Subject: [PATCH 00/10] memcg kmem limitation - slab.
Posted by Glauber Costa on Wed, 25 Jul 2012 14:38:11 GMT
View Forum Message <> Reply to Message

Hi,

This is the slab part of the kmem limitation mechanism in its last form. I would like to have comments on it to see if we can agree in its form. I consider it mature, since it doesn't change much in essence over the last forms. However, I would still prefer to defer merging it and merge the stack-only patchset first (even if inside the same merge window). That patchset contains most of the infrastructure needed here, and merging them separately would not only reduce the complexity for reviewers, but allow us a chance to have independent testing on them both. I would also likely benefit from some extra testing, to make sure the recent changes didn't introduce anything bad.

A general explanation of what this is all about follows:

The kernel memory limitation mechanism for memcg concerns itself with disallowing potentially non-reclaimable allocations to happen in exagerate quantities by a particular set of processes (cgroup). Those allocations could create pressure that affects the behavior of a different and unrelated set of processes.

Its basic working mechanism is to annotate some allocations with the _GFP_KMEMCG flag. When this flag is set, the current process allocating will have its memcg identified and charged against. When reaching a specific limit, further allocations will be denied.

One example of such problematic pressure that can be prevented by this work is a fork bomb conducted in a shell. We prevent it by noting that processes use a limited amount of stack pages. Seen this way, a fork bomb is just a special case of resource abuse. If the offender is unable to grab more pages for the stack, no new processes can be created.

There are also other things the general mechanism protects against. For example, using too much of pinned dentry and inode cache, by touching files an leaving them in memory forever.

In fact, a simple:

while true; do mkdir x; cd x; done

can halt your system easily, because the file system limits are hard to reach (big disks), but the kernel memory is not. Those are examples, but the list certainly don't stop here.

An important use case for all that is concerned with people offering hosting

services through containers. In a physical box, we can put a limit to some resources, like total number of processes or threads. But in an environment where each independent user gets its own piece of the machine, we don't want a potentially malicious user to destroy good users' services.

This might be true for systemd as well, that now groups services inside cgroups. They generally want to put forward a set of guarantees that limits the running service in a variety of ways, so that if they become badly behaved, they won't interfere with the rest of the system.

There is, of course, there is a cost for that. To attempt to mitigate that, static branches are used to make sure that even if the feature is compiled in with potentially a lot of memory cgroups deployed this code will only be enabled after the first user of this service configures any limit. Limits lower than the user limit effectively means there is a separate kernel memory limit that may be reached independently than the user limit. Values equal or greater than the user limit implies only that kernel memory is tracked. This provides a unified vision of "maximum memory", be it kernel or user memory. Because this is all default-off, existing deployments will see no change in behavior.

Glauber Costa (10):
  slab/slub: struct memcg_params
  consider a memcg parameter in kmem_create_cache
  memcg: infrastructure to match an allocation to the right cache
  memcg: skip memcg kmem allocations in specified code regions
  slab: allow enable_cpu_cache to use preset values for its tunables
  sl[au]b: Allocate objects from memcg cache
  memcg: destroy memcg caches
  memcg/sl[au]b Track all the memcg children of a kmem_cache.
  slab: slab-specific propagation changes.
  memcg/sl[au]b: shrink dead caches

 include/linux/memcontrol.h | 54 ++++++
 include/linux/sched.h | 1 +
 include/linux/slab.h | 25 +++
 include/linux/slab_def.h | 4 +
 include/linux/slub_def.h | 21 ++-
 init/Kconfig | 2 +-
 mm/memcontrol.c | 414 ++-
 mm/slab.c | 57 +++++--
 mm/slab.h | 55 +++++-
 mm/slab_common.c | 69 +++++++-
 mm/slub.c | 25 ++-
 11 files changed, 702 insertions(+), 25 deletions(-)

--
1.7.10.4

Subject: [PATCH 01/10] slab/slub: struct memcg_params
Posted by Glauber Costa on Wed, 25 Jul 2012 14:38:12 GMT
View Forum Message <> Reply to Message

For the kmem slab controller, we need to record some extra
information in the kmem_cache structure.

Signed-off-by: Glauber Costa <glommer@parallels.com>
Signed-off-by: Suleiman Souhlal <suleiman@google.com>
CC: Christoph Lameter <cl@linux.com>
CC: Pekka Enberg <penberg@cs.helsinki.fi>
CC: Michal Hocko <mhocko@suse.cz>
CC: Kamezawa Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com>
CC: Johannes Weiner <hannes@cmpxchg.org>
---
 include/linux/slab.h | 7 +++++++
 include/linux/slab_def.h | 4 ++++
 include/linux/slub_def.h | 3 +++
 3 files changed, 14 insertions(+)

diff --git a/include/linux/slab.h b/include/linux/slab.h
index 0dd2dfa..3152bcd 100644
--- a/include/linux/slab.h
+++ b/include/linux/slab.h
@@ -177,6 +177,13 @@ unsigned int kmem_cache_size(struct kmem_cache *);
 #define ARCH_SLAB_MINALIGN __alignof__(unsigned long long)
 #endif

+#ifdef CONFIG_MEMCG_KMEM
+struct mem_cgroup_cache_params {
+ struct mem_cgroup *memcg;
+ int id;
+};
+#endif
+
 /*
  * Common kmalloc functions provided by all allocators
  */
diff --git a/include/linux/slab_def.h b/include/linux/slab_def.h
index 0c634fa..39c5e9d 100644
--- a/include/linux/slab_def.h
+++ b/include/linux/slab_def.h
@@ -83,6 +83,10 @@ struct kmem_cache {
  int obj_offset;
 #endif /* CONFIG_DEBUG_SLAB */

+#ifdef CONFIG_MEMCG_KMEM
+ struct mem_cgroup_cache_params memcg_params;
+#endif

```
+
 /* 6) per-cpu/per-node data, touched during every alloc/free */
  /*
   * We put array[] at the end of kmem_cache, because we want to size
diff --git a/include/linux/slub_def.h b/include/linux/slub_def.h
index df448ad..8bb8ad2 100644
--- a/include/linux/slub_def.h
+++ b/include/linux/slub_def.h
@@ -101,6 +101,9 @@ struct kmem_cache {
 #ifdef CONFIG_SYSFS
  struct kobject kobj; /* For sysfs */
 #endif
+#ifdef CONFIG_MEMCG_KMEM
+ struct mem_cgroup_cache_params memcg_params;
+#endif

 #ifdef CONFIG_NUMA
  /*
--
1.7.10.4
```

Subject: [PATCH 02/10] consider a memcg parameter in kmem_create_cache
Posted by Glauber Costa on Wed, 25 Jul 2012 14:38:13 GMT
View Forum Message <> Reply to Message

Allow a memcg parameter to be passed during cache creation.
When the slub allocator is being used, it will only merge
caches that belong to the same memcg.

Default function is created as a wrapper, passing NULL
to the memcg version. We only merge caches that belong
to the same memcg.

>From the memcontrol.c side, 3 helper functions are created:

1) memcg_css_id: because slub needs a unique cache name
   for sysfs. Since this is visible, but not the canonical
   location for slab data, the cache name is not used, the
   css_id should suffice.

2) mem_cgroup_register_cache: is responsible for assigning
   a unique index to each cache, and other general purpose
   setup. The index is only assigned for the root caches. All
   others are assigned index == -1.

3) mem_cgroup_release_cache: can be called from the root cache
   destruction, and will release the index for

other caches.

We can't assign indexes until the basic slab is up and running
this is because the ida subsystem will itself call slab functions
such as kmalloc a couple of times. Because of that, we have
a late_initcall that scan all caches and register them after the
kernel is booted up. Only caches registered after that receive
their index right away.

This index mechanism was developed by Suleiman Souhlal.
Changed to a idr/ida based approach based on suggestion
from Kamezawa.

Signed-off-by: Glauber Costa <glommer@parallels.com>
CC: Christoph Lameter <cl@linux.com>
CC: Pekka Enberg <penberg@cs.helsinki.fi>
CC: Michal Hocko <mhocko@suse.cz>
CC: Kamezawa Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com>
CC: Johannes Weiner <hannes@cmpxchg.org>
CC: Suleiman Souhlal <suleiman@google.com>
---
 include/linux/memcontrol.h | 14 +++++++++++++
 include/linux/slab.h | 10 ++++++++++
 mm/memcontrol.c | 24 ++++++++++++++++++++++++
 mm/slab.h | 24 ++++++++++++++++++++----
 mm/slab_common.c | 36 ++++++++++++++++++++++++++++++++++++-----
 mm/slub.c | 16 ++++++++++++++++
 6 files changed, 111 insertions(+), 13 deletions(-)

diff --git a/include/linux/memcontrol.h b/include/linux/memcontrol.h
index 323d9e5..d9229a3 100644
--- a/include/linux/memcontrol.h
+++ b/include/linux/memcontrol.h
@@ -28,6 +28,7 @@ struct mem_cgroup;
 struct page_cgroup;
 struct page;
 struct mm_struct;
+struct kmem_cache;

 /* Stats that can be updated by kernel. */
 enum mem_cgroup_page_stat_item {
@@ -418,7 +419,20 @@ extern struct static_key memcg_kmem_enabled_key;
 bool __memcg_kmem_new_page(gfp_t gfp, void *handle, int order);
 void __memcg_kmem_commit_page(struct page *page, void *handle, int order);
 void __memcg_kmem_free_page(struct page *page, int order);
+int memcg_css_id(struct mem_cgroup *memcg);
+void memcg_register_cache(struct mem_cgroup *memcg,
+   struct kmem_cache *s);

```
+void memcg_release_cache(struct kmem_cache *cachep);
 #else
+static inline void memcg_register_cache(struct mem_cgroup *memcg,
+          struct kmem_cache *s)
+{
+}
+
+static inline void memcg_release_cache(struct kmem_cache *cachep)
+{
+}
+
 static inline void sock_update_memcg(struct sock *sk)
 {
 }
diff --git a/include/linux/slab.h b/include/linux/slab.h
index 3152bcd..9d3fd56 100644
--- a/include/linux/slab.h
+++ b/include/linux/slab.h
@@ -116,6 +116,7 @@ struct kmem_cache {
 };
 #endif

+struct mem_cgroup;
 /*
  * struct kmem_cache related prototypes
  */
@@ -125,6 +126,9 @@ int slab_is_available(void);
 struct kmem_cache *kmem_cache_create(const char *, size_t, size_t,
    unsigned long,
    void (*)(void *));
+struct kmem_cache *
+kmem_cache_create_memcg(struct mem_cgroup *, const char *, size_t, size_t,
+   unsigned long, void (*)(void *));
 void kmem_cache_destroy(struct kmem_cache *);
 int kmem_cache_shrink(struct kmem_cache *);
 void kmem_cache_free(struct kmem_cache *, void *);
@@ -337,6 +341,12 @@ extern void *__kmalloc_track_caller(size_t, gfp_t, unsigned long);
  __kmalloc(size, flags)
 #endif /* DEBUG_SLAB */

+#ifdef CONFIG_MEMCG_KMEM
+#define MAX_KMEM_CACHE_TYPES 400
+#else
+#define MAX_KMEM_CACHE_TYPES 0
+#endif /* CONFIG_MEMCG_KMEM */
+
 #ifdef CONFIG_NUMA
 /*
```

```
  * kmalloc_node_track_caller is a special version of kmalloc_node that
diff --git a/mm/memcontrol.c b/mm/memcontrol.c
index 1321a02..88bb826 100644
--- a/mm/memcontrol.c
+++ b/mm/memcontrol.c
@@ -371,6 +371,11 @@ static void memcg_kmem_clear_account_parent(struct mem_cgroup
*memcg)
 {
  clear_bit(KMEM_ACCOUNTED_PARENT, &memcg->kmem_accounted);
 }
+
+int memcg_css_id(struct mem_cgroup *memcg)
+{
+ return css_id(&memcg->css);
+}
 #endif /* CONFIG_MEMCG_KMEM */

 /* Stuffs for move charges at task migration. */
@@ -527,6 +532,24 @@ static inline bool memcg_kmem_enabled(struct mem_cgroup *memcg)
   memcg->kmem_accounted;
 }

+struct ida cache_types;
+
+void memcg_register_cache(struct mem_cgroup *memcg, struct kmem_cache *cachep)
+{
+ int id = -1;
+
+ if (!memcg)
+  id = ida_simple_get(&cache_types, 0, MAX_KMEM_CACHE_TYPES,
+        GFP_KERNEL);
+ cachep->memcg_params.id = id;
+}
+
+void memcg_release_cache(struct kmem_cache *cachep)
+{
+ if (cachep->memcg_params.id != -1)
+  ida_simple_remove(&cache_types, cachep->memcg_params.id);
+}
+
 /*
  * We need to verify if the allocation against current->mm->owner's memcg is
  * possible for the given order. But the page is not allocated yet, so we'll
@@ -5182,6 +5205,7 @@ mem_cgroup_create(struct cgroup *cont)
 #ifdef CONFIG_MEMCG_KMEM
   WARN_ON(cgroup_add_cftypes(&mem_cgroup_subsys,
         kmem_cgroup_files));
+  ida_init(&cache_types);
```

```
 #endif

   if (mem_cgroup_soft_limit_tree_init())
diff --git a/mm/slab.h b/mm/slab.h
index 124be70..3c637d2 100644
--- a/mm/slab.h
+++ b/mm/slab.h
@@ -39,12 +39,15 @@ unsigned long calculate_alignment(unsigned long flags,
 int __kmem_cache_create(struct kmem_cache *s);

 int __kmem_cache_initcall(void);
+struct mem_cgroup;
 #ifdef CONFIG_SLUB
-struct kmem_cache *__kmem_cache_alias(const char *name, size_t size,
- size_t align, unsigned long flags, void (*ctor)(void *));
+struct kmem_cache *
+__kmem_cache_alias(struct mem_cgroup *memcg, const char *name, size_t size,
+     size_t align, unsigned long flags, void (*ctor)(void *));
 #else
-static inline struct kmem_cache *__kmem_cache_alias(const char *name, size_t size,
- size_t align, unsigned long flags, void (*ctor)(void *))
+static inline struct kmem_cache *
+__kmem_cache_alias(struct mem_cgroup *memcg, const char *name, size_t size,
+     size_t align, unsigned long flags, void (*ctor)(void *))
 { return NULL; }
 #endif

@@ -56,9 +59,22 @@ static inline bool slab_is_parent(struct kmem_cache *s, struct
kmem_cache *p)
 {
  return p == s->memcg_params.parent;
 }
+
+static inline bool cache_match_memcg(struct kmem_cache *cachep,
+         struct mem_cgroup *memcg)
+{
+        return cachep->memcg_params.memcg == memcg;
+}
+
 #else
 static inline bool slab_is_parent(struct kmem_cache *s, struct kmem_cache *p)
 {
  return false;
 }
+
+static inline bool cache_match_memcg(struct kmem_cache *cachep,
+         struct mem_cgroup *memcg)
+{
```

```
+       return true;
+}
 #endif
diff --git a/mm/slab_common.c b/mm/slab_common.c
index 66df8d5..1080ef2 100644
--- a/mm/slab_common.c
+++ b/mm/slab_common.c
@@ -16,6 +16,7 @@
 #include <asm/cacheflush.h>
 #include <asm/tlbflush.h>
 #include <asm/page.h>
+#include <linux/memcontrol.h>

 #include "slab.h"

@@ -77,8 +78,9 @@ unsigned long calculate_alignment(unsigned long flags,
  * as davem.
  */

-struct kmem_cache *kmem_cache_create(const char *name, size_t size, size_t align,
-  unsigned long flags, void (*ctor)(void *))
+struct kmem_cache *
+kmem_cache_create_memcg(struct mem_cgroup *memcg, const char *name, size_t size,
+   size_t align, unsigned long flags, void (*ctor)(void *))
 {
  struct kmem_cache *s = NULL;
  char *n;
@@ -114,7 +116,7 @@ struct kmem_cache *kmem_cache_create(const char *name, size_t size,
size_t align
    continue;
   }

-  if (!strcmp(s->name, name)) {
+  if (cache_match_memcg(s, memcg) && !strcmp(s->name, name)) {
    printk(KERN_ERR "kmem_cache_create(%s): Cache name"
     " already exists.\n",
     name);
@@ -127,7 +129,7 @@ struct kmem_cache *kmem_cache_create(const char *name, size_t size,
size_t align
  WARN_ON(strchr(name, ' ')); /* It confuses parsers */
 #endif

- s = __kmem_cache_alias(name, size, align, flags, ctor);
+ s = __kmem_cache_alias(memcg, name, size, align, flags, ctor);
  if (s)
   goto oops;

@@ -148,11 +150,17 @@ struct kmem_cache *kmem_cache_create(const char *name, size_t
```

```
size, size_t align
  s->flags = flags;
  s->align = calculate_alignment(flags, align, size);

+#ifdef CONFIG_MEMCG_KMEM
+ s->memcg_params.memcg = memcg;
+#endif
+
  r = __kmem_cache_create(s);

  if (!r) {
   s->refcount = 1;
   list_add(&s->list, &slab_caches);
+  if (slab_state >= FULL)
+   memcg_register_cache(memcg, s);
  }
  else {
   kmem_cache_free(kmem_cache, s);
@@ -172,6 +180,12 @@ out:

  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_memcg(NULL, name, size, align, flags, ctor);
+}
 EXPORT_SYMBOL(kmem_cache_create);

 void kmem_cache_destroy(struct kmem_cache *s)
@@ -184,6 +198,7 @@ void kmem_cache_destroy(struct kmem_cache *s)
   if (s->flags & SLAB_DESTROY_BY_RCU)
    rcu_barrier();

+  memcg_release_cache(s);
   kfree(s->name);
   kmem_cache_free(kmem_cache, s);
  } else {
@@ -204,6 +219,17 @@ int slab_is_available(void)

 static int __init kmem_cache_initcall(void)
 {
- return __kmem_cache_initcall();
+ int r = __kmem_cache_initcall();
+#ifdef CONFIG_MEMCG_KMEM
+ struct kmem_cache *s;
+
```

```
+ if (r)
+  return r;
+ mutex_lock(&slab_mutex);
+ list_for_each_entry(s, &slab_caches, list)
+  memcg_register_cache(NULL, s);
+ mutex_unlock(&slab_mutex);
+#endif
+ return r;
 }
 __initcall(kmem_cache_initcall);
diff --git a/mm/slub.c b/mm/slub.c
index 824ea2e..417a806 100644
--- a/mm/slub.c
+++ b/mm/slub.c
@@ -31,6 +31,7 @@
 #include <linux/fault-inject.h>
 #include <linux/stacktrace.h>
 #include <linux/prefetch.h>
+#include <linux/memcontrol.h>

 #include <trace/events/kmem.h>

@@ -3819,7 +3820,7 @@ 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 *))
 {
@@ -3855,17 +3856,20 @@ static struct kmem_cache *find_mergeable(size_t size,
   if (s->size - size >= sizeof(void *))
    continue;

+  if (!cache_match_memcg(s, memcg))
+   continue;
   return s;
  }
  return NULL;
 }

-struct kmem_cache *__kmem_cache_alias(const char *name, size_t size,
-  size_t align, unsigned long flags, void (*ctor)(void *))
+struct kmem_cache *
+__kmem_cache_alias(struct mem_cgroup *memcg, const char *name, size_t size,
+     size_t align, unsigned long flags, void (*ctor)(void *))
 {
```

```
  struct kmem_cache *s;

- s = find_mergeable(size, align, flags, name, ctor);
+ s = find_mergeable(memcg, size, align, flags, name, ctor);
  if (s) {
   s->refcount++;
   /*
@@ -5195,6 +5199,10 @@ static char *create_unique_id(struct kmem_cache *s)
  if (p != name + 1)
   *p++ = '-';
  p += sprintf(p, "%07d", s->size);
+#ifdef CONFIG_MEMCG_KMEM
+ if (s->memcg_params.memcg)
+  p += sprintf(p, "-%08d", memcg_css_id(s->memcg_params.memcg));
+#endif
  BUG_ON(p > name + ID_STR_LENGTH - 1);
  return name;
 }
--
1.7.10.4
```

Subject: [PATCH 03/10] memcg: infrastructure to match an allocation to the right cache
Posted by Glauber Costa on Wed, 25 Jul 2012 14:38:14 GMT
View Forum Message <> Reply to Message

The page allocator is able to bind a page to a memcg when it is allocated. But for the caches, we'd like to have as many objects as possible in a page belonging to the same cache.

This is done in this patch by calling memcg_kmem_get_cache in the beginning of every allocation function. This routing is patched out by static branches when kernel memory controller is not being used.

It assumes that the task allocating, which determines the memcg in the page allocator, belongs to the same cgroup throughout the whole process. Misacounting can happen if the task calls memcg_kmem_get_cache() while belonging to a cgroup, and later on changes. This is considered acceptable, and should only happen upon task migration.

Before the cache is created by the memcg core, there is also a possible imbalance: the task belongs to a memcg, but the cache being allocated from is the global cache, since the child cache is not yet guaranteed to be ready. This case is also fine, since in this case the GFP_KMEMCG will not be passed and the page allocator will not attempt any cgroup accounting.

```
Signed-off-by: Glauber Costa <glommer@parallels.com>
CC: Christoph Lameter <cl@linux.com>
CC: Pekka Enberg <penberg@cs.helsinki.fi>
CC: Michal Hocko <mhocko@suse.cz>
CC: Kamezawa Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com>
CC: Johannes Weiner <hannes@cmpxchg.org>
CC: Suleiman Souhlal <suleiman@google.com>
---
 include/linux/memcontrol.h |   38 ++++++++
 init/Kconfig               |    2 +-
 mm/memcontrol.c            |  221 ++++++++++++++++++++++++++++++++++++++++++++-
 3 files changed, 259 insertions(+), 2 deletions(-)

diff --git a/include/linux/memcontrol.h b/include/linux/memcontrol.h
index d9229a3..bd1f34b 100644
--- a/include/linux/memcontrol.h
+++ b/include/linux/memcontrol.h
@@ -423,6 +423,8 @@ int memcg_css_id(struct mem_cgroup *memcg);
 void memcg_register_cache(struct mem_cgroup *memcg,
           struct kmem_cache *s);
 void memcg_release_cache(struct kmem_cache *cachep);
+struct kmem_cache *
+__memcg_kmem_get_cache(struct kmem_cache *cachep, gfp_t gfp);
 #else
 static inline void memcg_register_cache(struct mem_cgroup *memcg,
           struct kmem_cache *s)
@@ -456,6 +458,12 @@ __memcg_kmem_commit_page(struct page *page, struct mem_cgroup
*handle,
          int order)
 {
 }
+
+static inline struct kmem_cache *
+__memcg_kmem_get_cache(struct kmem_cache *cachep, gfp_t gfp)
+{
+ return cachep;
+}
 #endif /* CONFIG_MEMCG_KMEM */

 /**
@@ -515,5 +523,35 @@ void memcg_kmem_commit_page(struct page *page, struct
mem_cgroup *handle,
   if (memcg_kmem_on)
    __memcg_kmem_commit_page(page, handle, order);
 }
+
+/**
+ * memcg_kmem_get_kmem_cache: selects the correct per-memcg cache for allocation
```

```
+ * @cachep: the original global kmem cache
+ * @gfp: allocation flags.
+ *
+ * This function assumes that the task allocating, which determines the memcg
+ * in the page allocator, belongs to the same cgroup throughout the whole
+ * process.  Misacounting can happen if the task calls memcg_kmem_get_cache()
+ * while belonging to a cgroup, and later on changes. This is considered
+ * acceptable, and should only happen upon task migration.
+ *
+ * Before the cache is created by the memcg core, there is also a possible
+ * imbalance: the task belongs to a memcg, but the cache being allocated from
+ * is the global cache, since the child cache is not yet guaranteed to be
+ * ready. This case is also fine, since in this case the GFP_KMEMCG will not be
+ * passed and the page allocator will not attempt any cgroup accounting.
+ */
+static __always_inline struct kmem_cache *
+memcg_kmem_get_cache(struct kmem_cache *cachep, gfp_t gfp)
+{
+ if (!memcg_kmem_on)
+  return cachep;
+ if (gfp & __GFP_NOFAIL)
+  return cachep;
+ if (in_interrupt() || (!current->mm) || (current->flags & PF_KTHREAD))
+  return cachep;
+
+ return __memcg_kmem_get_cache(cachep, gfp);
+}
 #endif /* _LINUX_MEMCONTROL_H */

diff --git a/init/Kconfig b/init/Kconfig
index 547bd10..610cfd3 100644
--- a/init/Kconfig
+++ b/init/Kconfig
@@ -741,7 +741,7 @@ config MEMCG_SWAP_ENABLED
    then swapaccount=0 does the trick).
 config MEMCG_KMEM
  bool "Memory Resource Controller Kernel Memory accounting (EXPERIMENTAL)"
- depends on MEMCG && EXPERIMENTAL
+ depends on MEMCG && EXPERIMENTAL && !SLOB
  default n
  help
    The Kernel Memory extension for Memory Resource Controller can limit
diff --git a/mm/memcontrol.c b/mm/memcontrol.c
index 88bb826..8d012c7 100644
--- a/mm/memcontrol.c
+++ b/mm/memcontrol.c
@@ -14,6 +14,10 @@
  * Copyright (C) 2012 Parallels Inc. and Google Inc.
```

```
  * Authors: Glauber Costa and Suleiman Souhlal
  *
+ * Kernel Memory Controller
+ * Copyright (C) 2012 Parallels Inc. and Google Inc.
+ * Authors: Glauber Costa and Suleiman Souhlal
+ *
  * This program is free software; you can redistribute it and/or modify
  * it under the terms of the GNU General Public License as published by
  * the Free Software Foundation; either version 2 of the License, or
@@ -339,6 +343,11 @@ struct mem_cgroup {
 #ifdef CONFIG_INET
  struct tcp_memcontrol tcp_mem;
 #endif
+
+#ifdef CONFIG_MEMCG_KMEM
+ /* Slab accounting */
+ struct kmem_cache *slabs[MAX_KMEM_CACHE_TYPES];
+#endif
 };

 enum {
@@ -532,6 +541,40 @@ static inline bool memcg_kmem_enabled(struct mem_cgroup *memcg)
   memcg->kmem_accounted;
 }

+static char *memcg_cache_name(struct mem_cgroup *memcg, struct kmem_cache *cachep)
+{
+ char *name;
+ struct dentry *dentry;
+
+ rcu_read_lock();
+ dentry = rcu_dereference(memcg->css.cgroup->dentry);
+ rcu_read_unlock();
+
+ BUG_ON(dentry == NULL);
+
+ name = kasprintf(GFP_KERNEL, "%s(%d:%s)",
+     cachep->name, css_id(&memcg->css), dentry->d_name.name);
+
+ return name;
+}
+
+static struct kmem_cache *kmem_cache_dup(struct mem_cgroup *memcg,
+      struct kmem_cache *s)
+{
+ char *name;
+ struct kmem_cache *new;
+
```

```
+ name = memcg_cache_name(memcg, s);
+ if (!name)
+  return NULL;
+
+ new = kmem_cache_create_memcg(memcg, name, s->object_size, s->align,
+         (s->flags & ~SLAB_PANIC), s->ctor);
+
+ kfree(name);
+ return new;
+}
+
 struct ida cache_types;

 void memcg_register_cache(struct mem_cgroup *memcg, struct kmem_cache *cachep)
@@ -656,6 +699,14 @@ void __memcg_kmem_free_page(struct page *page, int order)
 }
 EXPORT_SYMBOL(__memcg_kmem_free_page);

+static void memcg_slab_init(struct mem_cgroup *memcg)
+{
+ int i;
+
+ for (i = 0; i < MAX_KMEM_CACHE_TYPES; i++)
+  memcg->slabs[i] = NULL;
+}
+
 static void disarm_kmem_keys(struct mem_cgroup *memcg)
 {
  if (test_bit(KMEM_ACCOUNTED_THIS, &memcg->kmem_accounted))
@@ -666,6 +717,170 @@ static void disarm_kmem_keys(struct mem_cgroup *memcg)
   */
  WARN_ON(res_counter_read_u64(&memcg->kmem, RES_USAGE) != 0);
 }
+
+static DEFINE_MUTEX(memcg_cache_mutex);
+static struct kmem_cache *memcg_create_kmem_cache(struct mem_cgroup *memcg,
+        struct kmem_cache *cachep)
+{
+ struct kmem_cache *new_cachep;
+ int idx;
+
+ BUG_ON(!memcg_kmem_enabled(memcg));
+
+ idx = cachep->memcg_params.id;
+
+ mutex_lock(&memcg_cache_mutex);
+ new_cachep = memcg->slabs[idx];
+ if (new_cachep)
```

```
+  goto out;
+
+ new_cachep = kmem_cache_dup(memcg, cachep);
+
+ if (new_cachep == NULL) {
+  new_cachep = cachep;
+  goto out;
+ }
+
+ mem_cgroup_get(memcg);
+ memcg->slabs[idx] = new_cachep;
+ new_cachep->memcg_params.memcg = memcg;
+out:
+ mutex_unlock(&memcg_cache_mutex);
+ return new_cachep;
+}
+
+struct create_work {
+ struct mem_cgroup *memcg;
+ struct kmem_cache *cachep;
+ struct list_head list;
+};
+
+/* Use a single spinlock for destruction and creation, not a frequent op */
+static DEFINE_SPINLOCK(cache_queue_lock);
+static LIST_HEAD(create_queue);
+
+/*
+ * Flush the queue of kmem_caches to create, because we're creating a cgroup.
+ *
+ * We might end up flushing other cgroups' creation requests as well, but
+ * they will just get queued again next time someone tries to make a slab
+ * allocation for them.
+ */
+void memcg_flush_cache_create_queue(void)
+{
+ struct create_work *cw, *tmp;
+ unsigned long flags;
+
+ spin_lock_irqsave(&cache_queue_lock, flags);
+ list_for_each_entry_safe(cw, tmp, &create_queue, list) {
+  list_del(&cw->list);
+  kfree(cw);
+ }
+ spin_unlock_irqrestore(&cache_queue_lock, flags);
+}
+
+static void memcg_create_cache_work_func(struct work_struct *w)
```

```
+{
+ struct create_work *cw, *tmp;
+ unsigned long flags;
+ LIST_HEAD(create_unlocked);
+
+ spin_lock_irqsave(&cache_queue_lock, flags);
+ list_for_each_entry_safe(cw, tmp, &create_queue, list)
+  list_move(&cw->list, &create_unlocked);
+ spin_unlock_irqrestore(&cache_queue_lock, flags);
+
+ list_for_each_entry_safe(cw, tmp, &create_unlocked, list) {
+  list_del(&cw->list);
+  memcg_create_kmem_cache(cw->memcg, cw->cachep);
+  /* Drop the reference gotten when we enqueued. */
+  css_put(&cw->memcg->css);
+  kfree(cw);
+ }
+}
+
+static DECLARE_WORK(memcg_create_cache_work, memcg_create_cache_work_func);
+
+/*
+ * Enqueue the creation of a per-memcg kmem_cache.
+ * Called with rcu_read_lock.
+ */
+static void memcg_create_cache_enqueue(struct mem_cgroup *memcg,
+          struct kmem_cache *cachep)
+{
+ struct create_work *cw;
+ unsigned long flags;
+
+ spin_lock_irqsave(&cache_queue_lock, flags);
+ list_for_each_entry(cw, &create_queue, list) {
+  if (cw->memcg == memcg && cw->cachep == cachep) {
+   spin_unlock_irqrestore(&cache_queue_lock, flags);
+   return;
+  }
+ }
+ spin_unlock_irqrestore(&cache_queue_lock, flags);
+
+ /* The corresponding put will be done in the workqueue. */
+ if (!css_tryget(&memcg->css))
+  return;
+
+ cw = kmalloc(sizeof(struct create_work), GFP_NOWAIT);
+ if (cw == NULL) {
+  css_put(&memcg->css);
+  return;
```

```
+ }
+
+ cw->memcg = memcg;
+ cw->cachep = cachep;
+ spin_lock_irqsave(&cache_queue_lock, flags);
+ list_add_tail(&cw->list, &create_queue);
+ spin_unlock_irqrestore(&cache_queue_lock, flags);
+
+ schedule_work(&memcg_create_cache_work);
+}
+
+/*
+ * Return the kmem_cache we're supposed to use for a slab allocation.
+ * We try to use the current memcg's version of the cache.
+ *
+ * If the cache does not exist yet, if we are the first user of it,
+ * we either create it immediately, if possible, or create it asynchronously
+ * in a workqueue.
+ * In the latter case, we will let the current allocation go through with
+ * the original cache.
+ *
+ * Can't be called in interrupt context or from kernel threads.
+ * This function needs to be called with rcu_read_lock() held.
+ */
+struct kmem_cache *__memcg_kmem_get_cache(struct kmem_cache *cachep,
+       gfp_t gfp)
+{
+ struct mem_cgroup *memcg;
+ int idx;
+ struct task_struct *p;
+
+ if (cachep->memcg_params.memcg)
+  return cachep;
+
+ idx = cachep->memcg_params.id;
+ VM_BUG_ON(idx == -1);
+
+ rcu_read_lock();
+ p = rcu_dereference(current->mm->owner);
+ memcg = mem_cgroup_from_task(p);
+ rcu_read_unlock();
+
+ if (!memcg_kmem_enabled(memcg))
+  return cachep;
+
+ if (memcg->slabs[idx] == NULL) {
+  memcg_create_cache_enqueue(memcg, cachep);
+  return cachep;
```

```
+ }
+
+ return memcg->slabs[idx];
+}
+EXPORT_SYMBOL(__memcg_kmem_get_cache);
 #else
 static void disarm_kmem_keys(struct mem_cgroup *memcg)
 {
@@ -4860,7 +5075,11 @@ static struct cftype kmem_cgroup_files[] = {

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

 static void kmem_cgroup_destroy(struct mem_cgroup *memcg)
--
1.7.10.4
```

Subject: [PATCH 04/10] memcg: skip memcg kmem allocations in specified code regions

Posted by Glauber Costa on Wed, 25 Jul 2012 14:38:15 GMT

View Forum Message <> Reply to Message

This patch creates a mechanism that skip memcg allocations during
certain pieces of our core code. It basically works in the same way
as preempt_disable()/preempt_enable(): By marking a region under
which all allocations will be accounted to the root memcg.

We need this to prevent races in early cache creation, when we
allocate data using caches that are not necessarily created already.

Signed-off-by: Glauber Costa <glommer@parallels.com>
CC: Christoph Lameter <cl@linux.com>
CC: Pekka Enberg <penberg@cs.helsinki.fi>
CC: Michal Hocko <mhocko@suse.cz>
CC: Kamezawa Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com>
CC: Johannes Weiner <hannes@cmpxchg.org>
CC: Suleiman Souhlal <suleiman@google.com>
---
 include/linux/sched.h | 1 +
 mm/memcontrol.c | 27 +++++++++++++++++++++++++++

```
 2 files changed, 28 insertions(+)

diff --git a/include/linux/sched.h b/include/linux/sched.h
index c350e60..c09cd1b 100644
--- a/include/linux/sched.h
+++ b/include/linux/sched.h
@@ -1562,6 +1562,7 @@ struct task_struct {
   unsigned long nr_pages; /* uncharged usage */
   unsigned long memsw_nr_pages; /* uncharged mem+swap usage */
  } memcg_batch;
+ unsigned int memcg_kmem_skip_account;
 #endif
 #ifdef CONFIG_HAVE_HW_BREAKPOINT
  atomic_t ptrace_bp_refcnt;
diff --git a/mm/memcontrol.c b/mm/memcontrol.c
index 8d012c7..854f6cc 100644
--- a/mm/memcontrol.c
+++ b/mm/memcontrol.c
@@ -541,6 +541,22 @@ static inline bool memcg_kmem_enabled(struct mem_cgroup *memcg)
   memcg->kmem_accounted;
 }

+static void memcg_stop_kmem_account(void)
+{
+ if (!current->mm)
+  return;
+
+ current->memcg_kmem_skip_account++;
+}
+
+static void memcg_resume_kmem_account(void)
+{
+ if (!current->mm)
+  return;
+
+ current->memcg_kmem_skip_account--;
+}
+
 static char *memcg_cache_name(struct mem_cgroup *memcg, struct kmem_cache *cachep)
 {
  char *name;
@@ -614,6 +630,10 @@ bool __memcg_kmem_new_page(gfp_t gfp, void *_handle, int order)
  struct task_struct *p;

  *handle = NULL;
+
+ if (current->memcg_kmem_skip_account)
+  return true;
```

```
+
  rcu_read_lock();
  p = rcu_dereference(current->mm->owner);
  memcg = mem_cgroup_from_task(p);
@@ -734,7 +754,9 @@ static struct kmem_cache *memcg_create_kmem_cache(struct
mem_cgroup *memcg,
  if (new_cachep)
   goto out;

+ memcg_stop_kmem_account();
  new_cachep = kmem_cache_dup(memcg, cachep);
+ memcg_resume_kmem_account();

  if (new_cachep == NULL) {
   new_cachep = cachep;
@@ -824,7 +846,9 @@ static void memcg_create_cache_enqueue(struct mem_cgroup *memcg,
  if (!css_tryget(&memcg->css))
   return;

+ memcg_stop_kmem_account();
  cw = kmalloc(sizeof(struct create_work), GFP_NOWAIT);
+ memcg_resume_kmem_account();
  if (cw == NULL) {
   css_put(&memcg->css);
   return;
@@ -859,6 +883,9 @@ struct kmem_cache *__memcg_kmem_get_cache(struct kmem_cache
*cachep,
  int idx;
  struct task_struct *p;

+ if (!current->mm || current->memcg_kmem_skip_account)
+  return cachep;
+
  if (cachep->memcg_params.memcg)
   return cachep;

--
1.7.10.4
```

Subject: [PATCH 05/10] slab: allow enable_cpu_cache to use preset values for its tunables

Posted by Glauber Costa on Wed, 25 Jul 2012 14:38:16 GMT

View Forum Message <> Reply to Message

SLAB allows us to tune a particular cache behavior with tunables.
When creating a new memcg cache copy, we'd like to preserve any tunables
the parent cache already had.

This could be done by an explicit call to do_tune_cpucache() after the
cache is created. But this is not very convenient now that the caches are
created from common code, since this function is SLAB-specific.

Another method of doing that is taking advantage of the fact that
do_tune_cpucache() is always called from enable_cpucache(), which is
called at cache initialization. We can just preset the values, and
then things work as expected.

Signed-off-by: Glauber Costa <glommer@parallels.com>
CC: Christoph Lameter <cl@linux.com>
CC: Pekka Enberg <penberg@cs.helsinki.fi>
CC: Michal Hocko <mhocko@suse.cz>
CC: Kamezawa Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com>
CC: Johannes Weiner <hannes@cmpxchg.org>
CC: Suleiman Souhlal <suleiman@google.com>
---
 include/linux/slab.h | 3 ++-
 mm/memcontrol.c | 2 +-
 mm/slab.c | 19 +++++++++++++++---
 mm/slab_common.c | 7 ++++---
 4 files changed, 23 insertions(+), 8 deletions(-)

diff --git a/include/linux/slab.h b/include/linux/slab.h
index 9d3fd56..249a0d3 100644
--- a/include/linux/slab.h
+++ b/include/linux/slab.h
@@ -128,7 +128,7 @@ struct kmem_cache *kmem_cache_create(const char *, size_t, size_t,
    void (*)(void *));
 struct kmem_cache *
 kmem_cache_create_memcg(struct mem_cgroup *, const char *, size_t, size_t,
-   unsigned long, void (*)(void *));
+   unsigned long, void (*)(void *), struct kmem_cache *);
 void kmem_cache_destroy(struct kmem_cache *);
 int kmem_cache_shrink(struct kmem_cache *);
 void kmem_cache_free(struct kmem_cache *, void *);
@@ -184,6 +184,7 @@ unsigned int kmem_cache_size(struct kmem_cache *);
 #ifdef CONFIG_MEMCG_KMEM
 struct mem_cgroup_cache_params {
  struct mem_cgroup *memcg;
+ struct kmem_cache *parent;
  int id;
 };
 #endif
diff --git a/mm/memcontrol.c b/mm/memcontrol.c
index 854f6cc..b933474 100644
--- a/mm/memcontrol.c

```
+++ b/mm/memcontrol.c
@@ -585,7 +585,7 @@ static struct kmem_cache *kmem_cache_dup(struct mem_cgroup
*memcg,
   return NULL;

  new = kmem_cache_create_memcg(memcg, name, s->object_size, s->align,
-         (s->flags & ~SLAB_PANIC), s->ctor);
+         (s->flags & ~SLAB_PANIC), s->ctor, s);

  kfree(name);
  return new;
diff --git a/mm/slab.c b/mm/slab.c
index 7e7ec59..76bc98f 100644
--- a/mm/slab.c
+++ b/mm/slab.c
@@ -3916,8 +3916,19 @@ static int do_tune_cpucache(struct kmem_cache *cachep, int limit,
 static int enable_cpucache(struct kmem_cache *cachep, gfp_t gfp)
 {
  int err;
- int limit, shared;
-
+ int limit = 0;
+ int shared = 0;
+ int batchcount = 0;
+
+#ifdef CONFIG_MEMCG_KMEM
+       if (cachep->memcg_params.parent) {
+               limit = cachep->memcg_params.parent->limit;
+               shared = cachep->memcg_params.parent->shared;
+               batchcount = cachep->memcg_params.parent->batchcount;
+       }
+#endif
+ if (limit && shared && batchcount)
+  goto skip_setup;
   /*
    * The head array serves three purposes:
    * - create a LIFO ordering, i.e. return objects that are cache-warm
@@ -3959,7 +3970,9 @@ static int enable_cpucache(struct kmem_cache *cachep, gfp_t gfp)
  if (limit > 32)
   limit = 32;
 #endif
- err = do_tune_cpucache(cachep, limit, (limit + 1) / 2, shared, gfp);
+ batchcount = (limit + 1) / 2;
+skip_setup:
+ err = do_tune_cpucache(cachep, limit, batchcount, shared, gfp);
  if (err)
   printk(KERN_ERR "enable_cpucache failed for %s, error %d.\n",
          cachep->name, -err);
```

```
diff --git a/mm/slab_common.c b/mm/slab_common.c
index 1080ef2..562146b 100644
--- a/mm/slab_common.c
+++ b/mm/slab_common.c
@@ -80,7 +80,8 @@ unsigned long calculate_alignment(unsigned long flags,

 struct kmem_cache *
 kmem_cache_create_memcg(struct mem_cgroup *memcg, const char *name, size_t size,
-   size_t align, unsigned long flags, void (*ctor)(void *))
+   size_t align, unsigned long flags, void (*ctor)(void *),
+   struct kmem_cache *parent_cache)
 {
  struct kmem_cache *s = NULL;
  char *n;
@@ -149,9 +150,9 @@ kmem_cache_create_memcg(struct mem_cgroup *memcg, const char
*name, size_t size,
  s->ctor = ctor;
  s->flags = flags;
  s->align = calculate_alignment(flags, align, size);
-
 #ifdef CONFIG_MEMCG_KMEM
  s->memcg_params.memcg = memcg;
+ s->memcg_params.parent = parent_cache;
 #endif

  r = __kmem_cache_create(s);
@@ -184,7 +185,7 @@ out:
 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_memcg(NULL, name, size, align, flags, ctor);
+ return kmem_cache_create_memcg(NULL, name, size, align, flags, ctor, NULL);
 }
 EXPORT_SYMBOL(kmem_cache_create);

--
1.7.10.4
```

Subject: [PATCH 06/10] sl[au]b: Allocate objects from memcg cache
Posted by Glauber Costa on Wed, 25 Jul 2012 14:38:17 GMT
View Forum Message <> Reply to Message

We are able to match a cache allocation to a particular memcg. If the task doesn't change groups during the allocation itself - a rare event, this will give us a good picture about who is the first group to touch a cache page.

This patch uses the now available infrastructure by calling
memcg_kmem_get_cache() before all the cache allocations.

Signed-off-by: Glauber Costa <glommer@parallels.com>
CC: Christoph Lameter <cl@linux.com>
CC: Pekka Enberg <penberg@cs.helsinki.fi>
CC: Michal Hocko <mhocko@suse.cz>
CC: Kamezawa Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com>
CC: Johannes Weiner <hannes@cmpxchg.org>
CC: Suleiman Souhlal <suleiman@google.com>
---
 include/linux/slub_def.h | 18 ++++++++++++-----
 mm/memcontrol.c | 2 ++
 mm/slab.c | 4 ++++
 mm/slub.c | 1 +
 4 files changed, 20 insertions(+), 5 deletions(-)

diff --git a/include/linux/slub_def.h b/include/linux/slub_def.h
index 8bb8ad2..148000a 100644
--- a/include/linux/slub_def.h
+++ b/include/linux/slub_def.h
@@ -13,6 +13,8 @@
 #include <linux/kobject.h>

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

 enum stat_item {
  ALLOC_FASTPATH, /* Allocation from cpu slab */
@@ -209,14 +211,14 @@ static __always_inline int kmalloc_index(size_t size)
  * This ought to end up with a global pointer to the right cache
  * in kmalloc_caches.
  */
-static __always_inline struct kmem_cache *kmalloc_slab(size_t size)
+static __always_inline struct kmem_cache *kmalloc_slab(gfp_t flags, size_t size)
 {
  int index = kmalloc_index(size);

  if (index == 0)
   return NULL;

- return kmalloc_caches[index];
+ return memcg_kmem_get_cache(kmalloc_caches[index], flags);
 }

 void *kmem_cache_alloc(struct kmem_cache *, gfp_t);
@@ -225,7 +227,13 @@ void *__kmalloc(size_t size, gfp_t flags);

```
 static __always_inline void *
 kmalloc_order(size_t size, gfp_t flags, unsigned int order)
 {
- void *ret = (void *) __get_free_pages(flags | __GFP_COMP, order);
+ void *ret;
+
+ flags = __GFP_COMP;
+#ifdef CONFIG_MEMCG_KMEM
+ flags |= __GFP_KMEMCG;
+#endif
+ ret = (void *) __get_free_pages(flags, order);
  kmemleak_alloc(ret, size, 1, flags);
  return ret;
 }
@@ -274,7 +282,7 @@ static __always_inline void *kmalloc(size_t size, gfp_t flags)
    return kmalloc_large(size, flags);

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

    if (!s)
     return ZERO_SIZE_PTR;
@@ -307,7 +315,7 @@ static __always_inline void *kmalloc_node(size_t size, gfp_t flags, int
node)
 {
  if (__builtin_constant_p(size) &&
   size <= SLUB_MAX_SIZE && !(flags & SLUB_DMA)) {
-   struct kmem_cache *s = kmalloc_slab(size);
+   struct kmem_cache *s = kmalloc_slab(flags, size);

   if (!s)
    return ZERO_SIZE_PTR;
diff --git a/mm/memcontrol.c b/mm/memcontrol.c
index b933474..2cc3acf 100644
--- a/mm/memcontrol.c
+++ b/mm/memcontrol.c
@@ -586,6 +586,8 @@ static struct kmem_cache *kmem_cache_dup(struct mem_cgroup
*memcg,

  new = kmem_cache_create_memcg(memcg, name, s->object_size, s->align,
         (s->flags & ~SLAB_PANIC), s->ctor, s);
+ if (new)
+  new->allocflags |= __GFP_KMEMCG;

  kfree(name);
  return new;
diff --git a/mm/slab.c b/mm/slab.c
```

```
index 76bc98f..ddc60a4 100644
--- a/mm/slab.c
+++ b/mm/slab.c
@@ -3316,6 +3316,8 @@ __cache_alloc_node(struct kmem_cache *cachep, gfp_t flags, int
nodeid,
  if (slab_should_failslab(cachep, flags))
   return NULL;

+ cachep = memcg_kmem_get_cache(cachep, flags);
+
  cache_alloc_debugcheck_before(cachep, flags);
  local_irq_save(save_flags);

@@ -3401,6 +3403,8 @@ __cache_alloc(struct kmem_cache *cachep, gfp_t flags, void *caller)
  if (slab_should_failslab(cachep, flags))
   return NULL;

+ cachep = memcg_kmem_get_cache(cachep, flags);
+
  cache_alloc_debugcheck_before(cachep, flags);
  local_irq_save(save_flags);
  objp = __do_cache_alloc(cachep, flags);
diff --git a/mm/slub.c b/mm/slub.c
index 417a806..6175a72 100644
--- a/mm/slub.c
+++ b/mm/slub.c
@@ -2292,6 +2292,7 @@ static __always_inline void *slab_alloc(struct kmem_cache *s,
  if (slab_pre_alloc_hook(s, gfpflags))
   return NULL;

+ s = memcg_kmem_get_cache(s, gfpflags);
 redo:

  /*
--
1.7.10.4
```

Subject: [PATCH 07/10] memcg: destroy memcg caches
Posted by Glauber Costa on Wed, 25 Jul 2012 14:38:18 GMT
View Forum Message <> Reply to Message

This patch implements destruction of memcg caches. Right now,
only caches where our reference counter is the last remaining are
deleted. If there are any other reference counters around, we just
leave the caches lying around until they go away.

When that happen, a destruction function is called from the cache

code. Caches are only destroyed in process context, so we queue them
up for later processing in the general case.

Signed-off-by: Glauber Costa <glommer@parallels.com>
CC: Christoph Lameter <cl@linux.com>
CC: Pekka Enberg <penberg@cs.helsinki.fi>
CC: Michal Hocko <mhocko@suse.cz>
CC: Kamezawa Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com>
CC: Johannes Weiner <hannes@cmpxchg.org>
CC: Suleiman Souhlal <suleiman@google.com>
---
 include/linux/memcontrol.h | 1 +
 include/linux/slab.h | 3 ++
 mm/memcontrol.c | 84 ++
 mm/slab.c | 4 +++
 mm/slab.h | 21 ++++++++++
 mm/slub.c | 7 +++-
 6 files changed, 119 insertions(+), 1 deletion(-)

diff --git a/include/linux/memcontrol.h b/include/linux/memcontrol.h
index bd1f34b..247019f 100644
--- a/include/linux/memcontrol.h
+++ b/include/linux/memcontrol.h
@@ -425,6 +425,7 @@ void memcg_register_cache(struct mem_cgroup *memcg,
 void memcg_release_cache(struct kmem_cache *cachep);
 struct kmem_cache *
 __memcg_kmem_get_cache(struct kmem_cache *cachep, gfp_t gfp);
+void mem_cgroup_destroy_cache(struct kmem_cache *cachep);
 #else
 static inline void memcg_register_cache(struct mem_cgroup *memcg,
                                         struct kmem_cache *s)
diff --git a/include/linux/slab.h b/include/linux/slab.h
index 249a0d3..b9310a4 100644
--- a/include/linux/slab.h
+++ b/include/linux/slab.h
@@ -186,6 +186,9 @@ struct mem_cgroup_cache_params {
   struct mem_cgroup *memcg;
   struct kmem_cache *parent;
   int id;
+ bool dead;
+ atomic_t nr_pages;
+ struct list_head destroyed_list; /* Used when deleting memcg cache */
 };
 #endif

diff --git a/mm/memcontrol.c b/mm/memcontrol.c
index 2cc3acf..1231d86 100644
--- a/mm/memcontrol.c

```
+++ b/mm/memcontrol.c
@@ -599,6 +599,8 @@ void memcg_register_cache(struct mem_cgroup *memcg, struct
kmem_cache *cachep)
 {
  int id = -1;

+ INIT_LIST_HEAD(&cachep->memcg_params.destroyed_list);
+
  if (!memcg)
   id = ida_simple_get(&cache_types, 0, MAX_KMEM_CACHE_TYPES,
        GFP_KERNEL);
@@ -768,6 +770,7 @@ static struct kmem_cache *memcg_create_kmem_cache(struct
mem_cgroup *memcg,
  mem_cgroup_get(memcg);
  memcg->slabs[idx] = new_cachep;
  new_cachep->memcg_params.memcg = memcg;
+ atomic_set(&new_cachep->memcg_params.nr_pages , 0);
 out:
  mutex_unlock(&memcg_cache_mutex);
  return new_cachep;
@@ -782,6 +785,55 @@ struct create_work {
 /* Use a single spinlock for destruction and creation, not a frequent op */
 static DEFINE_SPINLOCK(cache_queue_lock);
 static LIST_HEAD(create_queue);
+static LIST_HEAD(destroyed_caches);
+
+static void kmem_cache_destroy_work_func(struct work_struct *w)
+{
+ struct kmem_cache *cachep;
+ struct mem_cgroup_cache_params *p, *tmp;
+ unsigned long flags;
+ LIST_HEAD(del_unlocked);
+
+ spin_lock_irqsave(&cache_queue_lock, flags);
+ list_for_each_entry_safe(p, tmp, &destroyed_caches, destroyed_list) {
+  cachep = container_of(p, struct kmem_cache, memcg_params);
+  list_move(&cachep->memcg_params.destroyed_list, &del_unlocked);
+ }
+ spin_unlock_irqrestore(&cache_queue_lock, flags);
+
+ list_for_each_entry_safe(p, tmp, &del_unlocked, destroyed_list) {
+  cachep = container_of(p, struct kmem_cache, memcg_params);
+  list_del(&cachep->memcg_params.destroyed_list);
+  if (!atomic_read(&cachep->memcg_params.nr_pages)) {
+   mem_cgroup_put(cachep->memcg_params.memcg);
+   kmem_cache_destroy(cachep);
+  }
+ }
```

```
+}
+static DECLARE_WORK(kmem_cache_destroy_work, kmem_cache_destroy_work_func);
+
+static void __mem_cgroup_destroy_cache(struct kmem_cache *cachep)
+{
+ BUG_ON(cachep->memcg_params.id != -1);
+ list_add(&cachep->memcg_params.destroyed_list, &destroyed_caches);
+}
+
+void mem_cgroup_destroy_cache(struct kmem_cache *cachep)
+{
+ unsigned long flags;
+
+ if (!cachep->memcg_params.dead)
+  return;
+ /*
+  * We have to defer the actual destroying to a workqueue, because
+  * we might currently be in a context that cannot sleep.
+  */
+ spin_lock_irqsave(&cache_queue_lock, flags);
+ __mem_cgroup_destroy_cache(cachep);
+ spin_unlock_irqrestore(&cache_queue_lock, flags);
+
+ schedule_work(&kmem_cache_destroy_work);
+}

 /*
  * Flush the queue of kmem_caches to create, because we're creating a cgroup.
@@ -803,6 +855,33 @@ void memcg_flush_cache_create_queue(void)
  spin_unlock_irqrestore(&cache_queue_lock, flags);
 }

+static void mem_cgroup_destroy_all_caches(struct mem_cgroup *memcg)
+{
+ struct kmem_cache *cachep;
+ unsigned long flags;
+ int i;
+
+ /*
+  * pre_destroy() gets called with no tasks in the cgroup.
+  * this means that after flushing the create queue, no more caches
+  * will appear
+  */
+ memcg_flush_cache_create_queue();
+
+ spin_lock_irqsave(&cache_queue_lock, flags);
+ for (i = 0; i < MAX_KMEM_CACHE_TYPES; i++) {
+  cachep = memcg->slabs[i];
```

```
+  if (!cachep)
+   continue;
+
+  cachep->memcg_params.dead = true;
+  __mem_cgroup_destroy_cache(cachep);
+ }
+ spin_unlock_irqrestore(&cache_queue_lock, flags);
+
+ schedule_work(&kmem_cache_destroy_work);
+}
+
 static void memcg_create_cache_work_func(struct work_struct *w)
 {
  struct create_work *cw, *tmp;
@@ -914,6 +993,10 @@ EXPORT_SYMBOL(__memcg_kmem_get_cache);
 static void disarm_kmem_keys(struct mem_cgroup *memcg)
 {
 }
+
+static inline void mem_cgroup_destroy_all_caches(struct mem_cgroup *memcg)
+{
+}
 #endif /* CONFIG_MEMCG_KMEM */

 #if defined(CONFIG_INET) && defined(CONFIG_MEMCG_KMEM)
@@ -5517,6 +5600,7 @@ static int mem_cgroup_pre_destroy(struct cgroup *cont)
 {
  struct mem_cgroup *memcg = mem_cgroup_from_cont(cont);

+ mem_cgroup_destroy_all_caches(memcg);
  return mem_cgroup_force_empty(memcg, false);
 }

diff --git a/mm/slab.c b/mm/slab.c
index ddc60a4..21d7cf7 100644
--- a/mm/slab.c
+++ b/mm/slab.c
@@ -1769,6 +1769,8 @@ static void *kmem_getpages(struct kmem_cache *cachep, gfp_t flags,
int nodeid)
  for (i = 0; i < nr_pages; i++)
   __SetPageSlab(page + i);

+ mem_cgroup_bind_pages(cachep, cachep->gfporder);
+
  if (kmemcheck_enabled && !(cachep->flags & SLAB_NOTRACK)) {
   kmemcheck_alloc_shadow(page, cachep->gfporder, flags, nodeid);

@@ -1803,6 +1805,8 @@ static void kmem_freepages(struct kmem_cache *cachep, void *addr)
```

```
   __ClearPageSlab(page);
   page++;
  }
+
+ mem_cgroup_release_pages(cachep, cachep->gfporder);
  if (current->reclaim_state)
   current->reclaim_state->reclaimed_slab += nr_freed;
  free_pages((unsigned long)addr, cachep->gfporder);
diff --git a/mm/slab.h b/mm/slab.h
index 3c637d2..d9df178 100644
--- a/mm/slab.h
+++ b/mm/slab.h
@@ -1,5 +1,6 @@
 #ifndef MM_SLAB_H
 #define MM_SLAB_H
+#include <linux/memcontrol.h>
 /*
  * Internal slab definitions
  */
@@ -66,6 +67,19 @@ static inline bool cache_match_memcg(struct kmem_cache *cachep,
        return cachep->memcg_params.memcg == memcg;
 }

+static inline void mem_cgroup_bind_pages(struct kmem_cache *s, int order)
+{
+ if (s->memcg_params.id == -1)
+  atomic_add(1 << order, &s->memcg_params.nr_pages);
+}
+
+static inline void mem_cgroup_release_pages(struct kmem_cache *s, int order)
+{
+ if (s->memcg_params.id != -1)
+  return;
+ if (atomic_sub_and_test((1 << order), &s->memcg_params.nr_pages))
+  mem_cgroup_destroy_cache(s);
+}
 #else
 static inline bool slab_is_parent(struct kmem_cache *s, struct kmem_cache *p)
 {
@@ -77,4 +91,11 @@ static inline bool cache_match_memcg(struct kmem_cache *cachep,
 {
        return true;
 }
+
+static inline void mem_cgroup_bind_pages(struct kmem_cache *s, int order)
+{
+}
+static inline void mem_cgroup_release_pages(struct kmem_cache *s, int order)
```

```
+{
+}
 #endif
diff --git a/mm/slub.c b/mm/slub.c
index 6175a72..fdb1a0d 100644
--- a/mm/slub.c
+++ b/mm/slub.c
@@ -1344,6 +1344,7 @@ static struct page *new_slab(struct kmem_cache *s, gfp_t flags, int
node)
  void *start;
  void *last;
  void *p;
+ int order;

  BUG_ON(flags & GFP_SLAB_BUG_MASK);

@@ -1352,14 +1353,16 @@ static struct page *new_slab(struct kmem_cache *s, gfp_t flags, int
node)
  if (!page)
   goto out;

+ order = compound_order(page);
  inc_slabs_node(s, page_to_nid(page), page->objects);
+ mem_cgroup_bind_pages(s, order);
  page->slab = s;
  __SetPageSlab(page);

  start = page_address(page);

  if (unlikely(s->flags & SLAB_POISON))
-  memset(start, POISON_INUSE, PAGE_SIZE << compound_order(page));
+  memset(start, POISON_INUSE, PAGE_SIZE << order);

  last = start;
  for_each_object(p, s, start, page->objects) {
@@ -1399,6 +1402,8 @@ static void __free_slab(struct kmem_cache *s, struct page *page)
   -pages);

  __ClearPageSlab(page);
+
+ mem_cgroup_release_pages(s, order);
  reset_page_mapcount(page);
  if (current->reclaim_state)
   current->reclaim_state->reclaimed_slab += pages;
--
1.7.10.4
```

Subject: [PATCH 08/10] memcg/sl[au]b Track all the memcg children of a kmem_cache.
Posted by Glauber Costa on Wed, 25 Jul 2012 14:38:19 GMT
View Forum Message <> Reply to Message

This enables us to remove all the children of a kmem_cache being
destroyed, if for example the kernel module it's being used in
gets unloaded. Otherwise, the children will still point to the
destroyed parent.

Signed-off-by: Suleiman Souhlal <suleiman@google.com>
Signed-off-by: Glauber Costa <glommer@parallels.com>
CC: Christoph Lameter <cl@linux.com>
CC: Pekka Enberg <penberg@cs.helsinki.fi>
CC: Michal Hocko <mhocko@suse.cz>
CC: Kamezawa Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com>
CC: Johannes Weiner <hannes@cmpxchg.org>
---
 include/linux/memcontrol.h | 1 +
 include/linux/slab.h | 1 +
 mm/memcontrol.c | 16 ++++++++++++++-
 mm/slab_common.c | 31 +++++++++++++++++++++++++++++++
 4 files changed, 48 insertions(+), 1 deletion(-)

diff --git a/include/linux/memcontrol.h b/include/linux/memcontrol.h
index 247019f..f8d63f2 100644
--- a/include/linux/memcontrol.h
+++ b/include/linux/memcontrol.h
@@ -426,6 +426,7 @@ void memcg_release_cache(struct kmem_cache *cachep);
 struct kmem_cache *
 __memcg_kmem_get_cache(struct kmem_cache *cachep, gfp_t gfp);
 void mem_cgroup_destroy_cache(struct kmem_cache *cachep);
+void mem_cgroup_remove_child_kmem_cache(struct kmem_cache *cachep, int id);
 #else
 static inline void memcg_register_cache(struct mem_cgroup *memcg,
                                         struct kmem_cache *s)
diff --git a/include/linux/slab.h b/include/linux/slab.h
index b9310a4..3cfd784 100644
--- a/include/linux/slab.h
+++ b/include/linux/slab.h
@@ -189,6 +189,7 @@ struct mem_cgroup_cache_params {
   bool dead;
   atomic_t nr_pages;
   struct list_head destroyed_list; /* Used when deleting memcg cache */
+ struct list_head sibling_list;
 };
 #endif

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

```
index 1231d86..d92d963 100644
--- a/mm/memcontrol.c
+++ b/mm/memcontrol.c
@@ -586,8 +586,11 @@ static struct kmem_cache *kmem_cache_dup(struct mem_cgroup
*memcg,

  new = kmem_cache_create_memcg(memcg, name, s->object_size, s->align,
         (s->flags & ~SLAB_PANIC), s->ctor, s);
- if (new)
+ if (new) {
   new->allocflags |= __GFP_KMEMCG;
+  list_add(&new->memcg_params.sibling_list,
+    &s->memcg_params.sibling_list);
+ }

  kfree(name);
  return new;
@@ -600,6 +603,7 @@ void memcg_register_cache(struct mem_cgroup *memcg, struct
kmem_cache *cachep)
  int id = -1;

  INIT_LIST_HEAD(&cachep->memcg_params.destroyed_list);
+ INIT_LIST_HEAD(&cachep->memcg_params.sibling_list);

  if (!memcg)
   id = ida_simple_get(&cache_types, 0, MAX_KMEM_CACHE_TYPES,
@@ -611,6 +615,9 @@ void memcg_release_cache(struct kmem_cache *cachep)
 {
  if (cachep->memcg_params.id != -1)
   ida_simple_remove(&cache_types, cachep->memcg_params.id);
+ else
+  list_del(&cachep->memcg_params.sibling_list);
+
 }

 /*
@@ -944,6 +951,13 @@ static void memcg_create_cache_enqueue(struct mem_cgroup
*memcg,
  schedule_work(&memcg_create_cache_work);
 }

+void mem_cgroup_remove_child_kmem_cache(struct kmem_cache *cachep, int id)
+{
+ mutex_lock(&memcg_cache_mutex);
+ cachep->memcg_params.memcg->slabs[id] = NULL;
+ mutex_unlock(&memcg_cache_mutex);
+}
+
```

```
 /*
  * Return the kmem_cache we're supposed to use for a slab allocation.
  * We try to use the current memcg's version of the cache.
diff --git a/mm/slab_common.c b/mm/slab_common.c
index 562146b..6504557 100644
--- a/mm/slab_common.c
+++ b/mm/slab_common.c
@@ -189,8 +189,39 @@ struct kmem_cache *kmem_cache_create(const char *name, size_t
size, size_t align
 }
 EXPORT_SYMBOL(kmem_cache_create);

+static void kmem_cache_destroy_memcg_children(struct kmem_cache *s)
+{
+#ifdef CONFIG_MEMCG_KMEM
+ struct kmem_cache *c;
+ struct mem_cgroup_cache_params *p, *tmp;
+ int id = s->memcg_params.id;
+
+ if (id == -1)
+  return;
+
+ mutex_lock(&slab_mutex);
+ list_for_each_entry_safe(p, tmp,
+      &s->memcg_params.sibling_list, sibling_list) {
+  c = container_of(p, struct kmem_cache, memcg_params);
+  if (WARN_ON(c == s))
+   continue;
+
+  mutex_unlock(&slab_mutex);
+  BUG_ON(c->memcg_params.id != -1);
+  mem_cgroup_remove_child_kmem_cache(c, id);
+  kmem_cache_destroy(c);
+  mutex_lock(&slab_mutex);
+ }
+ mutex_unlock(&slab_mutex);
+#endif /* CONFIG_MEMCG_KMEM */
+}
+
 void kmem_cache_destroy(struct kmem_cache *s)
 {
+
+ /* Destroy all the children caches if we aren't a memcg cache */
+ kmem_cache_destroy_memcg_children(s);
+
  get_online_cpus();
  mutex_lock(&slab_mutex);
  list_del(&s->list);
```

--
1.7.10.4

Subject: [PATCH 09/10] slab: slab-specific propagation changes.
Posted by Glauber Costa on Wed, 25 Jul 2012 14:38:20 GMT
View Forum Message <> Reply to Message

When a parent cache does tune_cpucache, we need to propagate that to the
children as well. For that, we unfortunately need to tap into the slab core.

Signed-off-by: Glauber Costa <glommer@parallels.com>
CC: Christoph Lameter <cl@linux.com>
CC: Pekka Enberg <penberg@cs.helsinki.fi>
CC: Michal Hocko <mhocko@suse.cz>
CC: Kamezawa Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com>
CC: Johannes Weiner <hannes@cmpxchg.org>
CC: Suleiman Souhlal <suleiman@google.com>
---
 mm/slab.c | 28 ++++++++++++++++++++++++++-
 mm/slab_common.c | 1 +
 2 files changed, 28 insertions(+), 1 deletion(-)

diff --git a/mm/slab.c b/mm/slab.c
index 21d7cf7..6d8d449 100644
--- a/mm/slab.c
+++ b/mm/slab.c
@@ -3877,7 +3877,7 @@ static void do_ccupdate_local(void *info)
 }

 /* Always called with the slab_mutex held */
-static int do_tune_cpucache(struct kmem_cache *cachep, int limit,
+static int __do_tune_cpucache(struct kmem_cache *cachep, int limit,
     int batchcount, int shared, gfp_t gfp)
 {
  struct ccupdate_struct *new;
@@ -3920,6 +3920,32 @@ static int do_tune_cpucache(struct kmem_cache *cachep, int limit,
  return alloc_kmemlist(cachep, gfp);
 }

+static int do_tune_cpucache(struct kmem_cache *cachep, int limit,
+   int batchcount, int shared, gfp_t gfp)
+{
+ int ret;
+#ifdef CONFIG_MEMCG_KMEM
+ struct kmem_cache *c;
+ struct mem_cgroup_cache_params *p;
+#endif

```
+
+ ret = __do_tune_cpucache(cachep, limit, batchcount, shared, gfp);
+#ifdef CONFIG_MEMCG_KMEM
+ if (slab_state < FULL)
+  return ret;
+
+ if ((ret < 0) || (cachep->memcg_params.id == -1))
+  return ret;
+
+ list_for_each_entry(p, &cachep->memcg_params.sibling_list, sibling_list) {
+  c = container_of(p, struct kmem_cache, memcg_params);
+  /* return value determined by the parent cache only */
+  __do_tune_cpucache(c, limit, batchcount, shared, gfp);
+ }
+#endif
+ return ret;
+}
+
 /* Called with slab_mutex held always */
 static int enable_cpucache(struct kmem_cache *cachep, gfp_t gfp)
 {
diff --git a/mm/slab_common.c b/mm/slab_common.c
index 6504557..e340a7d 100644
--- a/mm/slab_common.c
+++ b/mm/slab_common.c
@@ -151,6 +151,7 @@ kmem_cache_create_memcg(struct mem_cgroup *memcg, const char
*name, size_t size,
  s->flags = flags;
  s->align = calculate_alignment(flags, align, size);
 #ifdef CONFIG_MEMCG_KMEM
+ s->memcg_params.id = -1; /* not registered yet */
  s->memcg_params.memcg = memcg;
  s->memcg_params.parent = parent_cache;
 #endif
--
1.7.10.4
```

Subject: [PATCH 10/10] memcg/sl[au]b: shrink dead caches
Posted by Glauber Costa on Wed, 25 Jul 2012 14:38:21 GMT
View Forum Message <> Reply to Message

In the slub allocator, when the last object of a page goes away, we
don't necessarily free it - there is not necessarily a test for empty
page in any slab_free path.

This means that when we destroy a memcg cache that happened to be empty,
those caches may take a lot of time to go away: removing the memcg

reference won't destroy them - because there are pending references, and
the empty pages will stay there, until a shrinker is called upon for any
reason.

This patch marks all memcg caches as dead. kmem_cache_shrink is called
for the ones who are not yet dead - this will force internal cache
reorganization, and then all references to empty pages will be removed.

An unlikely branch is used to make sure this case does not affect
performance in the usual slab_free path.

The slab allocator has a time based reaper that would eventually get rid
of the objects, but we can also call it explicitly, since dead caches
are not a likely event.

[v2: also call verify_dead for the slab]

Signed-off-by: Glauber Costa <glommer@parallels.com>
CC: Christoph Lameter <cl@linux.com>
CC: Pekka Enberg <penberg@cs.helsinki.fi>
CC: Michal Hocko <mhocko@suse.cz>
CC: Kamezawa Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com>
CC: Johannes Weiner <hannes@cmpxchg.org>
CC: Suleiman Souhlal <suleiman@google.com>
---
 include/linux/slab.h | 3 +++
 mm/memcontrol.c | 44 ++-
 mm/slab.c | 2 ++
 mm/slab.h | 10 ++++++++++
 mm/slub.c | 1 +
 5 files changed, 59 insertions(+), 1 deletion(-)

diff --git a/include/linux/slab.h b/include/linux/slab.h
index 3cfd784..6ca9ccb 100644
--- a/include/linux/slab.h
+++ b/include/linux/slab.h
@@ -182,6 +182,8 @@ unsigned int kmem_cache_size(struct kmem_cache *);
 #endif

 #ifdef CONFIG_MEMCG_KMEM
+#include <linux/workqueue.h>
+
 struct mem_cgroup_cache_params {
  struct mem_cgroup *memcg;
  struct kmem_cache *parent;
@@ -190,6 +192,7 @@ struct mem_cgroup_cache_params {
  atomic_t nr_pages;
  struct list_head destroyed_list; /* Used when deleting memcg cache */

```
  struct list_head sibling_list;
+ struct work_struct cache_shrinker;
 };
 #endif

diff --git a/mm/memcontrol.c b/mm/memcontrol.c
index d92d963..747efec 100644
--- a/mm/memcontrol.c
+++ b/mm/memcontrol.c
@@ -568,7 +568,7 @@ static char *memcg_cache_name(struct mem_cgroup *memcg, struct
kmem_cache *cache

  BUG_ON(dentry == NULL);

- name = kasprintf(GFP_KERNEL, "%s(%d:%s)",
+ name = kasprintf(GFP_KERNEL, "%s(%d:%s)dead",
      cachep->name, css_id(&memcg->css), dentry->d_name.name);

  return name;
@@ -749,12 +749,25 @@ static void disarm_kmem_keys(struct mem_cgroup *memcg)
  WARN_ON(res_counter_read_u64(&memcg->kmem, RES_USAGE) != 0);
 }

+static void cache_shrinker_work_func(struct work_struct *work)
+{
+ struct mem_cgroup_cache_params *params;
+ struct kmem_cache *cachep;
+
+ params = container_of(work, struct mem_cgroup_cache_params,
+         cache_shrinker);
+ cachep = container_of(params, struct kmem_cache, memcg_params);
+
+ kmem_cache_shrink(cachep);
+}
+
 static DEFINE_MUTEX(memcg_cache_mutex);
 static struct kmem_cache *memcg_create_kmem_cache(struct mem_cgroup *memcg,
         struct kmem_cache *cachep)
 {
  struct kmem_cache *new_cachep;
  int idx;
+ char *name;

  BUG_ON(!memcg_kmem_enabled(memcg));

@@ -774,10 +787,21 @@ static struct kmem_cache *memcg_create_kmem_cache(struct
mem_cgroup *memcg,
   goto out;
```

```
  }

+ /*
+  * Because the cache is expected to duplicate the string,
+  * we must make sure it has opportunity to copy its full
+  * name. Only now we can remove the dead part from it
+  */
+ name = (char *)new_cachep->name;
+ if (name)
+  name[strlen(name) - 4] = '\0';
+
  mem_cgroup_get(memcg);
  memcg->slabs[idx] = new_cachep;
  new_cachep->memcg_params.memcg = memcg;
  atomic_set(&new_cachep->memcg_params.nr_pages , 0);
+ INIT_WORK(&new_cachep->memcg_params.cache_shrinker,
+    cache_shrinker_work_func);
 out:
  mutex_unlock(&memcg_cache_mutex);
  return new_cachep;
@@ -800,6 +824,21 @@ static void kmem_cache_destroy_work_func(struct work_struct *w)
  struct mem_cgroup_cache_params *p, *tmp;
  unsigned long flags;
  LIST_HEAD(del_unlocked);
+ LIST_HEAD(shrinkers);
+
+ spin_lock_irqsave(&cache_queue_lock, flags);
+ list_for_each_entry_safe(p, tmp, &destroyed_caches, destroyed_list) {
+  cachep = container_of(p, struct kmem_cache, memcg_params);
+  if (atomic_read(&cachep->memcg_params.nr_pages) != 0)
+   list_move(&cachep->memcg_params.destroyed_list, &shrinkers);
+ }
+ spin_unlock_irqrestore(&cache_queue_lock, flags);
+
+ list_for_each_entry_safe(p, tmp, &shrinkers, destroyed_list) {
+  cachep = container_of(p, struct kmem_cache, memcg_params);
+  list_del(&cachep->memcg_params.destroyed_list);
+  kmem_cache_shrink(cachep);
+ }

  spin_lock_irqsave(&cache_queue_lock, flags);
  list_for_each_entry_safe(p, tmp, &destroyed_caches, destroyed_list) {
@@ -877,11 +916,14 @@ static void mem_cgroup_destroy_all_caches(struct mem_cgroup
*memcg)

  spin_lock_irqsave(&cache_queue_lock, flags);
  for (i = 0; i < MAX_KMEM_CACHE_TYPES; i++) {
+  char *name;
```

```
   cachep = memcg->slabs[i];
   if (!cachep)
    continue;

   cachep->memcg_params.dead = true;
+  name = (char *)cachep->name;
+  name[strlen(name)] = 'd';
   __mem_cgroup_destroy_cache(cachep);
  }
  spin_unlock_irqrestore(&cache_queue_lock, flags);
diff --git a/mm/slab.c b/mm/slab.c
index 6d8d449..68b293b 100644
--- a/mm/slab.c
+++ b/mm/slab.c
@@ -3558,6 +3558,8 @@ static inline void __cache_free(struct kmem_cache *cachep, void
*objp,
  }

  ac->entry[ac->avail++] = objp;
+
+ kmem_cache_verify_dead(cachep);
 }

 /**
diff --git a/mm/slab.h b/mm/slab.h
index d9df178..0e748e8 100644
--- a/mm/slab.h
+++ b/mm/slab.h
@@ -80,6 +80,12 @@ static inline void mem_cgroup_release_pages(struct kmem_cache *s, int
order)
  if (atomic_sub_and_test((1 << order), &s->memcg_params.nr_pages))
   mem_cgroup_destroy_cache(s);
 }
+
+static inline void kmem_cache_verify_dead(struct kmem_cache *s)
+{
+ if (unlikely(s->memcg_params.dead))
+  schedule_work(&s->memcg_params.cache_shrinker);
+}
 #else
 static inline bool slab_is_parent(struct kmem_cache *s, struct kmem_cache *p)
 {
@@ -98,4 +104,8 @@ static inline void mem_cgroup_bind_pages(struct kmem_cache *s, int
order)
 static inline void mem_cgroup_release_pages(struct kmem_cache *s, int order)
 {
 }
+
+static inline void kmem_cache_verify_dead(struct kmem_cache *s)
+{
+}
 #endif
```

```
+static inline void kmem_cache_verify_dead(struct kmem_cache *s)
+{
+}
 #endif
diff --git a/mm/slub.c b/mm/slub.c
index fdb1a0d..55946c3 100644
--- a/mm/slub.c
+++ b/mm/slub.c
@@ -2582,6 +2582,7 @@ redo:
  } else
   __slab_free(s, page, x, addr);

+ kmem_cache_verify_dead(s);
 }

 void kmem_cache_free(struct kmem_cache *s, void *x)
--
1.7.10.4
```

Subject: Re: [PATCH 05/10] slab: allow enable_cpu_cache to use preset values for its tunables
Posted by Christoph Lameter on Wed, 25 Jul 2012 17:05:28 GMT
View Forum Message <> Reply to Message

On Wed, 25 Jul 2012, Glauber Costa wrote:

> SLAB allows us to tune a particular cache behavior with tunables.
> When creating a new memcg cache copy, we'd like to preserve any tunables
> the parent cache already had.

So does SLUB but I do not see a patch for that allocator.

Subject: Re: [PATCH 09/10] slab: slab-specific propagation changes.
Posted by Christoph Lameter on Wed, 25 Jul 2012 17:07:00 GMT
View Forum Message <> Reply to Message

On Wed, 25 Jul 2012, Glauber Costa wrote:

> When a parent cache does tune_cpucache, we need to propagate that to the
> children as well. For that, we unfortunately need to tap into the slab core.

Slub also has tunables.

Subject: Re: [PATCH 10/10] memcg/sl[au]b: shrink dead caches
Posted by Christoph Lameter on Wed, 25 Jul 2012 17:13:23 GMT
View Forum Message <> Reply to Message

On Wed, 25 Jul 2012, Glauber Costa wrote:

> In the slub allocator, when the last object of a page goes away, we
> don't necessarily free it - there is not necessarily a test for empty
> page in any slab_free path.

That is true for the slab allocator as well. In either case calling
kmem_cache_shrink() will make the objects go away by draining the cached
objects and freeing the pages used for the objects back to the page
allocator. You do not need this patch. Just call the proper functions to
drop the objecgts in the caches in either allocator.

> The slab allocator has a time based reaper that would eventually get rid
> of the objects, but we can also call it explicitly, since dead caches
> are not a likely event.

So this is already for both allocators?

Subject: Re: [PATCH 02/10] consider a memcg parameter in kmem_create_cache
Posted by Kirill A. Shutsemov on Wed, 25 Jul 2012 18:10:13 GMT
View Forum Message <> Reply to Message

On Wed, Jul 25, 2012 at 06:38:13PM +0400, Glauber Costa wrote:

...

> @@ -337,6 +341,12 @@ extern void *__kmalloc_track_caller(size_t, gfp_t, unsigned long);
> __kmalloc(size, flags)
> #endif /* DEBUG_SLAB */
>
> +#ifdef CONFIG_MEMCG_KMEM
> +#define MAX_KMEM_CACHE_TYPES 400
> +#else
> +#define MAX_KMEM_CACHE_TYPES 0
> +#endif /* CONFIG_MEMCG_KMEM */
> +
> #ifdef CONFIG_NUMA
> /*
> * kmalloc_node_track_caller is a special version of kmalloc_node that

...

> @@ -527,6 +532,24 @@ static inline bool memcg_kmem_enabled(struct mem_cgroup

*memcg)
>   memcg->kmem_accounted;
> }
>
> +struct ida cache_types;
> +
> +void memcg_register_cache(struct mem_cgroup *memcg, struct kmem_cache *cachep)
> +{
> + int id = -1;
> +
> + if (!memcg)
> + id = ida_simple_get(&cache_types, 0, MAX_KMEM_CACHE_TYPES,
> +   GFP_KERNEL);

MAX_KMEM_CACHE_TYPES is 0 if CONFIG_MEMCG_KMEM undefined.
If 'end' parameter of ida_simple_get() is 0 it will use default max value
which is 0x80000000.
I guess you want MAX_KMEM_CACHE_TYPES to be 1 for !CONFIG_MEMCG_KMEM.

--
 Kirill A. Shutemov

Subject: Re: [PATCH 10/10] memcg/sl[au]b: shrink dead caches
Posted by Glauber Costa on Wed, 25 Jul 2012 18:16:34 GMT
View Forum Message <> Reply to Message

On 07/25/2012 09:13 PM, Christoph Lameter wrote:
> On Wed, 25 Jul 2012, Glauber Costa wrote:
>
>> In the slub allocator, when the last object of a page goes away, we
>> don't necessarily free it - there is not necessarily a test for empty
>> page in any slab_free path.
>
> That is true for the slab allocator as well. In either case calling
> kmem_cache_shrink() will make the objects go away by draining the cached
> objects and freeing the pages used for the objects back to the page
> allocator. You do not need this patch. Just call the proper functions to
> drop the objecgts in the caches in either allocator.
>
>> The slab allocator has a time based reaper that would eventually get rid
>> of the objects, but we can also call it explicitly, since dead caches
>> are not a likely event.
>
> So this is already for both allocators?
>
Yes, I just didn't updated the whole changelog. my bad.

Subject: Re: [PATCH 05/10] slab: allow enable_cpu_cache to use preset values for its tunables
Posted by Glauber Costa on Wed, 25 Jul 2012 18:24:57 GMT
View Forum Message <> Reply to Message

On 07/25/2012 09:05 PM, Christoph Lameter wrote:
> On Wed, 25 Jul 2012, Glauber Costa wrote:
>
>> SLAB allows us to tune a particular cache behavior with tunables.
>> When creating a new memcg cache copy, we'd like to preserve any tunables
>> the parent cache already had.
>
> So does SLUB but I do not see a patch for that allocator.
>
It is certainly not through does the same method as SLAB, right ?
Writing to /proc/slabinfo gives me an I/O error
I assume it is something through sysfs, but schiming through the code
now, I can't find any per-cache tunables. Would you mind pointing me to
them?

In any case, are you happy with the SLAB one, and how they are propagated?

Subject: Re: [PATCH 05/10] slab: allow enable_cpu_cache to use preset values for its tunables
Posted by Christoph Lameter on Wed, 25 Jul 2012 18:33:16 GMT
View Forum Message <> Reply to Message

On Wed, 25 Jul 2012, Glauber Costa wrote:

> It is certainly not through does the same method as SLAB, right ?
> Writing to /proc/slabinfo gives me an I/O error
> I assume it is something through sysfs, but schiming through the code
> now, I can't find any per-cache tunables. Would you mind pointing me to
> them?

The slab attributes in /sys/kernel/slab/<slabname>/<attr> can be modified
for some values. I think that could be the default method for the future
since it allows easy addition of new tunables as needed.

Subject: Re: [PATCH 01/10] slab/slub: struct memcg_params
Posted by Glauber Costa on Wed, 25 Jul 2012 19:25:31 GMT
View Forum Message <> Reply to Message

On 07/25/2012 11:26 PM, Kirill A. Shutemov wrote:
> On Wed, Jul 25, 2012 at 06:38:12PM +0400, Glauber Costa wrote:
>> For the kmem slab controller, we need to record some extra

>> information in the kmem_cache structure.
>>
>> Signed-off-by: Glauber Costa <glommer@parallels.com>
>> Signed-off-by: Suleiman Souhlal <suleiman@google.com>
>> CC: Christoph Lameter <cl@linux.com>
>> CC: Pekka Enberg <penberg@cs.helsinki.fi>
>> CC: Michal Hocko <mhocko@suse.cz>
>> CC: Kamezawa Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com>
>> CC: Johannes Weiner <hannes@cmpxchg.org>
>> ---
>> include/linux/slab.h | 7 +++++++
>> include/linux/slab_def.h | 4 ++++
>> include/linux/slub_def.h | 3 +++
>> 3 files changed, 14 insertions(+)
>>
>> diff --git a/include/linux/slab.h b/include/linux/slab.h
>> index 0dd2dfa..3152bcd 100644
>> --- a/include/linux/slab.h
>> +++ b/include/linux/slab.h
>> @@ -177,6 +177,13 @@ unsigned int kmem_cache_size(struct kmem_cache *);
>> #define ARCH_SLAB_MINALIGN __alignof__(unsigned long long)
>> #endif
>>
>> +#ifdef CONFIG_MEMCG_KMEM
>> +struct mem_cgroup_cache_params {
>> + struct mem_cgroup *memcg;
>> + int id;
>> +};
>
> IIUC, we only need the id to make slab name unique. Why can't we embed
> the id to struct mem_cgroup? Is it possible to have multiple slabs with
> the same combination of type, size, and memcg?
>
Humm, The id does not serve this purpose (perhaps deserves a comment here)

The purpose of the id is that given a slab, we can access it's memcg
equivalent in constant time through the cache array in memcg.

Subject: Re: [PATCH 01/10] slab/slub: struct memcg_params
Posted by Kirill A. Shutsemov on Wed, 25 Jul 2012 19:26:39 GMT

View Forum Message <> Reply to Message

On Wed, Jul 25, 2012 at 06:38:12PM +0400, Glauber Costa wrote:
> For the kmem slab controller, we need to record some extra
> information in the kmem_cache structure.
>
> Signed-off-by: Glauber Costa <glommer@parallels.com>

> Signed-off-by: Suleiman Souhlal <suleiman@google.com>
> CC: Christoph Lameter <cl@linux.com>
> CC: Pekka Enberg <penberg@cs.helsinki.fi>
> CC: Michal Hocko <mhocko@suse.cz>
> CC: Kamezawa Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com>
> CC: Johannes Weiner <hannes@cmpxchg.org>
> ---
> include/linux/slab.h | 7 +++++++
> include/linux/slab_def.h | 4 ++++
> include/linux/slub_def.h | 3 +++
> 3 files changed, 14 insertions(+)
>
> diff --git a/include/linux/slab.h b/include/linux/slab.h
> index 0dd2dfa..3152bcd 100644
> --- a/include/linux/slab.h
> +++ b/include/linux/slab.h
> @@ -177,6 +177,13 @@ unsigned int kmem_cache_size(struct kmem_cache *);
> #define ARCH_SLAB_MINALIGN __alignof__(unsigned long long)
> #endif
>
> +#ifdef CONFIG_MEMCG_KMEM
> +struct mem_cgroup_cache_params {
> + struct mem_cgroup *memcg;
> + int id;
> +};

IIUC, we only need the id to make slab name unique. Why can't we embed
the id to struct mem_cgroup? Is it possible to have multiple slabs with
the same combination of type, size, and memcg?

--
 Kirill A. Shutemov

Subject: Re: [PATCH 03/10] memcg: infrastructure to match an allocation to the right cache
Posted by Kirill A. Shutsemov on Wed, 25 Jul 2012 21:53:32 GMT
View Forum Message <> Reply to Message

On Wed, Jul 25, 2012 at 06:38:14PM +0400, Glauber Costa wrote:
> The page allocator is able to bind a page to a memcg when it is
> allocated. But for the caches, we'd like to have as many objects as
> possible in a page belonging to the same cache.
>
> This is done in this patch by calling memcg_kmem_get_cache in the
> beginning of every allocation function. This routing is patched out by
> static branches when kernel memory controller is not being used.
>

> It assumes that the task allocating, which determines the memcg in the
> page allocator, belongs to the same cgroup throughout the whole process.
> Misacounting can happen if the task calls memcg_kmem_get_cache() while
> belonging to a cgroup, and later on changes. This is considered
> acceptable, and should only happen upon task migration.
>
> Before the cache is created by the memcg core, there is also a possible
> imbalance: the task belongs to a memcg, but the cache being allocated
> from is the global cache, since the child cache is not yet guaranteed to
> be ready. This case is also fine, since in this case the GFP_KMEMCG will
> not be passed and the page allocator will not attempt any cgroup
> accounting.
>
> Signed-off-by: Glauber Costa <glommer@parallels.com>
> CC: Christoph Lameter <cl@linux.com>
> CC: Pekka Enberg <penberg@cs.helsinki.fi>
> CC: Michal Hocko <mhocko@suse.cz>
> CC: Kamezawa Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com>
> CC: Johannes Weiner <hannes@cmpxchg.org>
> CC: Suleiman Souhlal <suleiman@google.com>
> ---
> include/linux/memcontrol.h | 38 ++++++++
> init/Kconfig | 2 +-
> mm/memcontrol.c | 221 ++-
> 3 files changed, 259 insertions(+), 2 deletions(-)
>
> diff --git a/include/linux/memcontrol.h b/include/linux/memcontrol.h
> index d9229a3..bd1f34b 100644
> --- a/include/linux/memcontrol.h
> +++ b/include/linux/memcontrol.h
> @@ -423,6 +423,8 @@ int memcg_css_id(struct mem_cgroup *memcg);
> void memcg_register_cache(struct mem_cgroup *memcg,
>   struct kmem_cache *s);
> void memcg_release_cache(struct kmem_cache *cachep);
> +struct kmem_cache *
> +__memcg_kmem_get_cache(struct kmem_cache *cachep, gfp_t gfp);
> #else
> static inline void memcg_register_cache(struct mem_cgroup *memcg,
>   struct kmem_cache *s)
> @@ -456,6 +458,12 @@ __memcg_kmem_commit_page(struct page *page, struct
mem_cgroup *handle,
>   int order)
> {
> }
> +
> +static inline struct kmem_cache *
> +__memcg_kmem_get_cache(struct kmem_cache *cachep, gfp_t gfp)
> +{

> + return cachep;
> +}
> #endif /* CONFIG_MEMCG_KMEM */
>
> /**
> @@ -515,5 +523,35 @@ void memcg_kmem_commit_page(struct page *page, struct
mem_cgroup *handle,
>   if (memcg_kmem_on)
>   __memcg_kmem_commit_page(page, handle, order);
> }
> +
> +/**
> + * memcg_kmem_get_kmem_cache: selects the correct per-memcg cache for allocation
> + * @cachep: the original global kmem cache
> + * @gfp: allocation flags.
> + *
> + * This function assumes that the task allocating, which determines the memcg
> + * in the page allocator, belongs to the same cgroup throughout the whole
> + * process. Misacounting can happen if the task calls memcg_kmem_get_cache()
> + * while belonging to a cgroup, and later on changes. This is considered
> + * acceptable, and should only happen upon task migration.
> + *
> + * Before the cache is created by the memcg core, there is also a possible
> + * imbalance: the task belongs to a memcg, but the cache being allocated from
> + * is the global cache, since the child cache is not yet guaranteed to be
> + * ready. This case is also fine, since in this case the GFP_KMEMCG will not be
> + * passed and the page allocator will not attempt any cgroup accounting.
> + */
> +static __always_inline struct kmem_cache *
> +memcg_kmem_get_cache(struct kmem_cache *cachep, gfp_t gfp)
> +{
> + if (!memcg_kmem_on)
> + return cachep;
> + if (gfp & __GFP_NOFAIL)
> + return cachep;
> + if (in_interrupt() || (!current->mm) || (current->flags & PF_KTHREAD))
> + return cachep;
> +
> + return __memcg_kmem_get_cache(cachep, gfp);
> +}
> #endif /* _LINUX_MEMCONTROL_H */
>
> diff --git a/init/Kconfig b/init/Kconfig
> index 547bd10..610cfd3 100644
> --- a/init/Kconfig
> +++ b/init/Kconfig
> @@ -741,7 +741,7 @@ config MEMCG_SWAP_ENABLED
>   then swapaccount=0 does the trick).

> config MEMCG_KMEM
>   bool "Memory Resource Controller Kernel Memory accounting (EXPERIMENTAL)"
> - depends on MEMCG && EXPERIMENTAL
> + depends on MEMCG && EXPERIMENTAL && !SLOB
>   default n
>   help
>   The Kernel Memory extension for Memory Resource Controller can limit
> diff --git a/mm/memcontrol.c b/mm/memcontrol.c
> index 88bb826..8d012c7 100644
> --- a/mm/memcontrol.c
> +++ b/mm/memcontrol.c
> @@ -14,6 +14,10 @@
>   * Copyright (C) 2012 Parallels Inc. and Google Inc.
>   * Authors: Glauber Costa and Suleiman Souhlal
>   *
> + * Kernel Memory Controller
> + * Copyright (C) 2012 Parallels Inc. and Google Inc.
> + * Authors: Glauber Costa and Suleiman Souhlal
> + *
>   * This program is free software; you can redistribute it and/or modify
>   * it under the terms of the GNU General Public License as published by
>   * the Free Software Foundation; either version 2 of the License, or
> @@ -339,6 +343,11 @@ struct mem_cgroup {
> #ifdef CONFIG_INET
>   struct tcp_memcontrol tcp_mem;
> #endif
> +
> +#ifdef CONFIG_MEMCG_KMEM
> + /* Slab accounting */
> + struct kmem_cache *slabs[MAX_KMEM_CACHE_TYPES];
> +#endif
> };
>
> enum {
> @@ -532,6 +541,40 @@ static inline bool memcg_kmem_enabled(struct mem_cgroup
*memcg)
>   memcg->kmem_accounted;
> }
>
> +static char *memcg_cache_name(struct mem_cgroup *memcg, struct kmem_cache *cachep)
> +{
> + char *name;
> + struct dentry *dentry;
> +
> + rcu_read_lock();
> + dentry = rcu_dereference(memcg->css.cgroup->dentry);
> + rcu_read_unlock();
> +

```
> + BUG_ON(dentry == NULL);
> +
> + name = kasprintf(GFP_KERNEL, "%s(%d:%s)",
> +     cachep->name, css_id(&memcg->css), dentry->d_name.name);
> +
> + return name;
> +}
> +
> +static struct kmem_cache *kmem_cache_dup(struct mem_cgroup *memcg,
> +      struct kmem_cache *s)
> +{
> + char *name;
> + struct kmem_cache *new;
> +
> + name = memcg_cache_name(memcg, s);
> + if (!name)
> +  return NULL;
> +
> + new = kmem_cache_create_memcg(memcg, name, s->object_size, s->align,
> +          (s->flags & ~SLAB_PANIC), s->ctor);
> +
> + kfree(name);
> + return new;
> +}
> +
>  struct ida cache_types;
>
>  void memcg_register_cache(struct mem_cgroup *memcg, struct kmem_cache *cachep)
> @@ -656,6 +699,14 @@ void __memcg_kmem_free_page(struct page *page, int order)
>  }
>  EXPORT_SYMBOL(__memcg_kmem_free_page);
>
> +static void memcg_slab_init(struct mem_cgroup *memcg)
> +{
> + int i;
> +
> + for (i = 0; i < MAX_KMEM_CACHE_TYPES; i++)
> +  memcg->slabs[i] = NULL;
> +}
```

It seems redundant. mem_cgroup_alloc() uses kzalloc()/vzalloc() to
allocate struct mem_cgroup.

--
Kirill A. Shutemov

Subject: Re: [PATCH 02/10] consider a memcg parameter in kmem_create_cache
Posted by Glauber Costa on Thu, 26 Jul 2012 09:42:05 GMT
View Forum Message <> Reply to Message

On 07/25/2012 10:10 PM, Kirill A. Shutemov wrote:
> On Wed, Jul 25, 2012 at 06:38:13PM +0400, Glauber Costa wrote:
>
> ...
>
>> @@ -337,6 +341,12 @@ extern void *__kmalloc_track_caller(size_t, gfp_t, unsigned long);
>> __kmalloc(size, flags)
>> #endif /* DEBUG_SLAB */
>>
>> +#ifdef CONFIG_MEMCG_KMEM
>> +#define MAX_KMEM_CACHE_TYPES 400
>> +#else
>> +#define MAX_KMEM_CACHE_TYPES 0
>> +#endif /* CONFIG_MEMCG_KMEM */
>> +
>> #ifdef CONFIG_NUMA
>> /*
>> * kmalloc_node_track_caller is a special version of kmalloc_node that
>
> ...
>
>> @@ -527,6 +532,24 @@ static inline bool memcg_kmem_enabled(struct mem_cgroup
*memcg)
>> memcg->kmem_accounted;
>> }
>>
>> +struct ida cache_types;
>> +
>> +void memcg_register_cache(struct mem_cgroup *memcg, struct kmem_cache *cachep)
>> +{
>> + int id = -1;
>> +
>> + if (!memcg)
>> + id = ida_simple_get(&cache_types, 0, MAX_KMEM_CACHE_TYPES,
>> + GFP_KERNEL);
>
> MAX_KMEM_CACHE_TYPES is 0 if CONFIG_MEMCG_KMEM undefined.
> If 'end' parameter of ida_simple_get() is 0 it will use default max value
> which is 0x80000000.
> I guess you want MAX_KMEM_CACHE_TYPES to be 1 for !CONFIG_MEMCG_KMEM.
>
ida_simple_get will not, and should never be called for !CONFIG_MEMCG_KMEM.

Subject: Re: [PATCH 02/10] consider a memcg parameter in kmem_create_cache
Posted by Kirill A. Shutsemov on Thu, 26 Jul 2012 10:01:08 GMT
View Forum Message <> Reply to Message

On Thu, Jul 26, 2012 at 01:42:05PM +0400, Glauber Costa wrote:
> On 07/25/2012 10:10 PM, Kirill A. Shutemov wrote:
> > On Wed, Jul 25, 2012 at 06:38:13PM +0400, Glauber Costa wrote:
> >
> > ...
> >
> >> @@ -337,6 +341,12 @@ extern void *__kmalloc_track_caller(size_t, gfp_t, unsigned long);
> >> __kmalloc(size, flags)
> >> #endif /* DEBUG_SLAB */
> >>
> >> +#ifdef CONFIG_MEMCG_KMEM
> >> +#define MAX_KMEM_CACHE_TYPES 400
> >> +#else
> >> +#define MAX_KMEM_CACHE_TYPES 0
> >> +#endif /* CONFIG_MEMCG_KMEM */
> >> +
> >> #ifdef CONFIG_NUMA
> >> /*
> >> * kmalloc_node_track_caller is a special version of kmalloc_node that
> >
> > ...
> >
> >> @@ -527,6 +532,24 @@ static inline bool memcg_kmem_enabled(struct mem_cgroup
*memcg)
> >> memcg->kmem_accounted;
> >> }
> >>
> >> +struct ida cache_types;
> >> +
> >> +void memcg_register_cache(struct mem_cgroup *memcg, struct kmem_cache *cachep)
> >> +{
> >> + int id = -1;
> >> +
> >> + if (!memcg)
> >> + id = ida_simple_get(&cache_types, 0, MAX_KMEM_CACHE_TYPES,
> >> + GFP_KERNEL);
> >
> > MAX_KMEM_CACHE_TYPES is 0 if CONFIG_MEMCG_KMEM undefined.
> > If 'end' parameter of ida_simple_get() is 0 it will use default max value
> > which is 0x80000000.
> > I guess you want MAX_KMEM_CACHE_TYPES to be 1 for !CONFIG_MEMCG_KMEM.
> >
> ida_simple_get will not, and should never be called for !CONFIG_MEMCG_KMEM.

Okay. Is there an user for MAX_KMEM_CACHE_TYPES if !CONFIG_MEMCG_KMEM?

--
Kirill A. Shutemov

Subject: Re: [PATCH 02/10] consider a memcg parameter in kmem_create_cache
Posted by Glauber Costa on Thu, 26 Jul 2012 10:27:15 GMT
View Forum Message <> Reply to Message

On 07/26/2012 02:01 PM, Kirill A. Shutemov wrote:
> On Thu, Jul 26, 2012 at 01:42:05PM +0400, Glauber Costa wrote:
>> On 07/25/2012 10:10 PM, Kirill A. Shutemov wrote:
>>> On Wed, Jul 25, 2012 at 06:38:13PM +0400, Glauber Costa wrote:
>>>
>>> ...
>>>
>>>> @@ -337,6 +341,12 @@ extern void *__kmalloc_track_caller(size_t, gfp_t, unsigned long);
>>>> __kmalloc(size, flags)
>>>> #endif /* DEBUG_SLAB */
>>>>
>>>> +#ifdef CONFIG_MEMCG_KMEM
>>>> +#define MAX_KMEM_CACHE_TYPES 400
>>>> +#else
>>>> +#define MAX_KMEM_CACHE_TYPES 0
>>>> +#endif /* CONFIG_MEMCG_KMEM */
>>>> +
>>>> #ifdef CONFIG_NUMA
>>>> /*
>>>> * kmalloc_node_track_caller is a special version of kmalloc_node that
>>>
>>> ...
>>>
>>>> @@ -527,6 +532,24 @@ static inline bool memcg_kmem_enabled(struct mem_cgroup
*memcg)
>>>> memcg->kmem_accounted;
>>>> }
>>>>
>>>> +struct ida cache_types;
>>>> +
>>>> +void memcg_register_cache(struct mem_cgroup *memcg, struct kmem_cache *cachep)
>>>> +{
>>>> + int id = -1;
>>>> +
>>>> + if (!memcg)
>>>> + id = ida_simple_get(&cache_types, 0, MAX_KMEM_CACHE_TYPES,
>>>> + GFP_KERNEL);
>>>
>>> MAX_KMEM_CACHE_TYPES is 0 if CONFIG_MEMCG_KMEM undefined.

>>> If 'end' parameter of ida_simple_get() is 0 it will use default max value
>>> which is 0x80000000.
>>> I guess you want MAX_KMEM_CACHE_TYPES to be 1 for !CONFIG_MEMCG_KMEM.
>>>
>> ida_simple_get will not, and should never be called for !CONFIG_MEMCG_KMEM.
>
> Okay. Is there an user for MAX_KMEM_CACHE_TYPES if !CONFIG_MEMCG_KMEM?
>

Actually, not anymore. I will just remove it then, instead of forcing it
to 0.

Thanks!

Subject: Re: [PATCH 05/10] slab: allow enable_cpu_cache to use preset values for its tunables
Posted by Glauber Costa on Thu, 26 Jul 2012 14:02:45 GMT
View Forum Message <> Reply to Message

On 07/25/2012 10:33 PM, Christoph Lameter wrote:
> On Wed, 25 Jul 2012, Glauber Costa wrote:
>
>> It is certainly not through does the same method as SLAB, right ?
>> Writing to /proc/slabinfo gives me an I/O error
>> I assume it is something through sysfs, but schiming through the code
>> now, I can't find any per-cache tunables. Would you mind pointing me to
>> them?
>
> The slab attributes in /sys/kernel/slab/<slabname>/<attr> can be modified
> for some values. I think that could be the default method for the future
> since it allows easy addition of new tunables as needed.
>

Christoph, would the following PoC patch be enough?

File Attachments
1) 0001-slub-propagation.patch, downloaded 1746 times

Subject: Re: [PATCH 04/10] memcg: skip memcg kmem allocations in specified code regions
Posted by Kirill A. Shutsemov on Mon, 30 Jul 2012 12:49:50 GMT
View Forum Message <> Reply to Message

On Wed, Jul 25, 2012 at 06:38:15PM +0400, Glauber Costa wrote:
> This patch creates a mechanism that skip memcg allocations during

> certain pieces of our core code. It basically works in the same way
> as preempt_disable()/preempt_enable(): By marking a region under
> which all allocations will be accounted to the root memcg.
>
> We need this to prevent races in early cache creation, when we
> allocate data using caches that are not necessarily created already.

Why not a GFP_* flag?

--
Kirill A. Shutemov

Subject: Re: [PATCH 06/10] sl[au]b: Allocate objects from memcg cache
Posted by Kirill A. Shutsemov on Mon, 30 Jul 2012 12:58:30 GMT
View Forum Message <> Reply to Message

On Wed, Jul 25, 2012 at 06:38:17PM +0400, Glauber Costa wrote:
> We are able to match a cache allocation to a particular memcg. If the
> task doesn't change groups during the allocation itself - a rare event,
> this will give us a good picture about who is the first group to touch a
> cache page.
>
> This patch uses the now available infrastructure by calling
> memcg_kmem_get_cache() before all the cache allocations.
>
> Signed-off-by: Glauber Costa <glommer@parallels.com>
> CC: Christoph Lameter <cl@linux.com>
> CC: Pekka Enberg <penberg@cs.helsinki.fi>
> CC: Michal Hocko <mhocko@suse.cz>
> CC: Kamezawa Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com>
> CC: Johannes Weiner <hannes@cmpxchg.org>
> CC: Suleiman Souhlal <suleiman@google.com>
> ---
> include/linux/slub_def.h | 18 ++++++++++++-----
> mm/memcontrol.c | 2 ++
> mm/slab.c | 4 ++++
> mm/slub.c | 1 +
> 4 files changed, 20 insertions(+), 5 deletions(-)
>
> diff --git a/include/linux/slub_def.h b/include/linux/slub_def.h
> index 8bb8ad2..148000a 100644
> --- a/include/linux/slub_def.h
> +++ b/include/linux/slub_def.h
> @@ -13,6 +13,8 @@
> #include <linux/kobject.h>
>
> #include <linux/kmemleak.h>

> +#include <linux/memcontrol.h>
> +#include <linux/mm.h>
>
> enum stat_item {
>   ALLOC_FASTPATH, /* Allocation from cpu slab */
> @@ -209,14 +211,14 @@ static __always_inline int kmalloc_index(size_t size)
>   * This ought to end up with a global pointer to the right cache
>   * in kmalloc_caches.
>   */
> -static __always_inline struct kmem_cache *kmalloc_slab(size_t size)
> +static __always_inline struct kmem_cache *kmalloc_slab(gfp_t flags, size_t size)
> {
>   int index = kmalloc_index(size);
>
>   if (index == 0)
>   return NULL;
>
> - return kmalloc_caches[index];
> + return memcg_kmem_get_cache(kmalloc_caches[index], flags);
> }
>
> void *kmem_cache_alloc(struct kmem_cache *, gfp_t);
> @@ -225,7 +227,13 @@ void *__kmalloc(size_t size, gfp_t flags);
> static __always_inline void *
> kmalloc_order(size_t size, gfp_t flags, unsigned int order)
> {
> - void *ret = (void *) __get_free_pages(flags | __GFP_COMP, order);
> + void *ret;
> +
> + flags = __GFP_COMP;
> +#ifdef CONFIG_MEMCG_KMEM
> + flags |= __GFP_KMEMCG;
> +#endif

Em.. I don't see where __GFP_KMEMCG is defined.
It should be 0 for !CONFIG_MEMCG_KMEM.

--
 Kirill A. Shutemov

Subject: Re: [PATCH 06/10] sl[au]b: Allocate objects from memcg cache
Posted by Glauber Costa on Mon, 30 Jul 2012 13:11:06 GMT
View Forum Message <> Reply to Message

On 07/30/2012 04:58 PM, Kirill A. Shutemov wrote:
> On Wed, Jul 25, 2012 at 06:38:17PM +0400, Glauber Costa wrote:
>> We are able to match a cache allocation to a particular memcg. If the

>> task doesn't change groups during the allocation itself - a rare event,
>> this will give us a good picture about who is the first group to touch a
>> cache page.
>>
>> This patch uses the now available infrastructure by calling
>> memcg_kmem_get_cache() before all the cache allocations.
>>
>> Signed-off-by: Glauber Costa <glommer@parallels.com>
>> CC: Christoph Lameter <cl@linux.com>
>> CC: Pekka Enberg <penberg@cs.helsinki.fi>
>> CC: Michal Hocko <mhocko@suse.cz>
>> CC: Kamezawa Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com>
>> CC: Johannes Weiner <hannes@cmpxchg.org>
>> CC: Suleiman Souhlal <suleiman@google.com>
>> ---
>> include/linux/slub_def.h | 18 ++++++++++++-----
>> mm/memcontrol.c | 2 ++
>> mm/slab.c | 4 ++++
>> mm/slub.c | 1 +
>> 4 files changed, 20 insertions(+), 5 deletions(-)
>>
>> diff --git a/include/linux/slub_def.h b/include/linux/slub_def.h
>> index 8bb8ad2..148000a 100644
>> --- a/include/linux/slub_def.h
>> +++ b/include/linux/slub_def.h
>> @@ -13,6 +13,8 @@
>> #include <linux/kobject.h>
>>
>> #include <linux/kmemleak.h>
>> +#include <linux/memcontrol.h>
>> +#include <linux/mm.h>
>>
>> enum stat_item {
>>   ALLOC_FASTPATH, /* Allocation from cpu slab */
>> @@ -209,14 +211,14 @@ static __always_inline int kmalloc_index(size_t size)
>>   * This ought to end up with a global pointer to the right cache
>>   * in kmalloc_caches.
>>   */
>> -static __always_inline struct kmem_cache *kmalloc_slab(size_t size)
>> +static __always_inline struct kmem_cache *kmalloc_slab(gfp_t flags, size_t size)
>> {
>>   int index = kmalloc_index(size);
>>
>>   if (index == 0)
>>   return NULL;
>>
>> - return kmalloc_caches[index];
>> + return memcg_kmem_get_cache(kmalloc_caches[index], flags);

>> }
>>
>> void *kmem_cache_alloc(struct kmem_cache *, gfp_t);
>> @@ -225,7 +227,13 @@ void *__kmalloc(size_t size, gfp_t flags);
>> static __always_inline void *
>> kmalloc_order(size_t size, gfp_t flags, unsigned int order)
>> {
>> - void *ret = (void *) __get_free_pages(flags | __GFP_COMP, order);
>> + void *ret;
>> +
>> + flags = __GFP_COMP;
>> +#ifdef CONFIG_MEMCG_KMEM
>> + flags |= __GFP_KMEMCG;
>> +#endif
>
> Em.. I don't see where __GFP_KMEMCG is defined.
> It should be 0 for !CONFIG_MEMCG_KMEM.
>
It is not, sorry.

As I said, this is dependent on another patch series.
My main goal while sending this was to get the slab part - that will
eventually come ontop of that - discussed. Because they are both quite
complex, I believe they benefit from being discussed separately.

You can find the latest version of that here:

https://lkml.org/lkml/2012/6/25/251

Subject: Re: [PATCH 04/10] memcg: skip memcg kmem allocations in specified code regions
Posted by Glauber Costa on Mon, 30 Jul 2012 14:09:23 GMT
View Forum Message <> Reply to Message

On 07/30/2012 04:50 PM, Kirill A. Shutemov wrote:
> On Wed, Jul 25, 2012 at 06:38:15PM +0400, Glauber Costa wrote:
>> This patch creates a mechanism that skip memcg allocations during
>> certain pieces of our core code. It basically works in the same way
>> as preempt_disable()/preempt_enable(): By marking a region under
>> which all allocations will be accounted to the root memcg.
>>
>> We need this to prevent races in early cache creation, when we
>> allocate data using caches that are not necessarily created already.
>
> Why not a GFP_* flag?
>

The main reason for this is to prevent nested calls of
kmem_cache_create(), since they could create (and in my tests, do
create) funny circular dependencies with each other. So the cache
creation itself would proceed without involving memcg.

At first, it is a bit weird to have cache creation itself depending on a
allocation flag test.

Subject: Re: [PATCH 00/10] memcg kmem limitation - slab.
Posted by Frederic Weisbecker on Tue, 31 Jul 2012 16:30:31 GMT
View Forum Message <> Reply to Message

On Wed, Jul 25, 2012 at 06:38:11PM +0400, Glauber Costa wrote:
> Hi,
>
> This is the slab part of the kmem limitation mechanism in its last form. I
> would like to have comments on it to see if we can agree in its form. I
> consider it mature, since it doesn't change much in essence over the last
> forms. However, I would still prefer to defer merging it and merge the
> stack-only patchset first (even if inside the same merge window). That patchset
> contains most of the infrastructure needed here, and merging them separately
> would not only reduce the complexity for reviewers, but allow us a chance to
> have independent testing on them both. I would also likely benefit from some
> extra testing, to make sure the recent changes didn't introduce anything bad.

What is the status of the stack-only limitation patchset BTW? Does anybody oppose
to its merging?

Thanks.

Subject: Re: [PATCH 00/10] memcg kmem limitation - slab.
Posted by Glauber Costa on Tue, 31 Jul 2012 16:39:30 GMT
View Forum Message <> Reply to Message

On 07/31/2012 08:30 PM, Frederic Weisbecker wrote:
> On Wed, Jul 25, 2012 at 06:38:11PM +0400, Glauber Costa wrote:
>> Hi,
>>
>> This is the slab part of the kmem limitation mechanism in its last form. I
>> would like to have comments on it to see if we can agree in its form. I
>> consider it mature, since it doesn't change much in essence over the last
>> forms. However, I would still prefer to defer merging it and merge the
>> stack-only patchset first (even if inside the same merge window). That patchset
>> contains most of the infrastructure needed here, and merging them separately
>> would not only reduce the complexity for reviewers, but allow us a chance to

>> have independent testing on them both. I would also likely benefit from some
>> extra testing, to make sure the recent changes didn't introduce anything bad.
>
> What is the status of the stack-only limitation patchset BTW? Does anybody oppose
> to its merging?
>
> Thanks.
>
Andrew said he would like to see the slab patches in a relatively mature
state first.

I do believe they are in such a state. There are bugs, that I am working
on - but I don't see anything that would change them significantly at
this point.

If Andrew is happy with what he saw in this thread, I could post those
again.
