial pages, that comes from allocations
outside allocate_slab(). But in this sense, it is not very different
than tin foil hats to protect against mind reading. Probably works, but
I am not sure the threat is real (also remembering we can probably want
```

to port it to the original slab allocator later - and let me be honest -
I know 0 about how that works).

So if the slab starts with 0 pages, this is a nop. If in any case it
does not, it gets uncharged when the cgroup is destroyed.

In all my tests, this played no role. If we can be sure that this won't
be an issue, I'll be happy to remove it.

Subject: Re: [PATCH 0/7] memcg kernel memory tracking
Posted by Glauber Costa on Wed, 22 Feb 2012 14:11:41 GMT
View Forum Message <> Reply to Message

On 02/22/2012 11:08 AM, Pekka Enberg wrote:
> Hi Glauber,
>
> On Tue, Feb 21, 2012 at 1:34 PM, Glauber Costa<glommer@parallels.com> wrote:
>> This is a first structured approach to tracking general kernel
>> memory within the memory controller. Please tell me what you think.
>
> I like it! I only skimmed through the SLUB changes but they seemed
> reasonable enough. What kind of performance hit are we taking when
> memcg configuration option is enabled but the feature is disabled?
>
>   Pekka
Thanks Pekka.

Well, I didn't took any numbers, because I don't consider the whole work
any close to final form, but I wanted people to comment anyway.

In particular, I intend to use the same trick I used for tcp sock
buffers here for this case - (static_branch()), so the performance hit
should come from two pointers in the kmem_cache structure - and I
believe it is possible to remove one of them.

I can definitely measure when I implement that, but I think it is
reasonable to expect not that much of a hit.

Subject: Re: [PATCH 3/7] per-cgroup slab caches
Posted by Glauber Costa on Wed, 22 Feb 2012 14:25:46 GMT
View Forum Message <> Reply to Message

On 02/22/2012 05:21 AM, KAMEZAWA Hiroyuki wrote:
> On Tue, 21 Feb 2012 15:34:35 +0400
> Glauber Costa<glommer@parallels.com> wrote:

>
>> This patch creates the infrastructure to allow us to register
>> per-memcg slab caches. As an example implementation, I am tracking
>> the dentry cache, but others will follow.
>>
>> I am using an opt-in istead of opt-out system here: this means the
>> cache needs to explicitly register itself to be tracked by memcg.
>> I prefer this approach since:
>>
>> 1) not all caches are big enough to be relevant,
>> 2) most of the relevant ones will involve shrinking in some way,
>>   and it would be better be sure they are shrinker-aware
>> 3) some objects, like network sockets, have their very own idea
>>   of memory control, that goes beyond the allocation itself.
>>
>> Once registered, allocations made on behalf of a task living
>> on a cgroup will be billed to it. It is a first-touch mechanism,
>> but if follows what we have today, and the cgroup infrastructure
>> itself. No overhead is expected in object allocation: only when
>> slab pages are allocated and freed, any form of billing occurs.
>>
>> The allocation stays billed to a cgroup until it is destroyed.
>> It is kept if a task leaves the cgroup, since it is close to
>> impossible to efficiently map an object to a task - and the tracking
>> is done by pages, which contain multiple objects.
>>
>
> Hmm....can't we do this by
>
>   kmem_cache = kmem_cache_create(...., ..., SLAB_XXX_XXX | SLAB_MEMCG_AWARE)
>
>   kmem_cache_alloc(kmem_cache, flags)
>   => find a memcg_kmem_cache for the thread. if it doesn't exist, create it.

Where exactly? Apart for the slab flag, I don't follow the rest of your
proposal.

>
> BTW, comparing ANON/FILE caches, we'll have many kinds of gfp_t flags.
>
> Do we handle it correctly ?

TODO item =)

>
> Maybe I don't fully understand your implemenatation...but try to comment.

Thanks.

>
>
>>
>> +struct memcg_kmem_cache {
>> + struct kmem_cache *cache;
>> + struct mem_cgroup *memcg; /* Should be able to do without this */
>> +};
>> +
>> +struct memcg_cache_struct {
>> + int index;
>> + struct kmem_cache *cache;
>> +};
>> +
>> +enum memcg_cache_indexes {
>> + CACHE_DENTRY,
>> + NR_CACHES,
>> +};
>> +
>> +int memcg_kmem_newpage(struct mem_cgroup *memcg, struct page *page, unsigned long
pages);
>> +void memcg_kmem_freepage(struct mem_cgroup *memcg, struct page *page, unsigned long
pages);
>> +struct mem_cgroup *memcg_from_shrinker(struct shrinker *s);
>> +
>> +struct memcg_kmem_cache *memcg_cache_get(struct mem_cgroup *memcg, int index);
>> +void register_memcg_cache(struct memcg_cache_struct *cache);
>> +void memcg_slab_destroy(struct kmem_cache *cache, struct mem_cgroup *memcg);
>> +
>> +struct kmem_cache *
>> +kmem_cache_dup(struct mem_cgroup *memcg, struct kmem_cache *base);
>> +
>> /* Stats that can be updated by kernel. */
>> enum mem_cgroup_page_stat_item {
>> MEMCG_NR_FILE_MAPPED, /* # of pages charged as file rss */
>> diff --git a/include/linux/slab.h b/include/linux/slab.h
>> index 573c809..8a372cd 100644
>> --- a/include/linux/slab.h
>> +++ b/include/linux/slab.h
>> @@ -98,6 +98,14 @@
>> void __init kmem_cache_init(void);
>> int slab_is_available(void);
>>
>> +struct mem_cgroup;
>> +
>> +unsigned long slab_nr_pages(struct kmem_cache *s);
>> +
>> +struct kmem_cache *kmem_cache_create_cg(struct mem_cgroup *memcg,
>> + const char *, size_t, size_t,

>> +   unsigned long,
>> +   void (*)(void *));
>>   struct kmem_cache *kmem_cache_create(const char *, size_t, size_t,
>>   unsigned long,
>>   void (*)(void *));
>> diff --git a/include/linux/slub_def.h b/include/linux/slub_def.h
>> index a32bcfd..4f39fff 100644
>> --- a/include/linux/slub_def.h
>> +++ b/include/linux/slub_def.h
>> @@ -100,6 +100,8 @@ struct kmem_cache {
>>   struct kobject kobj; /* For sysfs */
>>   #endif
>>
>> + struct mem_cgroup *memcg;
>> + struct kmem_cache *parent_slab; /* can be moved out of here as well */
>>   #ifdef CONFIG_NUMA
>>   /*
>>   * Defragmentation by allocating from a remote node.
>> diff --git a/mm/memcontrol.c b/mm/memcontrol.c
>> index 26fda11..2aa35b0 100644
>> --- a/mm/memcontrol.c
>> +++ b/mm/memcontrol.c
>> @@ -49,6 +49,7 @@
>>   #include<linux/page_cgroup.h>
>>   #include<linux/cpu.h>
>>   #include<linux/oom.h>
>> +#include<linux/slab.h>
>>   #include "internal.h"
>>   #include<net/sock.h>
>>   #include<net/tcp_memcontrol.h>
>> @@ -302,8 +303,11 @@ struct mem_cgroup {
>>   #ifdef CONFIG_INET
>>   struct tcp_memcontrol tcp_mem;
>>   #endif
>> + struct memcg_kmem_cache kmem_cache[NR_CACHES];
>>   };
>>
>> +struct memcg_cache_struct *kmem_avail_caches[NR_CACHES];
>> +
>
> What this pointer array holds ? This can be accessed without any locks ?
This holds all tracked caches types, with a pointer to the original
cache from which the memcg-specific cache is created.

This one can be a list instead, but kmem_cache[] inside the memcg
structure, not so sure. I need a quick way to grab the memcg-specific
cache from within the cache code (for the shrinkers), therefore I've
chosen to do that with the indexes. (Hummm, now thinking, we can just

use the root's memcg for that...)

Since every cache has its index in the scheme I implemented, we don't
need locking. There is only one writer that always writes a pointer,
always to the same location, All the others are readers, and all callers
can cope with this being NULL.

>
>
>>   /* Stuffs for move charges at task migration. */
>>   /*
>>   * Types of charges to be moved. "move_charge_at_immitgrate" is treated as a
>> @@ -4980,6 +4984,93 @@ err_cleanup:
>>
>>   }
>>
>> +struct memcg_kmem_cache *memcg_cache_get(struct mem_cgroup *memcg, int index)
>> +{
>> + return&memcg->kmem_cache[index];
>> +}
>> +
>> +void register_memcg_cache(struct memcg_cache_struct *cache)
>> +{
>> + BUG_ON(kmem_avail_caches[cache->index]);
>> +
>> + kmem_avail_caches[cache->index] = cache;
>> +}
>> +
>> +#define memcg_kmem(memcg) \
>> + (memcg->kmem_independent_accounting ?&memcg->kmem :&memcg->res)
>> +
>> +struct kmem_cache *
>> +kmem_cache_dup(struct mem_cgroup *memcg, struct kmem_cache *base)
>> +{
>> + struct kmem_cache *s;
>> + unsigned long pages;
>> + struct res_counter *fail;
>> + /*
>> + * TODO: We should use an ida-like index here, instead
>> + * of the kernel address
>> + */
>> + char *kname = kasprintf(GFP_KERNEL, "%s-%p", base->name, memcg);
>> +
>> + WARN_ON(mem_cgroup_is_root(memcg));
>> +
>> + if (!kname)
>> + return NULL;
>> +

>> + s = kmem_cache_create_cg(memcg, kname, base->size,
>> +   base->align, base->flags, base->ctor);
>> + if (WARN_ON(!s))
>> + goto out;
>> +
>> +
>> + pages = slab_nr_pages(s);
>> +
>> + if (res_counter_charge(memcg_kmem(memcg), pages<< PAGE_SHIFT,&fail)) {
>> + kmem_cache_destroy(s);
>> + s = NULL;
>> + }
>
> Why 'pages' should be charged to a new memcg ?
> A newly created memcg starts with res_counter.usage != 0 ??

See my last e-mail to Suleiman. It starts with usage == 0, unless it
creates pages outside the allocate_slab() mechanism. For the next
version, I'll go read both slab.c and slub.c extra carefully, and if
this is in case impossible, I'll remove this test.

Heck, actually, I will just remove this test no matter what. Even if
there are allocations outside of the tracked mechanism, it is easier to
track those, than to clutter this here...

>> +
>> + mem_cgroup_get(memcg);
>
> get even if allocation failure ? (and for what updating refcnt ?)

No, if we reached this point, the allocation succeeded. We don't need to
keep the memcg structure alive itself, but we need to keep the cache
alive (since it will still have objects upon memcg destruction).

As a policy, I am trying to avoid putting more stuff in struct
kmem_cache, so I put it here.

>
>
>> +out:
>> + kfree(kname);
>> + return s;
>> +}
>
>
>> +
>> +int memcg_kmem_newpage(struct mem_cgroup *memcg, struct page *page, unsigned long
pages)

```
>> +{
>> + unsigned long size = pages<<  PAGE_SHIFT;
>> + struct res_counter *fail;
>> +
>> + return res_counter_charge(memcg_kmem(memcg), size,&fail);
>> +}
>> +
>> +void memcg_kmem_freepage(struct mem_cgroup *memcg, struct page *page, unsigned long
pages)
>> +{
>> + unsigned long size = pages<<  PAGE_SHIFT;
>> +
>> + res_counter_uncharge(memcg_kmem(memcg), size);
>> +}
>> +
>> +void memcg_create_kmem_caches(struct mem_cgroup *memcg)
>> +{
>> + int i;
>> +
>> + for (i = 0; i<  NR_CACHES; i++) {
>> +  struct kmem_cache *cache;
>> +
>> +  if (!kmem_avail_caches[i] || !kmem_avail_caches[i]->cache)
>> +   continue;
>> +
>> +  cache = kmem_avail_caches[i]->cache;
>> +
>> +  if (mem_cgroup_is_root(memcg))
>> +   memcg->kmem_cache[i].cache = cache;
>> +  else
>> +   memcg->kmem_cache[i].cache = kmem_cache_dup(memcg, cache);
>> +  memcg->kmem_cache[i].memcg = memcg;
>> + }
>> +}
>
> Hmm... memcg should know _all_ kmem_caches when it's created. Right ?
> Then, modules will not use memcg aware kmem_cache...
```

Which maybe is not a bad thing. Again, see my last e-mail to Suleiman, I described my reasons there.

But thing is, we can always reclaim user pages. But this is not true for kernel pages. We need the help of a shrinker. So if you don't have a memcg-aware shrinker, maybe it would be better not to get tracked in the first place.

```
>> +
>> +
```

>> static struct cgroup_subsys_state * __ref
>> mem_cgroup_create(struct cgroup_subsys *ss, struct cgroup *cont)
>> {
>> @@ -5039,6 +5130,8 @@ mem_cgroup_create(struct cgroup_subsys *ss, struct cgroup
*cont)
>>   if (parent)
>>   memcg->swappiness = mem_cgroup_swappiness(parent);
>>   atomic_set(&memcg->refcnt, 1);
>> +
>> + memcg_create_kmem_caches(memcg);
>>   memcg->move_charge_at_immigrate = 0;
>>   mutex_init(&memcg->thresholds_lock);
>>   return&memcg->css;
>> diff --git a/mm/slub.c b/mm/slub.c
>> index 4907563..f3815ec 100644
>> --- a/mm/slub.c
>> +++ b/mm/slub.c
>> @@ -31,6 +31,8 @@
>> #include<linux/stacktrace.h>
>>
>> #include<trace/events/kmem.h>
>> +#include<linux/cgroup.h>
>> +#include<linux/memcontrol.h>
>>
>> /*
>>   * Lock order:
>> @@ -1281,6 +1283,7 @@ static struct page *allocate_slab(struct kmem_cache *s, gfp_t flags,
int node)
>>   struct page *page;
>>   struct kmem_cache_order_objects oo = s->oo;
>>   gfp_t alloc_gfp;
>> + int pages;
>>
>>   flags&= gfp_allowed_mask;
>>
>> @@ -1314,9 +1317,17 @@ static struct page *allocate_slab(struct kmem_cache *s, gfp_t
flags, int node)
>>   if (!page)
>>   return NULL;
>>
>> + pages = 1<< oo_order(oo);
>> +
>> +#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
>> + if (s->memcg&& memcg_kmem_newpage(s->memcg, page, pages)< 0) {
>> + __free_pages(page, oo_order(oo));
>> + return NULL;
>> + }
>> +#endif

>
> Hmm. no reclaim happens at allocation failure ?
> Doesn't it turn to be very bad user experience ?

Yes, I'll handle it in the next respin.

Subject: Re: [PATCH 4/7] chained slab caches: move pages to a different cache when a cache is destroyed.
Posted by Glauber Costa on Wed, 22 Feb 2012 14:50:58 GMT
View Forum Message <> Reply to Message

On 02/22/2012 03:40 AM, Suleiman Souhlal wrote:
> On Tue, Feb 21, 2012 at 3:34 AM, Glauber Costa<glommer@parallels.com> wrote:
>> In the context of tracking kernel memory objects to a cgroup, the
>> following problem appears: we may need to destroy a cgroup, but
>> this does not guarantee that all objects inside the cache are dead.
>> This can't be guaranteed even if we shrink the cache beforehand.
>>
>> The simple option is to simply leave the cache around. However,
>> intensive workloads may have generated a lot of objects and thus
>> the dead cache will live in memory for a long while.
>
> Why is this a problem?
>
> Leaving the cache around while there are still active objects in it
> would certainly be a lot simpler to understand and implement.
>

Yeah, I agree on the simplicity. The chained stuff was probably the
hardest one in the patchset to get working alright. However, my
assumptions are as follow:

1) If we bother to be tracking kernel memory, it is because we believe
its usage can skyrocket under certain circumstances. In those scenarios,
we'll have a lot of objects around. If we just let them flowing, it's
just wasted memory that was created from the memcg, but can't be
reclaimed on its behalf.

2) We can reclaim that, if we have, as a policy, to always start
shrinking from those when global pressure kicks in. But then, we move
the complication from one part to another.

Subject: Re: [PATCH 5/7] shrink support for memcg kmem controller
Posted by Glauber Costa on Wed, 22 Feb 2012 14:53:25 GMT
View Forum Message <> Reply to Message

On 02/22/2012 05:42 AM, KAMEZAWA Hiroyuki wrote:
> On Tue, 21 Feb 2012 15:34:37 +0400
> Glauber Costa<glommer@parallels.com> wrote:
>
>> This patch adds the shrinker interface to memcg proposed kmem
>> controller. With this, softlimits starts being meaningful. I didn't
>> played to much with softlimits itself, since it is a bit in progress
>> for the general case as well. But this patch at least makes vmscan.c
>> no longer skip shrink_slab for the memcg case.
>>
>> It also allows us to set the hard limit to a lower value than
>> current usage, as it is possible for the current memcg: a reclaim
>> is carried on, and if we succeed in freeing enough of kernel memory,
>> we can lower the limit.
>>
>> Signed-off-by: Glauber Costa<glommer@parallels.com>
>> CC: Kirill A. Shutemov<kirill@shutemov.name>
>> CC: Greg Thelen<gthelen@google.com>
>> CC: Johannes Weiner<jweiner@redhat.com>
>> CC: Michal Hocko<mhocko@suse.cz>
>> CC: Hiroyouki Kamezawa<kamezawa.hiroyu@jp.fujitsu.com>
>> CC: Paul Turner<pjt@google.com>
>> CC: Frederic Weisbecker<fweisbec@gmail.com>
>> CC: Pekka Enberg<penberg@kernel.org>
>> CC: Christoph Lameter<cl@linux.com>
>> ---
>>   include/linux/memcontrol.h | 5 +++
>>   include/linux/shrinker.h | 4 ++
>>   mm/memcontrol.c | 87 ++--
>>   mm/vmscan.c | 60 ++++++++++++++++++++++++++++--
>>   4 files changed, 150 insertions(+), 6 deletions(-)
>>
>> diff --git a/include/linux/memcontrol.h b/include/linux/memcontrol.h
>> index 6138d10..246b2d4 100644
>> --- a/include/linux/memcontrol.h
>> +++ b/include/linux/memcontrol.h
>> @@ -33,12 +33,16 @@ struct mm_struct;
>>   struct memcg_kmem_cache {
>>   struct kmem_cache *cache;
>>   struct work_struct destroy;
>> + struct list_head lru;
>> + u32 nr_objects;
>>   struct mem_cgroup *memcg; /* Should be able to do without this */
>>   };
>>
>>   struct memcg_cache_struct {
>>   int index;
>>   struct kmem_cache *cache;

>> + int (*shrink_fn)(struct shrinker *shrink, struct shrink_control *sc);
>> + struct shrinker shrink;
>> };
>>
>> enum memcg_cache_indexes {
>> @@ -53,6 +57,7 @@ struct mem_cgroup *memcg_from_shrinker(struct shrinker *s);
>> struct memcg_kmem_cache *memcg_cache_get(struct mem_cgroup *memcg, int index);
>> void register_memcg_cache(struct memcg_cache_struct *cache);
>> void memcg_slab_destroy(struct kmem_cache *cache, struct mem_cgroup *memcg);
>> +bool memcg_slab_reclaim(struct mem_cgroup *memcg);
>>
>> struct kmem_cache *
>> kmem_cache_dup(struct mem_cgroup *memcg, struct kmem_cache *base);
>> diff --git a/include/linux/shrinker.h b/include/linux/shrinker.h
>> index 07ceb97..11efdba 100644
>> --- a/include/linux/shrinker.h
>> +++ b/include/linux/shrinker.h
>> @@ -1,6 +1,7 @@
>> #ifndef _LINUX_SHRINKER_H
>> #define _LINUX_SHRINKER_H
>>
>> +struct mem_cgroup;
>> /*
>>   * This struct is used to pass information from page reclaim to the shrinkers.
>>   * We consolidate the values for easier extention later.
>> @@ -10,6 +11,7 @@ struct shrink_control {
>>
>>   /* How many slab objects shrinker() should scan and try to reclaim */
>>   unsigned long nr_to_scan;
>> + struct mem_cgroup *memcg;
>> };
>>
>> /*
>> @@ -40,4 +42,6 @@ struct shrinker {
>> #define DEFAULT_SEEKS 2 /* A good number if you don't know better. */
>> extern void register_shrinker(struct shrinker *);
>> extern void unregister_shrinker(struct shrinker *);
>> +
>> +extern void register_shrinker_memcg(struct shrinker *);
>> #endif
>> diff --git a/mm/memcontrol.c b/mm/memcontrol.c
>> index 1b1db88..9c89a3c 100644
>> --- a/mm/memcontrol.c
>> +++ b/mm/memcontrol.c
>> @@ -3460,6 +3460,54 @@ static int mem_cgroup_resize_limit(struct mem_cgroup *memcg,
>>   return ret;
>> }
>>

```
>> +static int mem_cgroup_resize_kmem_limit(struct mem_cgroup *memcg,
>> +     unsigned long long val)
>> +{
>> +
>> + int retry_count;
>> + int ret = 0;
>> + int children = mem_cgroup_count_children(memcg);
>> + u64 curusage, oldusage;
>> +
>> + struct shrink_control shrink = {
>> +  .gfp_mask = GFP_KERNEL,
>> +  .memcg = memcg,
>> + };
>> +
>> + /*
>> +  * For keeping hierarchical_reclaim simple, how long we should retry
>> +  * is depends on callers. We set our retry-count to be function
>> +  * of # of children which we should visit in this loop.
>> +  */
>> + retry_count = MEM_CGROUP_RECLAIM_RETRIES * children;
>> +
>> + oldusage = res_counter_read_u64(&memcg->kmem, RES_USAGE);
>> +
>> + while (retry_count) {
>> +  if (signal_pending(current)) {
>> +   ret = -EINTR;
>> +   break;
>> +  }
>> +  mutex_lock(&set_limit_mutex);
>> +  ret = res_counter_set_limit(&memcg->kmem, val);
>> +  mutex_unlock(&set_limit_mutex);
>> +  if (!ret)
>> +   break;
>> +
>> +  shrink_slab(&shrink, 0, 0);
>> +
>> +  curusage = res_counter_read_u64(&memcg->kmem, RES_USAGE);
>> +
>> +  /* Usage is reduced ? */
>> +  if (curusage>= oldusage)
>> +   retry_count--;
>> +  else
>> +   oldusage = curusage;
>> + }
>> + return ret;
>> +
>> +}
>> +
```

>> static int mem_cgroup_resize_memsw_limit(struct mem_cgroup *memcg,
>> unsigned long long val)
>> {
>> @@ -3895,13 +3943,17 @@ static int mem_cgroup_write(struct cgroup *cont, struct cftype
*cft,
>> break;
>> if (type == _MEM)
>> ret = mem_cgroup_resize_limit(memcg, val);
>> +#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
>> else if (type == _KMEM) {
>> if (!memcg->kmem_independent_accounting) {
>> ret = -EINVAL;
>> break;
>> }
>> - ret = res_counter_set_limit(&memcg->kmem, val);
>> - } else
>> +
>> + ret = mem_cgroup_resize_kmem_limit(memcg, val);
>> + }
>> +#endif
>> + else
>> ret = mem_cgroup_resize_memsw_limit(memcg, val);
>> break;
>> case RES_SOFT_LIMIT:
>> @@ -5007,9 +5059,19 @@ struct memcg_kmem_cache *memcg_cache_get(struct
mem_cgroup *memcg, int index)
>>
>> void register_memcg_cache(struct memcg_cache_struct *cache)
>> {
>> + struct shrinker *shrink;
>> +
>> BUG_ON(kmem_avail_caches[cache->index]);
>>
>> kmem_avail_caches[cache->index] = cache;
>> + if (!kmem_avail_caches[cache->index]->shrink_fn)
>> + return;
>> +
>> + shrink =&kmem_avail_caches[cache->index]->shrink;
>> + shrink->seeks = DEFAULT_SEEKS;
>> + shrink->shrink = kmem_avail_caches[cache->index]->shrink_fn;
>> + shrink->batch = 1024;
>> + register_shrinker_memcg(shrink);
>> }
>>
>> #define memcg_kmem(memcg) \
>> @@ -5055,8 +5117,21 @@ int memcg_kmem_newpage(struct mem_cgroup *memcg, struct
page *page, unsigned lon
>> {

>>   unsigned long size = pages<< PAGE_SHIFT;
>>   struct res_counter *fail;
>> + int ret;
>> + bool do_softlimit;
>> +
>> + ret = res_counter_charge(memcg_kmem(memcg), size,&fail);
>> + if (unlikely(mem_cgroup_event_ratelimit(memcg,
>> +   MEM_CGROUP_TARGET_THRESH))) {
>> +
>> + do_softlimit = mem_cgroup_event_ratelimit(memcg,
>> +   MEM_CGROUP_TARGET_SOFTLIMIT);
>> + mem_cgroup_threshold(memcg);
>> + if (unlikely(do_softlimit))
>> +   mem_cgroup_update_tree(memcg, page);
>> + }
>
> Do we need to have this hook here ?
> (BTW, please don't duplicate...)
>
>
>>
>> - return res_counter_charge(memcg_kmem(memcg), size,&fail);
>> + return ret;
>>   }
>>
>>   void memcg_kmem_freepage(struct mem_cgroup *memcg, struct page *page, unsigned long
pages)
>> @@ -5083,6 +5158,7 @@ void memcg_create_kmem_caches(struct mem_cgroup *memcg)
>>   else
>>   memcg->kmem_cache[i].cache = kmem_cache_dup(memcg, cache);
>>   INIT_WORK(&memcg->kmem_cache[i].destroy, memcg_cache_destroy);
>> + INIT_LIST_HEAD(&memcg->kmem_cache[i].lru);
>>   memcg->kmem_cache[i].memcg = memcg;
>>   }
>>   }
>> @@ -5157,6 +5233,11 @@ free_out:
>>   return ERR_PTR(error);
>>   }
>>
>> +bool memcg_slab_reclaim(struct mem_cgroup *memcg)
>> +{
>> + return !memcg->kmem_independent_accounting;
>> +}
>> +
>>   void memcg_slab_destroy(struct kmem_cache *cache, struct mem_cgroup *memcg)
>>   {
>>   int i;
>> diff --git a/mm/vmscan.c b/mm/vmscan.c

>> index c52b235..b9bceb6 100644
>> --- a/mm/vmscan.c
>> +++ b/mm/vmscan.c
>> @@ -159,6 +159,23 @@ long vm_total_pages; /* The total number of pages which the VM controls */
>> static LIST_HEAD(shrinker_list);
>> static DECLARE_RWSEM(shrinker_rwsem);
>>
>> +#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
>> +/*
>> + * If we could guarantee the root mem cgroup will always exist, we could just
>> + * use the normal shrinker_list, and assume that the root memcg is passed
>> + * as a parameter. But we're not quite there yet. Because of that, the shinkers
>> + * from the memcg case can be different from the normal shrinker for the same
>> + * object. This is not the ideal situation but is a step towards that.
>> + *
>> + * Also, not all caches will have their memcg version (also likely to change),
>> + * so scanning the whole list is a waste.
>> + *
>> + * I am using, however, the same lock for both lists. Updates to it should
>> + * be unfrequent, so I don't expect that to generate contention
>> + */
>> +static LIST_HEAD(shrinker_memcg_list);
>> +#endif
>> +
>> #ifdef CONFIG_CGROUP_MEM_RES_CTLR
>> static bool global_reclaim(struct scan_control *sc)
>> {
>> @@ -169,6 +186,11 @@ static bool scanning_global_lru(struct mem_cgroup_zone *mz)
>> {
>>   return !mz->mem_cgroup;
>> }
>> +
>> +static bool global_slab_reclaim(struct scan_control *sc)
>> +{
>> + return !memcg_slab_reclaim(sc->target_mem_cgroup);
>> +}
>
> Do we need this new function ? global_reclaim() isn't enough ?

When we're tracking kmem separately, yes, because the softlimit happens
on a different counter.

However, I'll try to merge it somehow with the normal code in the next
run, and I'll keep this in mind. Let's see how it ends up...

Subject: Re: [PATCH 4/7] chained slab caches: move pages to a different cache when a cache is destroyed.
Posted by Glauber Costa on Wed, 22 Feb 2012 14:57:17 GMT
View Forum Message <> Reply to Message

On 02/22/2012 05:25 AM, KAMEZAWA Hiroyuki wrote:
> On Tue, 21 Feb 2012 15:34:36 +0400
> Glauber Costa<glommer@parallels.com> wrote:
>
>> In the context of tracking kernel memory objects to a cgroup, the
>> following problem appears: we may need to destroy a cgroup, but
>> this does not guarantee that all objects inside the cache are dead.
>> This can't be guaranteed even if we shrink the cache beforehand.
>>
>> The simple option is to simply leave the cache around. However,
>> intensive workloads may have generated a lot of objects and thus
>> the dead cache will live in memory for a long while.
>>
>> Scanning the list of objects in the dead cache takes time, and
>> would probably require us to lock the free path of every objects
>> to make sure we're not racing against the update.
>>
>> I decided to give a try to a different idea then - but I'd be
>> happy to pursue something else if you believe it would be better.
>>
>> Upon memcg destruction, all the pages on the partial list
>> are moved to the new slab (usually the parent memcg, or root memcg)
>> When an object is freed, there are high stakes that no list locks
>> are needed - so this case poses no overhead. If list manipulation
>> is indeed needed, we can detect this case, and perform it
>> in the right slab.
>>
>> If all pages were residing in the partial list, we can free
>> the cache right away. Otherwise, we do it when the last cache
>> leaves the full list.
>>
>
> How about starting from 'don't handle slabs on dead memcg'
> if shrink_slab() can find them....
>
> This "move" complicates all implementation, I think...
>

You mean, whenever pressure kicks in, start by reclaiming from dead
memcg? Well, I can work with that, for sure. I am not that sure that
this will be a win, but there is only way to know for sure.

Note that in this case, we need to keep the memcg around anyway. Also,
it is yet another reason why I believe we should explicit register a

cache for being tracked. If your memcg is gone, but the objects are not,
you really depend on the shrinker to make them go away. So we better
make sure this works before registering.

Also, I have another question for you guys: How would you feel if we
triggered an agressive slab reclaim before deleting the memcg? With
this, maybe we can reduce the pages considerably - and probably get rid
of the memcg altogether at destruction stage.

Subject: Re: [PATCH 0/7] memcg kernel memory tracking
Posted by Suleiman Souhlal on Wed, 22 Feb 2012 20:32:03 GMT
View Forum Message <> Reply to Message

On Wed, Feb 22, 2012 at 5:58 AM, Glauber Costa <glommer@parallels.com> wrote:
>>> As previously proposed, one has the option of keeping kernel memory
>>> accounted separatedly, or together with the normal userspace memory.
>>> However, this time I made the option to, in this later case, bill
>>> the memory directly to memcg->res. It has the disadvantage that it
>>> becomes
>>> complicated to know which memory came from user or kernel, but OTOH,
>>> it does not create any overhead of drawing from multiple res_counters
>>> at read time. (and if you want them to be joined, you probably don't
>>> care)
>>
>>
>> It would be nice to still keep a kernel memory counter (that gets
>> updated at the same time as memcg->res) even when the limits are not
>> independent, because sometimes it's important to know how much kernel
>> memory is being used by a cgroup.
>
>
> Can you clarify in this "sometimes" ? The way I see it, we either always use
> two counters - as did in my original proposal - or use a single counter for
> this case. Keeping a separated counter and still billing to the user memory
> is the worst of both worlds to me, since you get the performance hit of
> updating two resource counters.

By "sometimes", I mean pretty much any time we have to debug why a
cgroup is out of memory.
If there is no counter for how much kernel memory is used, it's pretty
much impossible to determine why the cgroup is full.

As for the performance, I do not think it is bad, as the accounting is
done in the slow path of slab allocation, when we allocate/free pages.

>
>

>>> Kernel memory is never tracked for the root memory cgroup. This means
>>> that a system where no memory cgroups exists other than the root, the
>>> time cost of this implementation is a couple of branches in the slub
>>> code - none of them in fast paths. At the moment, this works only
>>> with the slub.
>>>
>>> At cgroup destruction, memory is billed to the parent. With no hierarchy,
>>> this would mean the root memcg. But since we are not billing to that,
>>> it simply ceases to be tracked.
>>>
>>> The caches that we want to be tracked need to explicit register into
>>> the infrastructure.
>>
>>
>> Why not track every cache unless otherwise specified? If you don't,
>> you might end up polluting code all around the kernel to create
>> per-cgroup caches.
>> From what I've seen, there are a fair amount of different caches that
>> can end up using a significant amount of memory, and having to go
>> around and explicitly mark each one doesn't seem like the right thing
>> to do.
>>
> The registration code is quite simple, so I don't agree this is polluting
> code all around the kernel. It is just a couple of lines.
>
> Of course, in an opt-out system, this count would be zero. So is it better?
>
> Let's divide the caches in two groups: Ones that use shrinkers, and simple
> ones that won't do. I am assuming most of the ones we need to track use
> shrinkers somehow.
>
> So if they do use a shrinker, it is very unlikely that the normal shrinkers
> will work without being memcg-aware. We then end up in a scenario in which
> we track memory, we create a bunch of new caches, but we can't really force
> reclaim on that cache. We then depend on luck to have the objects reclaimed
> from the root shrinker. Note that this is a problem that did not exist
> before: a dcache shrinker would shrink dcache objects and that's it, but we
> didn't have more than one cache with those objects.
>
> So in this context, registering a cache explicitly is better IMHO, because
> what you are doing is telling: "I examined this cache, and I believe it will
> work okay with the memcg. It either does not need changes to the shrinker,
> or I made them already"
>
> Also, everytime we create a new cache, we're wasting some memory, as we
> duplicate state. That is fine, since we're doing this to prevent the usage
> to explode.
>

> But I am not sure it pays of in a lot of caches, even if they use a lot of
> pages: Like, quickly scanning slabinfo:
>
> task_struct 512 570 5920 5 8 : tunables 0 0 0 :
> slabdata 114 114 0
>
> Can only grow if # of processes grow. Likely to hit a limit on that first.
>
> Acpi-Namespace 4348 5304 40 102 1 : tunables 0 0 0 :
> slabdata 52 52 0
>
> I doubt we can take down a sane system by using this cache...
>
> and so on and so forth.
>
> What do you think?

Well, we've seen several slabs that don't have shrinkers use
significant amounts of memory. For example, size-64, size-32,
vm_area_struct, buffer_head, radix_tree_node, TCP, filp..

For example, consider this perl program (with high enough file
descriptor limits):
use POSIX; use Socket; my $i; for ($i = 0; $i < 100000; $i++) {
socket($i, PF_INET, SOCK_STREAM, 0) || die "socket: $!"; }

One can make other simple programs like this that use significant
amounts of slab memory.

Having to look at a running kernel, having to find out which caches
are significant, and then going back and marking them for accounting,
really doesn't seem the right approach to me.

-- Suleiman

Subject: Re: [PATCH 0/7] memcg kernel memory tracking
Posted by Ying Han on Thu, 23 Feb 2012 18:18:22 GMT
View Forum Message <> Reply to Message

On Tue, Feb 21, 2012 at 3:34 AM, Glauber Costa <glommer@parallels.com> wrote:
> This is a first structured approach to tracking general kernel
> memory within the memory controller. Please tell me what you think.
>
> As previously proposed, one has the option of keeping kernel memory
> accounted separatedly, or together with the normal userspace memory.
> However, this time I made the option to, in this later case, bill
> the memory directly to memcg->res. It has the disadvantage that it becomes

> complicated to know which memory came from user or kernel, but OTOH,
> it does not create any overhead of drawing from multiple res_counters
> at read time. (and if you want them to be joined, you probably don't care)

Keeping one counter for user and kernel pages makes it easier for
admins to configure the system. About reporting, we should still
report the user and kernel memory separately. It will be extremely
useful when diagnosing the system like heavily memory pressure or OOM.

> Kernel memory is never tracked for the root memory cgroup. This means
> that a system where no memory cgroups exists other than the root, the
> time cost of this implementation is a couple of branches in the slub
> code - none of them in fast paths. At the moment, this works only
> with the slub.
>
> At cgroup destruction, memory is billed to the parent. With no hierarchy,
> this would mean the root memcg. But since we are not billing to that,
> it simply ceases to be tracked.
>
> The caches that we want to be tracked need to explicit register into
> the infrastructure.

It would be hard to let users to register which slab to track
explicitly. We should track them all in general, even with the ones
without shrinker, we want to understand how much is used by which
cgroup.

--Ying

>
> If you would like to give it a try, you'll need one of Frederic's patches
> that is used as a basis for this
> (cgroups: ability to stop res charge propagation on bounded ancestor)
>
> Glauber Costa (7):
> small cleanup for memcontrol.c
> Basic kernel memory functionality for the Memory Controller
> per-cgroup slab caches
> chained slab caches: move pages to a different cache when a cache is
>   destroyed.
> shrink support for memcg kmem controller
> track dcache per-memcg
> example shrinker for memcg-aware dcache
>
> fs/dcache.c | 136 ++++++++++++++++++-
> include/linux/dcache.h | 4 +
> include/linux/memcontrol.h | 35 +++++
> include/linux/shrinker.h | 4 +

> include/linux/slab.h | 12 ++
> include/linux/slub_def.h | 3 +
> mm/memcontrol.c | 344
++-
> mm/slub.c | 237 ++++++++++++++++++++++++++---
> mm/vmscan.c | 60 ++++++++-
> 9 files changed, 806 insertions(+), 29 deletions(-)
>
> --
> 1.7.7.6
>
> --
> To unsubscribe, send a message with 'unsubscribe linux-mm' in
> the body to majordomo@kvack.org. For more info on Linux MM,
> see: http://www.linux-mm.org/ .
> Fight unfair telecom internet charges in Canada: sign http://stopthemeter.ca/
> Don't email: email@kvack.org

Subject: Re: [PATCH 0/7] memcg kernel memory tracking
Posted by Glauber Costa on Tue, 28 Feb 2012 19:02:32 GMT
View Forum Message <> Reply to Message

On 02/23/2012 04:18 PM, Ying Han wrote:
> On Tue, Feb 21, 2012 at 3:34 AM, Glauber Costa<glommer@parallels.com> wrote:
>> This is a first structured approach to tracking general kernel
>> memory within the memory controller. Please tell me what you think.
>>
>> As previously proposed, one has the option of keeping kernel memory
>> accounted separatedly, or together with the normal userspace memory.
>> However, this time I made the option to, in this later case, bill
>> the memory directly to memcg->res. It has the disadvantage that it becomes
>> complicated to know which memory came from user or kernel, but OTOH,
>> it does not create any overhead of drawing from multiple res_counters
>> at read time. (and if you want them to be joined, you probably don't care)
>
> Keeping one counter for user and kernel pages makes it easier for
> admins to configure the system. About reporting, we should still
> report the user and kernel memory separately. It will be extremely
> useful when diagnosing the system like heavily memory pressure or OOM.

It will also make us charge two different res_counters, which is not a
cheap operation.

I was wondering if we can do something smarter within the res_counter
itself to avoid taking locks for two different res_counters in the
charge path?

Subject: Re: [PATCH 1/7] small cleanup for memcontrol.c
Posted by Glauber Costa on Wed, 29 Feb 2012 17:30:35 GMT
View Forum Message <> Reply to Message

On 02/22/2012 12:01 PM, Glauber Costa wrote:
> On 02/22/2012 04:46 AM, KAMEZAWA Hiroyuki wrote:
>> On Tue, 21 Feb 2012 15:34:33 +0400
>> Glauber Costa<glommer@parallels.com> wrote:
>>
>>> Move some hardcoded definitions to an enum type.
>>>
>>> Signed-off-by: Glauber Costa<glommer@parallels.com>
>>> CC: Kirill A. Shutemov<kirill@shutemov.name>
>>> CC: Greg Thelen<gthelen@google.com>
>>> CC: Johannes Weiner<jweiner@redhat.com>
>>> CC: Michal Hocko<mhocko@suse.cz>
>>> CC: Hiroyouki Kamezawa<kamezawa.hiroyu@jp.fujitsu.com>
>>> CC: Paul Turner<pjt@google.com>
>>> CC: Frederic Weisbecker<fweisbec@gmail.com>
>>
>> seems ok to me.
>>
>> Acked-by: KAMEZAWA Hiroyuki<kamezawa.hiroyu@jp.fujitsu.com>
>
> BTW, this series is likely to go through many rounds of discussion.
> This patch can be probably picked separately, if you want to.
>
>> a nitpick..
>>
>>> ---
>>> mm/memcontrol.c | 10 +++++++---
>>> 1 files changed, 7 insertions(+), 3 deletions(-)
>>>
>>> diff --git a/mm/memcontrol.c b/mm/memcontrol.c
>>> index 6728a7a..b15a693 100644
>>> --- a/mm/memcontrol.c
>>> +++ b/mm/memcontrol.c
>>> @@ -351,9 +351,13 @@ enum charge_type {
>>> };
>>>
>>> /* for encoding cft->private value on file */
>>> -#define _MEM (0)
>>> -#define _MEMSWAP (1)
>>> -#define _OOM_TYPE (2)
>>> +
>>> +enum mem_type {
>>> + _MEM = 0,
>>
>> =0 is required ?

> I believe not, but I always liked to use it to be 100 % explicit.
> Personal taste... Can change it, if this is a big deal.

Kame, would you like me to send this cleanup without the = 0 ?

Subject: Re: [PATCH 1/7] small cleanup for memcontrol.c
Posted by KAMEZAWA Hiroyuki on Thu, 01 Mar 2012 02:11:47 GMT
View Forum Message <> Reply to Message

On Wed, 29 Feb 2012 14:30:35 -0300
Glauber Costa <glommer@parallels.com> wrote:

> On 02/22/2012 12:01 PM, Glauber Costa wrote:
> > On 02/22/2012 04:46 AM, KAMEZAWA Hiroyuki wrote:
> >> On Tue, 21 Feb 2012 15:34:33 +0400
> >> Glauber Costa<glommer@parallels.com> wrote:
> >>
> >>> Move some hardcoded definitions to an enum type.
> >>>
> >>> Signed-off-by: Glauber Costa<glommer@parallels.com>
> >>> CC: Kirill A. Shutemov<kirill@shutemov.name>
> >>> CC: Greg Thelen<gthelen@google.com>
> >>> CC: Johannes Weiner<jweiner@redhat.com>
> >>> CC: Michal Hocko<mhocko@suse.cz>
> >>> CC: Hiroyouki Kamezawa<kamezawa.hiroyu@jp.fujitsu.com>
> >>> CC: Paul Turner<pjt@google.com>
> >>> CC: Frederic Weisbecker<fweisbec@gmail.com>
> >>
> >> seems ok to me.
> >>
> >> Acked-by: KAMEZAWA Hiroyuki<kamezawa.hiroyu@jp.fujitsu.com>
> >
> > BTW, this series is likely to go through many rounds of discussion.
> > This patch can be probably picked separately, if you want to.
> >
> >> a nitpick..
> >>
> >>> ---
> >>> mm/memcontrol.c | 10 +++++++---
> >>> 1 files changed, 7 insertions(+), 3 deletions(-)
> >>>
> >>> diff --git a/mm/memcontrol.c b/mm/memcontrol.c
> >>> index 6728a7a..b15a693 100644
> >>> --- a/mm/memcontrol.c
> >>> +++ b/mm/memcontrol.c
> >>> @@ -351,9 +351,13 @@ enum charge_type {
> >>> };

> >>>
> >>> /* for encoding cft->private value on file */
> >>> -#define _MEM (0)
> >>> -#define _MEMSWAP (1)
> >>> -#define _OOM_TYPE (2)
> >>> +
> >>> +enum mem_type {
> >>> + _MEM = 0,
> >>
> >> =0 is required ?
> > I believe not, but I always liked to use it to be 100 % explicit.
> > Personal taste... Can change it, if this is a big deal.
>
> Kame, would you like me to send this cleanup without the = 0 ?
>

Not necessary. Sorry for delayed response.
Lots of memcg patches are flying..

Thanks,
-Kame
