Subject: What does glommer think about kmem cgroup ?
Posted by Glauber Costa on Thu, 13 Oct 2011 15:50:29 GMT
View Forum Message <> Reply to Message

Hi guys,

So, linuxcon is approaching. To help making our discussions more productive, I sketched a basic prototype of a kmem cgroup that can control the size of the dentry cache. I am sending the code here so you guys can have an idea, but keep in mind this is a *sketch*. This is my view of how our controller *could be*, not necessarily what it *should be*. All your input is more than welcome.

Let me first explain a bit of my approach: (there are some comments inline as well)

* So far it only works with the slab (you will see that something similar can be done for at least the slub) Since most of us is concerned mostly with memory abuse (I think), I neglected for simplicity the initial memory allocated for the arrays. Only when cache_grow is called to allocated more pages, is that we bill then.

* I avoid resorting to the shrinkers, trying to free the slab pages themselves whenever possible.

* We don't limit the size of all caches. They have to register themselves explicitly (and in this PoC, I am using the dentry cache as an example)

* The object is billed to whoever touched it first. Other policies are of course possible.

What I am *not* concerned about in this PoC: (left for future work, if needed)
- unified user/memory kernel memory reclaim
- changes to the shrinkers.
- changes to the limit once it is already in place
- per-cgroup display in /proc/slabinfo
- task movement
- a whole lot of other stuff.

* Hey glommer, do you have numbers?
Yes, I have 8 numbers. And since 8 is also a number, then I have 9 numbers.

So what I did was to type "find /" in a freshly booted system (my laptop). I just ran each iteration once, so nothing scientific. I halved the limits until the allocations started to fail, which was more or less around 256K hard limit. Find is also not a workload that

pins the dentries in memory for very long. Other kinds of workloads will display different results here...

Base: (non-patched kernel)
real 0m16.091s
user 0m0.567s
sys 0m6.649s

Patched kernel, root cgroup (unlimited. max used mem: 22Mb)
real 0m15.853s
user 0m0.511s
sys 0m6.417s

16Mb/4Mb (HardLimit/SoftLimit)
real 0m16.596s
user 0m0.560s
sys 0m6.947s

8Mb/4Mb
real 0m16.975s
user 0m0.568s
sys 0m7.047s

4Mb/2Mb
real 0m16.713s
user 0m0.554s
sys 0m7.022s

2Mb/1Mb
real 0m17.001s
user 0m0.544s
sys 0m7.118s

1Mb/512K
real 0m16.671s
user 0m0.530s
sys 0m7.067s

512k/256k
real 0m17.395s
user 0m0.567s
sys 0m7.179s

So, what those initial numbers do tell us, is that the performance penalty for the root cgroup is not expected to be that bad. When the limits start to be hit, a penalty is incurred, which is under the expectations.

```
diff --git a/fs/dcache.c b/fs/dcache.c
index a88948b..cd5d091 100644
--- a/fs/dcache.c
+++ b/fs/dcache.c
@@ -142,7 +142,7 @@ 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->dentry_cache, dentry);
 }

 /*
@@ -1160,6 +1160,8 @@ void shrink_dcache_parent(struct dentry * parent)
 }
 EXPORT_SYMBOL(shrink_dcache_parent);

+static struct memcg_slab_ctlr dcache_ctlr;
+
 /**
  * __d_alloc - allocate a dcache entry
  * @sb: filesystem it will belong to
@@ -1173,9 +1175,14 @@ EXPORT_SYMBOL(shrink_dcache_parent);
 struct dentry *__d_alloc(struct super_block *sb, const struct qstr *name)
 {
  struct dentry *dentry;
+ struct kmem_cache *current_dcache;
  char *dname;

- dentry = kmem_cache_alloc(dentry_cache, GFP_KERNEL);
+ current_dcache = current_kmem_cache(dcache_ctlr.token);
+ if (unlikely(!current_dcache))
+  current_dcache = dentry_cache;
+
+ dentry = kmem_cache_alloc(current_dcache, GFP_KERNEL);
  if (!dentry)
   return NULL;

@@ -1210,6 +1217,7 @@ struct dentry *__d_alloc(struct super_block *sb, const struct qstr
*name)
  INIT_LIST_HEAD(&dentry->d_alias);
  INIT_LIST_HEAD(&dentry->d_u.d_child);
  d_set_d_op(dentry, dentry->d_sb->s_d_op);
+ dentry->dentry_cache = current_dcache;

  this_cpu_inc(nr_dentry);
```

```
@@ -2945,6 +2953,35 @@ static void __init dcache_init_early(void)
  INIT_HLIST_BL_HEAD(dentry_hashtable + loop);
 }

+static struct kmem_cache *dcache_new_kmem(struct mem_cgroup *memcg,
+       struct mem_cgroup *parent)
+{
+ char *name;
+
+ if (!parent)
+  return dentry_cache;
+
+ name = kasprintf(GFP_KERNEL, "memcg-dentry-%p", memcg);
+
+ return kmem_cache_create(name, sizeof(struct dentry),
+    __alignof__(struct dentry),
+    SLAB_RECLAIM_ACCOUNT|SLAB_PANIC|SLAB_MEM_SPREAD, NULL);
+}
+
+static void dcache_destroy_kmem(struct kmem_cache *cachep)
+{
+ if (cachep == dentry_cache)
+  return;
+
+ kfree(cachep->name);
+ kmem_cache_destroy(cachep);
+}
+
+static struct memcg_slab_ctlr dcache_ctlr = {
+ .memcg_create_kmem_cache = dcache_new_kmem,
+ .memcg_destroy_kmem_cache = dcache_destroy_kmem,
+};
+
 static void __init dcache_init(void)
 {
  int loop;
@@ -2957,6 +2994,8 @@ static void __init dcache_init(void)
  dentry_cache = KMEM_CACHE(dentry,
   SLAB_RECLAIM_ACCOUNT|SLAB_PANIC|SLAB_MEM_SPREAD);

+ register_memcg_slab_ctlr(&dcache_ctlr);
+
  /* Hash may have been set up in dcache_init_early */
  if (!hashdist)
   return;
diff --git a/include/linux/dcache.h b/include/linux/dcache.h
index 62157c0..a002292 100644
--- a/include/linux/dcache.h
```

```
+++ b/include/linux/dcache.h
@@ -142,6 +142,7 @@ struct dentry {
  } d_u;
  struct list_head d_subdirs; /* our children */
  struct list_head d_alias; /* inode alias list */
+ struct kmem_cache *dentry_cache;
 };

 /*
diff --git a/include/linux/memcontrol.h b/include/linux/memcontrol.h
index 343bd76..1b7ef0c 100644
--- a/include/linux/memcontrol.h
+++ b/include/linux/memcontrol.h
@@ -158,6 +158,9 @@ void mem_cgroup_split_huge_fixup(struct page *head, struct page *tail);
 bool mem_cgroup_bad_page_check(struct page *page);
 void mem_cgroup_print_bad_page(struct page *page);
 #endif
+
+struct kmem_cache *current_kmem_cache(int token);
+
 #else /* CONFIG_CGROUP_MEM_RES_CTLR */
 struct mem_cgroup;

@@ -376,5 +379,33 @@ mem_cgroup_print_bad_page(struct page *page)
 }
 #endif

+struct memcg_slab_ctlr {
+ struct list_head list;
+ struct kmem_cache *cachep;
+ struct kmem_cache *(*memcg_create_kmem_cache)(struct mem_cgroup *memcg,
+           struct mem_cgroup *parent);
+ void (*memcg_destroy_kmem_cache)(struct kmem_cache *cachep);
+ int token;
+};
+
+#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
+int mem_cgroup_kmem_charge(struct kmem_cache *cachep, gfp_t flags, int nr_pages);
+void mem_cgroup_kmem_uncharge(struct kmem_cache *cachep, int nr_pages);
+void register_memcg_slab_ctlr(struct memcg_slab_ctlr *ctlr);
+
+#else
+static inline int mem_cgroup_kmem_charge(struct kmem_cache *cachep, gfp_t flags, int
nr_pages)
+{
+ return 0;
+}
+static inline void mem_cgroup_kmem_uncharge(struct kmem_cache *cachep, int nr_pages)
```

```
+{
+}
+
+static inline void register_memcg_slab_ctlr(struct memcg_slab_ctlr *ctlr)
+{
+}
+#endif
+
 #endif /* _LINUX_MEMCONTROL_H */

diff --git a/include/linux/slab.h b/include/linux/slab.h
index 573c809..0bdefe2 100644
--- a/include/linux/slab.h
+++ b/include/linux/slab.h
@@ -103,6 +103,7 @@ struct kmem_cache *kmem_cache_create(const char *, size_t, size_t,
    void (*)(void *));
 void kmem_cache_destroy(struct kmem_cache *);
 int kmem_cache_shrink(struct kmem_cache *);
+int kmem_cache_shrink_pages(struct kmem_cache *, int nr_pages);
 void kmem_cache_free(struct kmem_cache *, void *);
 unsigned int kmem_cache_size(struct kmem_cache *);

diff --git a/include/linux/slab_def.h b/include/linux/slab_def.h
index d00e0ba..1ec3bc0 100644
--- a/include/linux/slab_def.h
+++ b/include/linux/slab_def.h
@@ -23,6 +23,8 @@
  * manages a cache.
  */

+struct mem_cgroup;
+
 struct kmem_cache {
 /* 1) Cache tunables. Protected by cache_chain_mutex */
  unsigned int batchcount;
@@ -55,6 +57,8 @@ struct kmem_cache {
 /* 4) cache creation/removal */
  const char *name;
  struct list_head next;
+ struct mem_cgroup *memcg;
+ struct list_head memcg_list;

 /* 5) statistics */
 #ifdef CONFIG_DEBUG_SLAB
@@ -111,6 +115,15 @@ extern struct cache_sizes malloc_sizes[];
 void *kmem_cache_alloc(struct kmem_cache *, gfp_t);
 void *__kmalloc(size_t size, gfp_t flags);
```

```
+static inline void slab_associate_memcg(struct kmem_cache *kmem, struct mem_cgroup
*memcg)
+{
+ kmem->memcg = memcg;
+}
+static inline struct mem_cgroup *slab_cache_memcg(struct kmem_cache *kmem)
+{
+ return kmem->memcg;
+}
+
 #ifdef CONFIG_TRACING
 extern void *kmem_cache_alloc_trace(size_t size,
         struct kmem_cache *cachep, gfp_t flags);
diff --git a/mm/memcontrol.c b/mm/memcontrol.c
index 71de028..616e9f1 100644
--- a/mm/memcontrol.c
+++ b/mm/memcontrol.c
@@ -294,8 +294,16 @@ struct mem_cgroup {
   */
  struct mem_cgroup_stat_cpu nocpu_base;
  spinlock_t pcp_counter_lock;
+
+ struct kmem_cache *kmem_caches[1024];
+
+// struct kmem_cache *dentry_cache;
 };

+static DEFINE_RWLOCK(memcg_slab_lock);
+static LIST_HEAD(memcg_slab_list);
+static int memcg_token = 0;
+
 /* Stuffs for move charges at task migration. */
 /*
  * Types of charges to be moved. "move_charge_at_immitgrate" is treated as a
@@ -2730,6 +2738,84 @@ static int mem_cgroup_charge_common(struct page *page, struct
mm_struct *mm,
  return 0;
 }

+/* this header is purely bogus */
+int mem_cgroup_kmem_charge(struct kmem_cache *cachep, gfp_t gfp_mask, int pages)
+{
+ struct mem_cgroup *memcg = slab_cache_memcg(cachep);
+ struct res_counter *dummy;
+ unsigned long long excess;
+ int ret, shrink_attempts;
+
+ if (!memcg)
```

```
+  return 0;
+
+ ret = res_counter_charge(&memcg->kmem, PAGE_SIZE * pages, &dummy);
+
+ /* FIXME: This could very well come from charge for free!! */
+ excess = res_counter_soft_limit_excess(&memcg->kmem) >> PAGE_SHIFT;
+
+ /*
+  * If we really have a total kmem cgroup, this one assumes the current
+  * allocator is the one to blame. It might not be ideal in some cases,
+  * but it should get it right in general. An alternative approach would
+  * be to loop the following code through all registered caches in this
+  * memcg.
+  */
+ if (!excess)
+  return ret;
+
+ /*
+  * root memcg should never be in excess, thus, never triggering any
+  * reclaim
+  */
+ WARN_ON(mem_cgroup_is_root(memcg));
+
+ /*
+  * First, we try a generic reclaim. This only touches the current
+  * cachep, not other users in the system. However, it doesn't know
+  * anything about the objects, and is not guaranteed to do a good job.
+  *
+  */
+ excess -= kmem_cache_shrink_pages(cachep, excess << 1);
+
+ shrink_attempts = 0;
+ /*
+  * Second attempt: If we don't think we freed enough
+  * entities, try the shrinkers. 5 attempts is pretty arbitrary.
+  */
+ while (excess > 0 && shrink_attempts++ < 5) {
+  int nr;
+
+  struct shrink_control shrink = {
+   .gfp_mask = GFP_KERNEL,
+  };
+
+  /*
+   * We need a better reclaim mechanism that
+   * at least does not punish other cgroups
+   */
+  nr = shrink_slab(&shrink, 1000, 1000);
```

```
+  if (!nr)
+   break;
+        }
+
+ /* After all these shrinks, worth another attempt */
+ if (ret)
+  ret = res_counter_charge(&memcg->kmem, PAGE_SIZE * pages, &dummy);
+
+ return ret;
+}
+
+void mem_cgroup_kmem_uncharge(struct kmem_cache *cachep, int pages)
+{
+ struct mem_cgroup *memcg = slab_cache_memcg(cachep);
+
+ if (!memcg)
+  return;
+
+ res_counter_uncharge(&memcg->kmem, PAGE_SIZE * pages);
+}
+
 int mem_cgroup_newpage_charge(struct page *page,
         struct mm_struct *mm, gfp_t gfp_mask)
 {
@@ -3928,6 +4014,8 @@ static int mem_cgroup_write(struct cgroup *cont, struct cftype *cft,
    break;
   if (type == _MEM)
    ret = mem_cgroup_resize_limit(memcg, val);
+  else if (type == _KMEM)
+   WARN_ON(res_counter_set_limit(&memcg->kmem, val));
   else
    ret = mem_cgroup_resize_memsw_limit(memcg, val);
   break;
@@ -3942,6 +4030,8 @@ static int mem_cgroup_write(struct cgroup *cont, struct cftype *cft,
    */
   if (type == _MEM)
    ret = res_counter_set_soft_limit(&memcg->res, val);
+  else if (type == _KMEM)
+   WARN_ON(res_counter_set_soft_limit(&memcg->kmem, val));
   else
    ret = -EINVAL;
   break;
@@ -3998,6 +4088,8 @@ static int mem_cgroup_reset(struct cgroup *cont, unsigned int event)
  case RES_FAILCNT:
   if (type == _MEM)
    res_counter_reset_failcnt(&mem->res);
+  else if (type == _KMEM)
+   res_counter_reset_failcnt(&mem->kmem);
```

```
   else
    res_counter_reset_failcnt(&mem->memsw);
   break;
@@ -4754,6 +4846,7 @@ static struct cftype memsw_cgroup_files[] = {
   .trigger = mem_cgroup_reset,
   .read_u64 = mem_cgroup_read,
  },
+
 };

 static int register_memsw_files(struct cgroup *cont, struct cgroup_subsys *ss)
@@ -4786,8 +4879,28 @@ static struct cftype kmem_cgroup_files[] = {
  {
   .name = "kmem.limit_in_bytes",
   .private = MEMFILE_PRIVATE(_KMEM, RES_LIMIT),
+  .write_string = mem_cgroup_write,
   .read_u64 = mem_cgroup_read,
  },
+ {
+  .name = "kmem.soft_limit_in_bytes",
+  .private = MEMFILE_PRIVATE(_KMEM, RES_SOFT_LIMIT),
+  .write_string = mem_cgroup_write,
+  .read_u64 = mem_cgroup_read,
+ },
+ {
+  .name = "kmem.failcnt",
+  .private = MEMFILE_PRIVATE(_KMEM, RES_FAILCNT),
+  .trigger = mem_cgroup_reset,
+  .read_u64 = mem_cgroup_read,
+ },
+ {
+  .name = "kmem.max_usage_in_bytes",
+  .private = MEMFILE_PRIVATE(_KMEM, RES_MAX_USAGE),
+  .trigger = mem_cgroup_reset,
+  .read_u64 = mem_cgroup_read,
+ },
+
 };

 static int register_kmem_files(struct cgroup *cont, struct cgroup_subsys *ss)
@@ -4966,6 +5079,65 @@ static int mem_cgroup_soft_limit_tree_init(void)
  return 0;
 }

+struct kmem_cache *current_kmem_cache(int token)
+{
+ struct mem_cgroup *memcg;
+ struct kmem_cache *cachep = NULL;
```

```
+
+ rcu_read_lock();
+ memcg = mem_cgroup_from_task(current);
+
+ if (!memcg)
+  goto out;
+
+ cachep = memcg->kmem_caches[token];
+out:
+ rcu_read_unlock();
+ return cachep;
+}
+
+static void create_slab_caches(struct mem_cgroup *memcg, struct mem_cgroup *parent)
+{
+ struct memcg_slab_ctlr *ctlr;
+ read_lock(&memcg_slab_lock);
+
+ list_for_each_entry(ctlr, &memcg_slab_list, list) {
+  struct kmem_cache *cachep;
+  cachep = ctlr->memcg_create_kmem_cache(memcg, parent);
+  memcg->kmem_caches[ctlr->token] = cachep;
+  slab_associate_memcg(cachep, memcg);
+ }
+
+ read_unlock(&memcg_slab_lock);
+}
+
+static void destroy_slab_caches(struct mem_cgroup *memcg)
+{
+ struct memcg_slab_ctlr *ctlr;
+
+ read_lock(&memcg_slab_lock);
+ list_for_each_entry(ctlr, &memcg_slab_list, list)
+  ctlr->memcg_destroy_kmem_cache(memcg->kmem_caches[ctlr->token]);
+
+ read_unlock(&memcg_slab_lock);
+}
+
+void register_memcg_slab_ctlr(struct memcg_slab_ctlr *ctlr)
+{
+ int tk;
+
+ write_lock(&memcg_slab_lock);
+ tk = memcg_token++;
+ if (tk > 1024) {
+  printk(KERN_WARNING "Too many slab caches for kmem cgroup\n");
+  goto out;
```

```
+ }
+ INIT_LIST_HEAD(&ctlr->list);
+ list_add(&ctlr->list, &memcg_slab_list);
+ ctlr->token = tk;
+out:
+ write_unlock(&memcg_slab_lock);
+}
 static struct cgroup_subsys_state * __ref
 mem_cgroup_create(struct cgroup_subsys *ss, struct cgroup *cont)
 {
@@ -4995,11 +5167,14 @@ mem_cgroup_create(struct cgroup_subsys *ss, struct cgroup *cont)
    INIT_WORK(&stock->work, drain_local_stock);
   }
   hotcpu_notifier(memcg_cpu_hotplug_callback, 0);
+  //mem->dentry_cache = dentry_cache;
  } else {
   parent = mem_cgroup_from_cont(cont->parent);
   mem->use_hierarchy = parent->use_hierarchy;
   mem->oom_kill_disable = parent->oom_kill_disable;
+  //mem->dentry_cache = new_dcache();
  }
+ create_slab_caches(mem, parent);

  if (parent && parent->use_hierarchy) {
   res_counter_init(&mem->res, &parent->res);
@@ -5046,6 +5221,8 @@ static void mem_cgroup_destroy(struct cgroup_subsys *ss,
 {
  struct mem_cgroup *mem = mem_cgroup_from_cont(cont);

+ destroy_slab_caches(mem);
+
  mem_cgroup_put(mem);
 }

diff --git a/mm/slab.c b/mm/slab.c
index 6d90a09..d83f312 100644
--- a/mm/slab.c
+++ b/mm/slab.c
@@ -1742,6 +1742,12 @@ static void *kmem_getpages(struct kmem_cache *cachep, gfp_t
flags, int nodeid)
   return NULL;

  nr_pages = (1 << cachep->gfporder);
+
+ if (mem_cgroup_kmem_charge(cachep, flags, nr_pages)) {
+  free_pages((unsigned long)page_address(page), cachep->gfporder);
+  return NULL;
+ }
```

```
+
  if (cachep->flags & SLAB_RECLAIM_ACCOUNT)
   add_zone_page_state(page_zone(page),
    NR_SLAB_RECLAIMABLE, nr_pages);
@@ -1787,6 +1793,9 @@ static void kmem_freepages(struct kmem_cache *cachep, void *addr)
  }
  if (current->reclaim_state)
   current->reclaim_state->reclaimed_slab += nr_freed;
+
+ mem_cgroup_kmem_uncharge(cachep, nr_freed);
+
  free_pages((unsigned long)addr, cachep->gfporder);
 }

@@ -2604,6 +2613,29 @@ out:
 }

 /* Called with cache_chain_mutex held to protect against cpu hotplug */
+static int __cache_shrink_pages(struct kmem_cache *cachep, int pages)
+{
+ int i = 0;
+ struct kmem_list3 *l3;
+ int freed = 0;
+
+ drain_cpu_caches(cachep);
+
+ check_irq_on();
+ for_each_online_node(i) {
+  l3 = cachep->nodelists[i];
+  if (!l3)
+   continue;
+
+  freed += drain_freelist(cachep, l3, l3->free_objects);
+  if (freed >= pages)
+   break;
+ }
+
+ return freed;
+}
+
+/* Called with cache_chain_mutex held to protect against cpu hotplug */
 static int __cache_shrink(struct kmem_cache *cachep)
 {
  int ret = 0, i = 0;
@@ -2622,8 +2654,29 @@ static int __cache_shrink(struct kmem_cache *cachep)
   ret += !list_empty(&l3->slabs_full) ||
    !list_empty(&l3->slabs_partial);
  }
```

```
+
  return (ret ? 1 : 0);
 }
+/**
+ * kmem_cache_shrink_pages - Shrink a cache.
+ * @cachep: The cache to shrink.
+ *
+ * Releases as many slabs as possible for a cache.
+ * To help debugging, a zero exit status indicates all slabs were released.
+ */
+int kmem_cache_shrink_pages(struct kmem_cache *cachep, int nr_pages)
+{
+ int ret;
+ BUG_ON(!cachep || in_interrupt());
+
+ get_online_cpus();
+ mutex_lock(&cache_chain_mutex);
+ ret = __cache_shrink_pages(cachep, nr_pages);
+ mutex_unlock(&cache_chain_mutex);
+ put_online_cpus();
+ return ret;
+}
+EXPORT_SYMBOL(kmem_cache_shrink_pages);

 /**
  * kmem_cache_shrink - Shrink a cache.
@@ -2643,6 +2696,7 @@ int kmem_cache_shrink(struct kmem_cache *cachep)
  mutex_unlock(&cache_chain_mutex);
  put_online_cpus();
  return ret;
+
 }
 EXPORT_SYMBOL(kmem_cache_shrink);
```

File Attachments

1) basic-code.patch, downloaded 646 times