Subject: [PATCH v3 00/16] slab accounting for memcg
Posted by Glauber Costa on Tue, 18 Sep 2012 14:11:54 GMT
View Forum Message <> Reply to Message

This is a followup to the previous kmem series. I divided them logically
so it gets easier for reviewers. But I believe they are ready to be merged
together (although we can do a two-pass merge if people would prefer)

Throwaway git tree found at:

```
  git://git.kernel.org/pub/scm/linux/kernel/git/glommer/memcg. git kmemcg-slab
```

There are mostly bugfixes since last submission.

For a detailed explanation about this series, please refer to my previous post
(Subj: [PATCH v3 00/13] kmem controller for memcg.)

```
Glauber Costa (16):
  slab/slub: struct memcg_params
  slub: use free_page instead of put_page for freeing kmalloc
    allocation
  slab: Ignore the cflgs bit in cache creation
  provide a common place for initcall processing in kmem_cache
  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: always get the cache from its page in kfree
  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.
  slub: slub-specific propagation changes.
  memcg/sl[au]b: shrink dead caches
  Add documentation about the kmem controller

 Documentation/cgroups/memory.txt |  73 ++++++-
 include/linux/memcontrol.h       |  60 ++++++
 include/linux/sched.h            |   1 +
 include/linux/slab.h             |  23 +++
 include/linux/slab_def.h         |   4 +
 include/linux/slub_def.h         |  18 +-
 init/Kconfig                     |   2 +-
 mm/memcontrol.c                  | 403 +++++++++++++++++++++++++++++++++++++++
 mm/slab.c                        |  70 ++++++-
 mm/slab.h                        |  72 ++++++-
 mm/slab_common.c                 |  85 ++++++++-
 mm/slob.c                 |   5 +
 mm/slub.c                 |  54 ++++--
 13 files changed, 829 insertions(+), 41 deletions(-)

--
1.7.11.4
```

Subject: [PATCH v3 01/16] slab/slub: struct memcg_params
Posted by Glauber Costa on Tue, 18 Sep 2012 14:11:55 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
  /*
--
1.7.11.4
```

Subject: [PATCH v3 02/16] slub: use free_page instead of put_page for freeing kmalloc allocation

Posted by Glauber Costa on Tue, 18 Sep 2012 14:11:56 GMT

View Forum Message <> Reply to Message

When freeing objects, the slub allocator will most of the time free
empty pages by calling __free_pages(). But high-order kmalloc will be
diposed by means of put_page() instead. It makes no sense to call
put_page() in kernel pages that are provided by the object allocators,
so we shouldn't be doing this ourselves. Aside from the consistency
change, we don't change the flow too much. put_page()'s would call its
dtor function, which is __free_pages. We also already do all of the
Compound page tests ourselves, and the Mlock test we lose don't really
matter.

[v2: modified Changelog]

```
Signed-off-by: Glauber Costa <glommer@parallels.com>
Acked-by: Christoph Lameter <cl@linux.com>
CC: David Rientjes <rientjes@google.com>
CC: Pekka Enberg <penberg@kernel.org>
---
 mm/slub.c | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)

diff --git a/mm/slub.c b/mm/slub.c
index 9f86353..09a91d0 100644
--- a/mm/slub.c
+++ b/mm/slub.c
@@ -3451,7 +3451,7 @@ void kfree(const void *x)
  if (unlikely(!PageSlab(page))) {
   BUG_ON(!PageCompound(page));
   kmemleak_free(x);
-  put_page(page);
+  __free_pages(page, compound_order(page));
   return;
  }
  slab_free(page->slab, page, object, _RET_IP_);
--
1.7.11.4
```

Subject: [PATCH v3 03/16] slab: Ignore the cflgs bit in cache creation
Posted by Glauber Costa on Tue, 18 Sep 2012 14:11:57 GMT

View Forum Message <> Reply to Message

No cache should ever pass that as a creation flag, since this bit is used to mark an internal decision of the slab about object placement. We can just ignore this bit if it happens to be passed (such as when duplicating a cache in the kmem memcg patches)

```
Signed-off-by: Glauber Costa <glommer@parallels.com>
CC: Christoph Lameter <cl@linux.com>
CC: Pekka Enberg <penberg@cs.helsinki.fi>
CC: David Rientjes <rientjes@google.com>
---
 mm/slab.c | 1 +
 1 file changed, 1 insertion(+)

diff --git a/mm/slab.c b/mm/slab.c
index a7ed60f..ccf496c 100644
--- a/mm/slab.c
+++ b/mm/slab.c
@@ -2373,6 +2373,7 @@ __kmem_cache_create (struct kmem_cache *cachep, unsigned long flags)
```

```
  int err;
  size_t size = cachep->size;

+ flags &= ~CFLGS_OFF_SLAB;
 #if DEBUG
 #if FORCED_DEBUG
  /*
--
1.7.11.4
```

Subject: [PATCH v3 04/16] provide a common place for initcall processing in kmem_cache

Posted by Glauber Costa on Tue, 18 Sep 2012 14:11:58 GMT

View Forum Message <> Reply to Message

Both SLAB and SLUB depend on some initialization to happen when the
system is already booted, with all subsystems working. This is done
by issuing an initcall that does the final initialization.

This patch moves that to slab_common.c, while creating an empty
placeholder for the SLOB.

Signed-off-by: Glauber Costa <glommer@parallels.com>
CC: Christoph Lameter <cl@linux.com>
CC: Pekka Enberg <penberg@cs.helsinki.fi>
CC: David Rientjes <rientjes@google.com>

```
---
 mm/slab.c        | 5 ++---
 mm/slab.h        | 2 ++
 mm/slab_common.c | 6 ++++++
 mm/slob.c        | 5 +++++
 mm/slub.c        | 4 +---
 5 files changed, 16 insertions(+), 6 deletions(-)

diff --git a/mm/slab.c b/mm/slab.c
index ccf496c..3bac667 100644
--- a/mm/slab.c
+++ b/mm/slab.c
@@ -894,7 +894,7 @@ static void __cpuinit start_cpu_timer(int cpu)
  struct delayed_work *reap_work = &per_cpu(slab_reap_work, cpu);

  /*
-  * When this gets called from do_initcalls via cpucache_init(),
+  * When this gets called from do_initcalls via __kmem_cache_initcall(),
   * init_workqueues() has already run, so keventd will be setup
   * at that time.
   */
```

```
@@ -1822,7 +1822,7 @@ void __init kmem_cache_init_late(void)
   */
 }

-static int __init cpucache_init(void)
+int __init __kmem_cache_initcall(void)
 {
  int cpu;

@@ -1836,7 +1836,6 @@ static int __init cpucache_init(void)
  slab_state = FULL;
  return 0;
 }
-__initcall(cpucache_init);

 static noinline void
 slab_out_of_memory(struct kmem_cache *cachep, gfp_t gfpflags, int nodeid)
diff --git a/mm/slab.h b/mm/slab.h
index 7deeb44..7a2698b 100644
--- a/mm/slab.h
+++ b/mm/slab.h
@@ -47,4 +47,6 @@ static inline struct kmem_cache *__kmem_cache_alias(const char *name, size_t siz

 int __kmem_cache_shutdown(struct kmem_cache *);

+int __kmem_cache_initcall(void);
+
 #endif
diff --git a/mm/slab_common.c b/mm/slab_common.c
index 9c21725..eddbb8a 100644
--- a/mm/slab_common.c
+++ b/mm/slab_common.c
@@ -189,3 +189,9 @@ int slab_is_available(void)
 {
  return slab_state >= UP;
 }
+
+static int __init kmem_cache_initcall(void)
+{
+ return __kmem_cache_initcall();
+}
+__initcall(kmem_cache_initcall);
diff --git a/mm/slob.c b/mm/slob.c
index 3edfeaa..ad91d67 100644
--- a/mm/slob.c
+++ b/mm/slob.c
@@ -622,3 +622,8 @@ void __init kmem_cache_init_late(void)
```

```
 {
  slab_state = FULL;
 }
+
+int __init __kmem_cache_initcall(void)
+{
+ return 0;
+}
diff --git a/mm/slub.c b/mm/slub.c
index 09a91d0..7ac46c6 100644
--- a/mm/slub.c
+++ b/mm/slub.c
@@ -5332,7 +5332,7 @@ static int sysfs_slab_alias(struct kmem_cache *s, const char *name)
  return 0;
 }

-static int __init slab_sysfs_init(void)
+int __init __kmem_cache_initcall(void)
 {
  struct kmem_cache *s;
  int err;
@@ -5370,8 +5370,6 @@ static int __init slab_sysfs_init(void)
  resiliency_test();
  return 0;
 }
-
-__initcall(slab_sysfs_init);
 #endif /* CONFIG_SYSFS */

 /*
--
1.7.11.4
```

Subject: [PATCH v3 05/16] consider a memcg parameter in kmem_create_cache

Posted by Glauber Costa on Tue, 18 Sep 2012 14:11:59 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.

[v2: moved to idr/ida instead of redoing the indexes]
[v3: moved call to ida_init away from cgroup creation to fix a bug]

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 | 20 ++++++++++++++++++++
 include/linux/slab.h       |  8 ++++++++
 mm/memcontrol.c            | 28 ++++++++++++++++++++++++++++
 mm/slab.c                  |  1 +
 mm/slab.h                  | 26 ++++++++++++++++++++++----
 mm/slab_common.c           | 47 ++++++++++++++++++++++++++++++++++++-----------
 mm/slub.c                  | 17 +++++++++++++----
 7 files changed, 128 insertions(+), 19 deletions(-)

diff --git a/include/linux/memcontrol.h b/include/linux/memcontrol.h
index 4ec9fd5..a5f3055 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 {
@@ -413,7 +414,26 @@ extern bool __memcg_kmem_newpage_charge(gfp_t gfp, struct mem_cgroup **memcg,
 extern void __memcg_kmem_commit_charge(struct page *page,
            struct mem_cgroup *memcg, int order);
 extern void __memcg_kmem_uncharge_page(struct page *page, int order);
+extern int memcg_css_id(struct mem_cgroup *memcg);
+extern void memcg_init_kmem_cache(void);
+extern void memcg_register_cache(struct mem_cgroup *memcg,
+          struct kmem_cache *s);
+extern void memcg_release_cache(struct kmem_cache *cachep);
 #else
+
+static inline void memcg_init_kmem_cache(void)
+{
+}
+
+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..dc6daac 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 +341,10 @@ 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
+#endif
+
 #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 74654f0..04851bb 100644
--- a/mm/memcontrol.c
+++ b/mm/memcontrol.c
@@ -376,6 +376,11 @@ static bool memcg_kmem_dead(struct mem_cgroup *memcg)
 {
  return test_and_clear_bit(KMEM_ACCOUNTED_DEAD, &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. */
@@ -534,6 +539,29 @@ static inline bool memcg_can_account_kmem(struct mem_cgroup *memcg)
    (memcg->kmem_accounted & (KMEM_ACCOUNTED_MASK));
 }

+struct ida cache_types;
+
+void __init memcg_init_kmem_cache(void)
+{
+ ida_init(&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
diff --git a/mm/slab.c b/mm/slab.c
index 3bac667..e2cf984 100644
--- a/mm/slab.c
+++ b/mm/slab.c
@@ -1799,6 +1799,7 @@ void __init kmem_cache_init_late(void)
   BUG();
  mutex_unlock(&slab_mutex);

+ memcg_init_kmem_cache();
  /* Done! */
  slab_state = FULL;

diff --git a/mm/slab.h b/mm/slab.h
index 7a2698b..6f2a34d 100644
--- a/mm/slab.h
+++ b/mm/slab.h
@@ -35,12 +35,15 @@ extern struct kmem_cache *kmem_cache;
 /* Functions provided by the slab allocators */
 extern int __kmem_cache_create(struct kmem_cache *, unsigned long flags);

+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

@@ -49,4 +52,19 @@ int __kmem_cache_shutdown(struct kmem_cache *);

 int __kmem_cache_initcall(void);

+void __init memcg_slab_register_all(void);
+#ifdef CONFIG_MEMCG_KMEM
+static inline bool cache_match_memcg(struct kmem_cache *cachep,
+         struct mem_cgroup *memcg)
+{
+ return cachep->memcg_params.memcg == memcg;
+}
+
+#else
+static inline bool cache_match_memcg(struct kmem_cache *cachep,
+         struct mem_cgroup *memcg)
+{
+ return true;
+}
+#endif
 #endif
diff --git a/mm/slab_common.c b/mm/slab_common.c
index eddbb8a..8f06849 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"

@@ -25,7 +26,8 @@ DEFINE_MUTEX(slab_mutex);
 struct kmem_cache *kmem_cache;

 #ifdef CONFIG_DEBUG_VM
-static int kmem_cache_sanity_check(const char *name, size_t size)
+static int kmem_cache_sanity_check(struct mem_cgroup *memcg, const char *name,
+       size_t size)
```

```
 {
  struct kmem_cache *s = NULL;

@@ -51,7 +53,7 @@ static int kmem_cache_sanity_check(const char *name, size_t size)
    continue;
   }

-  if (!strcmp(s->name, name)) {
+  if (cache_match_memcg(s, memcg) && !strcmp(s->name, name)) {
    pr_err("%s (%s): Cache name already exists.\n",
           __func__, name);
    dump_stack();
@@ -64,7 +66,8 @@ static int kmem_cache_sanity_check(const char *name, size_t size)
  return 0;
 }
 #else
-static inline int kmem_cache_sanity_check(const char *name, size_t size)
+static inline int kmem_cache_sanity_check(struct mem_cgroup *memcg,
+       const char *name, size_t size)
 {
  return 0;
 }
@@ -95,8 +98,9 @@ static inline int kmem_cache_sanity_check(const char *name, size_t size)
  * 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;
  int err = 0;
@@ -104,11 +108,10 @@ struct kmem_cache *kmem_cache_create(const char *name, size_t
size, size_t align
  get_online_cpus();
  mutex_lock(&slab_mutex);

- if (!kmem_cache_sanity_check(name, size) == 0)
+ if (!kmem_cache_sanity_check(memcg, name, size) == 0)
   goto out_locked;

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

```
@@ -117,6 +120,9 @@ struct kmem_cache *kmem_cache_create(const char *name, size_t size, size_t align
   s->object_size = s->size = size;
   s->align = align;
   s->ctor = ctor;
+#ifdef CONFIG_MEMCG_KMEM
+  s->memcg_params.memcg = memcg;
+#endif
   s->name = kstrdup(name, GFP_KERNEL);
   if (!s->name) {
    kmem_cache_free(kmem_cache, s);
@@ -126,14 +132,14 @@ struct kmem_cache *kmem_cache_create(const char *name, size_t size, size_t align

   err = __kmem_cache_create(s, flags);
   if (!err) {
-
    s->refcount = 1;
    list_add(&s->list, &slab_caches);
-
   } else {
    kfree(s->name);
    kmem_cache_free(kmem_cache, s);
   }
+  if (slab_state >= FULL)
+   memcg_register_cache(memcg, s);
  } else
   err = -ENOMEM;

@@ -157,6 +163,13 @@ out_locked:

  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)
@@ -171,6 +184,7 @@ 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 {
@@ -192,6 +206,17 @@ 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 7ac46c6..4778548 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>

@@ -3864,7 +3865,7 @@ 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 *))
 {
@@ -3900,17 +3901,20 @@ 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++;
   /*
@@ -5230,6 +5234,10 @@ 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;
 }
@@ -5346,6 +5354,7 @@ int __init __kmem_cache_initcall(void)
   return -ENOSYS;
  }

+ memcg_init_kmem_cache();
  slab_state = FULL;

  list_for_each_entry(s, &slab_caches, list) {
--
1.7.11.4
```

Subject: [PATCH v3 06/16] memcg: infrastructure to match an allocation to the right cache
Posted by Glauber Costa on Tue, 18 Sep 2012 14:12:00 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            | 203 +++++++++++++++++++++++++++++++++++++++++++++
 3 files changed, 242 insertions(+), 1 deletion(-)

diff --git a/include/linux/memcontrol.h b/include/linux/memcontrol.h
index a5f3055..c44a5f2 100644
--- a/include/linux/memcontrol.h
+++ b/include/linux/memcontrol.h
@@ -419,6 +419,8 @@ extern void memcg_init_kmem_cache(void);
 extern void memcg_register_cache(struct mem_cgroup *memcg,
       struct kmem_cache *s);
 extern 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_init_kmem_cache(void)
@@ -460,6 +462,12 @@ static inline void
```

```
 __memcg_kmem_commit_charge(struct page *page, struct mem_cgroup *memcg, int order)
 {
 }
+
+static inline struct kmem_cache *
+__memcg_kmem_get_cache(struct kmem_cache *cachep, gfp_t gfp)
+{
+ return cachep;
+}
 #endif /* CONFIG_MEMCG_KMEM */

 /**
@@ -526,5 +534,35 @@ memcg_kmem_commit_charge(struct page *page, struct mem_cgroup
*memcg, int order)
  if (memcg_kmem_enabled() && memcg)
   __memcg_kmem_commit_charge(page, memcg, 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_enabled())
+  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 707d015..31c4f74 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 04851bb..1cce5c3 100644
--- a/mm/memcontrol.c
+++ b/mm/memcontrol.c
@@ -339,6 +339,11 @@ 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 {
@@ -539,6 +544,40 @@ static inline bool memcg_can_account_kmem(struct mem_cgroup
*memcg)
    (memcg->kmem_accounted & (KMEM_ACCOUNTED_MASK));
 }

+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 __init memcg_init_kmem_cache(void)
@@ -665,6 +704,170 @@ 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_can_account_kmem(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_can_account_kmem(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)
 {
--
1.7.11.4
```

Subject: [PATCH v3 07/16] memcg: skip memcg kmem allocations in specified code regions
Posted by Glauber Costa on Tue, 18 Sep 2012 14:12:01 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       | 23 +++++++++++++++++++++++
 2 files changed, 24 insertions(+)

diff --git a/include/linux/sched.h b/include/linux/sched.h
index 81967b1..4c983c6 100644
--- a/include/linux/sched.h
+++ b/include/linux/sched.h
@@ -1574,6 +1574,7 @@ 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 1cce5c3..54247ec 100644
--- a/mm/memcontrol.c
+++ b/mm/memcontrol.c
```

```
@@ -544,6 +544,22 @@ static inline bool memcg_can_account_kmem(struct mem_cgroup
*memcg)
    (memcg->kmem_accounted & (KMEM_ACCOUNTED_MASK));
 }

+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;
@@ -721,7 +737,9 @@ 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;
@@ -811,7 +829,9 @@ 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;
@@ -846,6 +866,9 @@ 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;

--
1.7.11.4
```

Subject: [PATCH v3 08/16] slab: allow enable_cpu_cache to use preset values for its tunables

Posted by Glauber Costa on Tue, 18 Sep 2012 14:12:02 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     |  6 ++++--
 4 files changed, 23 insertions(+), 7 deletions(-)

diff --git a/include/linux/slab.h b/include/linux/slab.h
index dc6daac..9d298db 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 54247ec..ee982aa 100644
--- a/mm/memcontrol.c
+++ b/mm/memcontrol.c
@@ -588,7 +588,7 @@ 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 e2cf984..f2d760c 100644
--- a/mm/slab.c
+++ b/mm/slab.c
@@ -4141,8 +4141,19 @@ 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
@@ -4184,7 +4195,9 @@ 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 8f06849..6829aa4 100644
--- a/mm/slab_common.c
+++ b/mm/slab_common.c
@@ -100,7 +100,8 @@ static inline int kmem_cache_sanity_check(struct mem_cgroup *memcg,

 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;
  int err = 0;
@@ -122,6 +123,7 @@ kmem_cache_create_memcg(struct mem_cgroup *memcg, const char
*name, size_t size,
   s->ctor = ctor;
 #ifdef CONFIG_MEMCG_KMEM
   s->memcg_params.memcg = memcg;
+  s->memcg_params.parent = parent_cache;
 #endif
   s->name = kstrdup(name, GFP_KERNEL);
   if (!s->name) {
@@ -168,7 +170,7 @@ 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);

--
1.7.11.4
```

Subject: [PATCH v3 09/16] sl[au]b: always get the cache from its page in kfree
Posted by Glauber Costa on Tue, 18 Sep 2012 14:12:03 GMT
View Forum Message <> Reply to Message

struct page already have this information. If we start chaining
caches, this information will always be more trustworthy than
whatever is passed into the function

A parent pointer is added to the slub structure, so we can make sure
the freeing comes from either the right slab, or from its rightful
parent.

[v3: added parent testing with VM_BUG_ON]

Signed-off-by: Glauber Costa <glommer@parallels.com>
CC: Christoph Lameter <cl@linux.com>
CC: Pekka Enberg <penberg@cs.helsinki.fi>

```
 mm/slab.c |  5 ++++-
 mm/slab.h | 11 +++++++++++
 mm/slub.c |  4 +++-
 3 files changed, 18 insertions(+), 2 deletions(-)

diff --git a/mm/slab.c b/mm/slab.c
index f2d760c..18de3f6 100644
--- a/mm/slab.c
+++ b/mm/slab.c
@@ -3938,9 +3938,12 @@ EXPORT_SYMBOL(__kmalloc);
  * Free an object which was previously allocated from this
  * cache.
  */
-void kmem_cache_free(struct kmem_cache *cachep, void *objp)
+void kmem_cache_free(struct kmem_cache *s, void *objp)
 {
  unsigned long flags;
+ struct kmem_cache *cachep = virt_to_cache(objp);
+
+ VM_BUG_ON(!slab_equal_or_parent(cachep, s));

  local_irq_save(flags);
  debug_check_no_locks_freed(objp, cachep->object_size);
diff --git a/mm/slab.h b/mm/slab.h
index 6f2a34d..f2501ab 100644
--- a/mm/slab.h
+++ b/mm/slab.h
@@ -60,11 +60,22 @@ static inline bool cache_match_memcg(struct kmem_cache *cachep,
  return cachep->memcg_params.memcg == memcg;
 }

+static inline bool slab_equal_or_parent(struct kmem_cache *s,
+     struct kmem_cache *p)
+{
+ return (p == s) || (p == s->memcg_params.parent);
+}
 #else
 static inline bool cache_match_memcg(struct kmem_cache *cachep,
          struct mem_cgroup *memcg)
 {
  return true;
 }
+
+static inline bool slab_equal_or_parent(struct kmem_cache *s,
+     struct kmem_cache *p)
+{
+ return true;
+}
 #endif
 #endif
diff --git a/mm/slub.c b/mm/slub.c
index 4778548..a045dfc 100644
--- a/mm/slub.c
+++ b/mm/slub.c
@@ -2604,7 +2604,9 @@ void kmem_cache_free(struct kmem_cache *s, void *x)

  page = virt_to_head_page(x);

- slab_free(s, page, x, _RET_IP_);
+ VM_BUG_ON(!slab_equal_or_parent(page->slab, s));
+
+ slab_free(page->slab, page, x, _RET_IP_);

  trace_kmem_cache_free(_RET_IP_, x);
 }
--
1.7.11.4
```

Subject: [PATCH v3 10/16] sl[au]b: Allocate objects from memcg cache
Posted by Glauber Costa on Tue, 18 Sep 2012 14:12:04 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 | 15 ++++++++++-----
 mm/memcontrol.c          |  9 +++++++++
 mm/slab.c                |  6 +++++-
 mm/slub.c                |  5 +++--
 4 files changed, 27 insertions(+), 8 deletions(-)

diff --git a/include/linux/slub_def.h b/include/linux/slub_def.h
index 8bb8ad2..76bf6da 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 +227,10 @@ 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 | __GFP_KMEMCG);
+ ret = (void *) __get_free_pages(flags, order);
  kmemleak_alloc(ret, size, 1, flags);
  return ret;
 }
@@ -274,7 +279,7 @@ 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 +312,7 @@ 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 ee982aa..0068b7d 100644
--- a/mm/memcontrol.c
+++ b/mm/memcontrol.c
@@ -477,7 +477,14 @@ struct mem_cgroup *mem_cgroup_from_css(struct
cgroup_subsys_state *s)
 #include <net/sock.h>
 #include <net/ip.h>

+/*
```

```
+ * A lot of the calls to the cache allocation functions are expected to be
+ * inlined by the compiler. Since the calls to memcg_kmem_get_cache are
+ * conditional to this static branch, we'll have to allow modules that does
+ * kmem_cache_alloc and the such to see this symbol as well
+ */
 struct static_key memcg_kmem_enabled_key;
+EXPORT_SYMBOL(memcg_kmem_enabled_key);

 static bool mem_cgroup_is_root(struct mem_cgroup *memcg);
 static int memcg_charge_kmem(struct mem_cgroup *memcg, gfp_t gfp, u64 size);
@@ -589,6 +596,8 @@ 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 18de3f6..f9c7e03 100644
--- a/mm/slab.c
+++ b/mm/slab.c
@@ -1971,7 +1971,7 @@ static void kmem_freepages(struct kmem_cache *cachep, void *addr)
  }
  if (current->reclaim_state)
   current->reclaim_state->reclaimed_slab += nr_freed;
- free_pages((unsigned long)addr, cachep->gfporder);
+ free_accounted_pages((unsigned long)addr, cachep->gfporder);
 }

 static void kmem_rcu_free(struct rcu_head *head)
@@ -3541,6 +3541,8 @@ __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);

@@ -3626,6 +3628,8 @@ __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 a045dfc..ac79595 100644
--- a/mm/slub.c
+++ b/mm/slub.c
@@ -1403,7 +1403,7 @@ static void __free_slab(struct kmem_cache *s, struct page *page)
  reset_page_mapcount(page);
  if (current->reclaim_state)
   current->reclaim_state->reclaimed_slab += pages;
- __free_pages(page, order);
+ __free_accounted_pages(page, order);
 }

 #define need_reserve_slab_rcu      \
@@ -2313,6 +2313,7 @@ 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:

  /*
@@ -3454,7 +3455,7 @@ void kfree(const void *x)
  if (unlikely(!PageSlab(page))) {
   BUG_ON(!PageCompound(page));
   kmemleak_free(x);
-  __free_pages(page, compound_order(page));
+  __free_accounted_pages(page, compound_order(page));
   return;
  }
  slab_free(page->slab, page, object, _RET_IP_);
--
1.7.11.4
```

Subject: [PATCH v3 11/16] memcg: destroy memcg caches
Posted by Glauber Costa on Tue, 18 Sep 2012 14:12:05 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                  |  3 ++
 mm/slab.h                  | 23 ++++++++++++++
 mm/slub.c                  |  7 +++-
 6 files changed, 120 insertions(+), 1 deletion(-)

diff --git a/include/linux/memcontrol.h b/include/linux/memcontrol.h
index c44a5f2..204a43a 100644
--- a/include/linux/memcontrol.h
+++ b/include/linux/memcontrol.h
@@ -421,6 +421,7 @@ extern void memcg_register_cache(struct mem_cgroup *memcg,
 extern 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_init_kmem_cache(void)
diff --git a/include/linux/slab.h b/include/linux/slab.h
index 9d298db..9a41d85 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 0068b7d..a184d42 100644
--- a/mm/memcontrol.c
```

```
+++ b/mm/memcontrol.c
@@ -614,6 +614,8 @@ 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);
@@ -758,6 +760,7 @@ 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;
@@ -772,6 +775,55 @@ 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.
@@ -793,6 +845,33 @@ 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;
@@ -904,6 +983,10 @@ 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)
@@ -5590,6 +5673,7 @@ 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 f9c7e03..d5c196b 100644
--- a/mm/slab.c
+++ b/mm/slab.c
@@ -1933,6 +1933,7 @@ static void *kmem_getpages(struct kmem_cache *cachep, gfp_t flags,
int nodeid)
   if (page->pfmemalloc)
    SetPageSlabPfmemalloc(page + i);
  }
+ mem_cgroup_bind_pages(cachep, cachep->gfporder);

  if (kmemcheck_enabled && !(cachep->flags & SLAB_NOTRACK)) {
   kmemcheck_alloc_shadow(page, cachep->gfporder, flags, nodeid);
@@ -1969,6 +1970,8 @@ 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_accounted_pages((unsigned long)addr, cachep->gfporder);
diff --git a/mm/slab.h b/mm/slab.h
index f2501ab..6024ad1 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
  */
@@ -60,6 +61,20 @@ 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);
+}
+
 static inline bool slab_equal_or_parent(struct kmem_cache *s,
      struct kmem_cache *p)
 {
@@ -72,6 +87,14 @@ 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)
```

```
+{
+}
+
 static inline bool slab_equal_or_parent(struct kmem_cache *s,
      struct kmem_cache *p)
 {
diff --git a/mm/slub.c b/mm/slub.c
index ac79595..f90f612 100644
--- a/mm/slub.c
+++ b/mm/slub.c
@@ -1342,6 +1342,7 @@ 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);

@@ -1350,7 +1351,9 @@ 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);
  if (page->pfmemalloc)
@@ -1359,7 +1362,7 @@ static struct page *new_slab(struct kmem_cache *s, gfp_t flags, int
node)
  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) {
@@ -1400,6 +1403,8 @@ static void __free_slab(struct kmem_cache *s, struct page *page)

  __ClearPageSlabPfmemalloc(page);
  __ClearPageSlab(page);
+
+ mem_cgroup_release_pages(s, order);
  reset_page_mapcount(page);
  if (current->reclaim_state)
  current->reclaim_state->reclaimed_slab += pages;
--
1.7.11.4
```

Subject: [PATCH v3 12/16] memcg/sl[au]b Track all the memcg children of a kmem_cache.

Posted by Glauber Costa on Tue, 18 Sep 2012 14:12:06 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 204a43a..6d5e212 100644
--- a/include/linux/memcontrol.h
+++ b/include/linux/memcontrol.h
@@ -422,6 +422,7 @@ extern 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_init_kmem_cache(void)
diff --git a/include/linux/slab.h b/include/linux/slab.h
index 9a41d85..9badb8c 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 a184d42..da38652 100644
--- a/mm/memcontrol.c
+++ b/mm/memcontrol.c
@@ -596,8 +596,11 @@ 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;
@@ -615,6 +618,7 @@ 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,
@@ -626,6 +630,9 @@ 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);
+
 }

 /*
@@ -934,6 +941,13 @@ 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 6829aa4..c6fb4a7 100644
--- a/mm/slab_common.c
+++ b/mm/slab_common.c
@@ -174,8 +174,39 @@ 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);
  s->refcount--;
--
1.7.11.4
```

Subject: [PATCH v3 13/16] slab: slab-specific propagation changes.
Posted by Glauber Costa on Tue, 18 Sep 2012 14:12:07 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 d5c196b..bd9928f 100644
--- a/mm/slab.c
+++ b/mm/slab.c
@@ -4104,7 +4104,7 @@ 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;
@@ -4147,6 +4147,32 @@ 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 c6fb4a7..341c3e1 100644
--- a/mm/slab_common.c
+++ b/mm/slab_common.c
@@ -124,6 +124,7 @@ kmem_cache_create_memcg(struct mem_cgroup *memcg, const char
*name, size_t size,
 #ifdef CONFIG_MEMCG_KMEM
   s->memcg_params.memcg = memcg;
   s->memcg_params.parent = parent_cache;
+  s->memcg_params.id = -1; /* not registered yet */
 #endif
   s->name = kstrdup(name, GFP_KERNEL);
   if (!s->name) {
--
1.7.11.4
```

Subject: [PATCH v3 14/16] slub: slub-specific propagation changes.
Posted by Glauber Costa on Tue, 18 Sep 2012 14:12:08 GMT
View Forum Message <> Reply to Message

When a parent cache changes a sysfs attr, we need to propagate that to

the children as well. For that, we unfortunately need to tap into the
slub 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/slub.c | 16 ++++++++++++++++
 1 file changed, 16 insertions(+)

diff --git a/mm/slub.c b/mm/slub.c
index f90f612..0b68d15 100644
--- a/mm/slub.c
+++ b/mm/slub.c
@@ -5174,6 +5174,10 @@ static ssize_t slab_attr_store(struct kobject *kobj,
  struct slab_attribute *attribute;
  struct kmem_cache *s;
  int err;
+#ifdef CONFIG_MEMCG_KMEM
+ struct kmem_cache *c;
+ struct mem_cgroup_cache_params *p;
+#endif

  attribute = to_slab_attr(attr);
  s = to_slab(kobj);
@@ -5182,7 +5186,19 @@ static ssize_t slab_attr_store(struct kobject *kobj,
   return -EIO;

  err = attribute->store(s, buf, len);
+#ifdef CONFIG_MEMCG_KMEM
+ if (slab_state < FULL)
+  return err;

+ if ((err < 0) || (s->memcg_params.id == -1))
+  return err;
+
+ list_for_each_entry(p, &s->memcg_params.sibling_list, sibling_list) {
+  c = container_of(p, struct kmem_cache, memcg_params);
+  /* return value determined by the parent cache only */
+  attribute->store(c, buf, len);
+ }
+#endif
  return err;
 }
```

--
1.7.11.4

Subject: [PATCH v3 15/16] memcg/sl[au]b: shrink dead caches
Posted by Glauber Costa on Tue, 18 Sep 2012 14:12:09 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 9badb8c..765e12c 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 da38652..c0cf564 100644
--- a/mm/memcontrol.c
+++ b/mm/memcontrol.c
@@ -578,7 +578,7 @@ 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;
@@ -739,12 +739,25 @@ 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_can_account_kmem(memcg));

@@ -764,10 +777,21 @@ 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;
@@ -790,6 +814,21 @@ 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) {
@@ -867,11 +906,14 @@ 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 bd9928f..6cb4abf 100644
--- a/mm/slab.c
+++ b/mm/slab.c
@@ -3785,6 +3785,8 @@ static inline void __cache_free(struct kmem_cache *cachep, void
*objp,
  }

  ac_put_obj(cachep, ac, objp);
+
+ kmem_cache_verify_dead(cachep);
 }

 /**
diff --git a/mm/slab.h b/mm/slab.h
index 6024ad1..d21b982 100644
--- a/mm/slab.h
+++ b/mm/slab.h
@@ -80,6 +80,12 @@ static inline bool slab_equal_or_parent(struct kmem_cache *s,
 {
  return (p == s) || (p == s->memcg_params.parent);
 }
+
+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 cache_match_memcg(struct kmem_cache *cachep,
         struct mem_cgroup *memcg)
@@ -100,5 +106,9 @@ static inline bool slab_equal_or_parent(struct kmem_cache *s,
 {
  return true;
 }
+
+static inline void kmem_cache_verify_dead(struct kmem_cache *s)
+{
+}
 #endif
 #endif
diff --git a/mm/slub.c b/mm/slub.c
index 0b68d15..9d79216 100644
--- a/mm/slub.c
+++ b/mm/slub.c
@@ -2602,6 +2602,7 @@ redo:
  } else
   __slab_free(s, page, x, addr);

+ kmem_cache_verify_dead(s);
 }

 void kmem_cache_free(struct kmem_cache *s, void *x)
--
1.7.11.4
```

Subject: [PATCH v3 16/16] Add documentation about the kmem controller
Posted by Glauber Costa on Tue, 18 Sep 2012 14:12:10 GMT

View Forum Message <> Reply to Message

Signed-off-by: Glauber Costa <glommer@parallels.com>
CC: Randy Dunlap <rdunlap@xenotime.net>
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>
CC: Mel Gorman <mgorman@suse.de>

```
---
 Documentation/cgroups/memory.txt | 73 +++++++++++++++++++++++++++++++++++++++++-
 1 file changed, 72 insertions(+), 1 deletion(-)

diff --git a/Documentation/cgroups/memory.txt b/Documentation/cgroups/memory.txt
index 4372e6b..6356379 100644
--- a/Documentation/cgroups/memory.txt
+++ b/Documentation/cgroups/memory.txt
@@ -71,6 +71,11 @@ Brief summary of control files.
  memory.oom_control   # set/show oom controls.
  memory.numa_stat   # show the number of memory usage per numa node

+ memory.kmem.limit_in_bytes      # set/show hard limit for kernel memory
+ memory.kmem.usage_in_bytes      # show current kernel memory allocation
+ memory.kmem.failcnt             # show the number of kernel memory usage hits limits
+ memory.kmem.max_usage_in_bytes  # show max kernel memory usage recorded
+
  memory.kmem.tcp.limit_in_bytes  # set/show hard limit for tcp buf memory
  memory.kmem.tcp.usage_in_bytes  # show current tcp buf memory allocation
  memory.kmem.tcp.failcnt            # show the number of tcp buf memory usage hits limits
@@ -268,20 +273,80 @@ the amount of kernel memory used by the system. Kernel memory is
fundamentally
 different than user memory, since it can't be swapped out, which makes it
 possible to DoS the system by consuming too much of this precious resource.

+Kernel memory won't be accounted at all until it is limited. This allows for
+existing setups to continue working without disruption. Note that it is
+possible to account it without an effective limit by setting the limits
+to a very high number (like RESOURCE_MAX -1page). After a controller is first
+limited, it will be kept being accounted until it is removed. The memory
+limitation itself, can of course be removed by writing -1 to
+memory.kmem.limit_in_bytes
+
 Kernel memory limits are not imposed for the root cgroup. Usage for the root
-cgroup may or may not be accounted.
+cgroup may or may not be accounted. The memory used is accumulated into
+memory.kmem.usage_in_bytes, or in a separate counter when it makes sense.
+The main "kmem" counter is fed into the main counter, so kmem charges will
+also be visible from the user counter.

 Currently no soft limit is implemented for kernel memory. It is future work
 to trigger slab reclaim when those limits are reached.

 2.7.1 Current Kernel Memory resources accounted

+* stack pages: every process consumes some stack pages. By accounting into
+kernel memory, we prevent new processes from being created when the kernel
+memory usage is too high.
```

```
+
+* slab pages: pages allocated by the SLAB or SLUB allocator are tracked. A copy
+of each kmem_cache is created everytime the cache is touched by the first time
+from inside the memcg. The creation is done lazily, so some objects can still be
+skipped while the cache is being created. All objects in a slab page should
+belong to the same memcg. This only fails to hold when a task is migrated to a
+different memcg during the page allocation by the cache.
+
 * sockets memory pressure: some sockets protocols have memory pressure
 thresholds. The Memory Controller allows them to be controlled individually
 per cgroup, instead of globally.

 * tcp memory pressure: sockets memory pressure for the tcp protocol.

+2.7.2 The kmem_accounted field (internal, non user visible)
+
+The bitmap "kmem_accounted" is responsible for controlling the lifecycle of
+kmem accounting in a particular memcg. The bits have the following meaning:
+
+bit0, KMEM_ACCOUNTED_ACTIVE: will be set to indicate that the memcg is kmem
+	limited due to userspace action directly to this memcg.
+bit1, KMEM_ACCOUNTED_PARENT: only valid in hierarchical setups, will trigger
+	kernel memory accounting because any ancestor of this group was accounted.
+bit2, KMEM_ACCOUNTED_DEAD: will be set when a memcg is removed, and pending kmem
+	charges are still present. When the memcg can be removed without delays,
+	it will be, and this bit will never be set.
+
+2.7.3 Common use cases
+
+Because the "kmem" counter is fed to the main user counter, kernel memory can
+never be limited completely independently of user memory. Say "U" is the user
+limit, and "K" the kernel limit. There are three possible ways limits can be
+set:
+
+    U != 0, K = 0:
+    This is the standard memcg limitation mechanism already present before kmem
+    accounting. Kernel memory is completely ignored.
+
+    U,K != 0, K < U:
+    Kernel memory is effectively set as a percentage of the user memory. This
+    setup is useful in deployments where the total amount of memory per-cgroup
+    is overcommited. Overcommiting kernel memory limits is definitely not
+    recommended, since the box can still run out of non-reclaimable memory.
+    In this case, the admin could set up K so that the sum of all groups is
+    never greater than the total memory, and freely set U at the cost of his
+    QoS.
+
+    U,K != 0, K >= U:
```

```
+    Since kmem charges will also be fed to the user counter, this setup gives
+    the admin a unified view of memory. Reclaim will be triggered for the cgroup
+    for both kinds of memory.
+
 3. User Interface

 0. Configuration
@@ -290,6 +355,7 @@ a. Enable CONFIG_CGROUPS
 b. Enable CONFIG_RESOURCE_COUNTERS
 c. Enable CONFIG_MEMCG
 d. Enable CONFIG_MEMCG_SWAP (to use swap extension)
+d. Enable CONFIG_MEMCG_KMEM (to use kmem extension)

 1. Prepare the cgroups (see cgroups.txt, Why are cgroups needed?)
 # mount -t tmpfs none /sys/fs/cgroup
@@ -402,6 +468,11 @@ About use_hierarchy, see Section 6.
   moved to parent(if use_hierarchy==1) or root (if use_hierarchy==0) and this
   cgroup will be empty.

+  Also, note that when memory.kmem.limit_in_bytes is set the charges due to
+  kernel pages will still be seen. This is not considered a failure and the
+  write will still return success. In this case, it is expected that
+  memory.kmem.usage_in_bytes == memory.usage_in_bytes.
+
   Typical use case of this interface is that calling this before rmdir().
   Because rmdir() moves all pages to parent, some out-of-use page caches can be
   moved to the parent. If you want to avoid that, force_empty will be useful.
--
1.7.11.4
```

Subject: Re: [PATCH v3 03/16] slab: Ignore the cflgs bit in cache creation
Posted by Christoph Lameter on Tue, 18 Sep 2012 15:20:11 GMT

View Forum Message <> Reply to Message

On Tue, 18 Sep 2012, Glauber Costa wrote:

> No cache should ever pass that as a creation flag, since this bit is
> used to mark an internal decision of the slab about object placement. We
> can just ignore this bit if it happens to be passed (such as when
> duplicating a cache in the kmem memcg patches)

If we do this then I would like to see a general masking of internal allocator bits in kmem_cache_create. We could declare the highest byte to be the internal slab flags. SLUB uses two flags in that area. SLAB uses one.

F.e. add

#define SLAB_INTERNAL 0xFF00000000UL

to slab.h.

Then the flags can then be masked in mm/slab_common.c

Subject: Re: [PATCH v3 04/16] provide a common place for initcall processing in kmem_cache
Posted by Christoph Lameter on Tue, 18 Sep 2012 15:22:00 GMT

View Forum Message <> Reply to Message

On Tue, 18 Sep 2012, Glauber Costa wrote:

> Both SLAB and SLUB depend on some initialization to happen when the
> system is already booted, with all subsystems working. This is done
> by issuing an initcall that does the final initialization.
>
> This patch moves that to slab_common.c, while creating an empty
> placeholder for the SLOB.

Acked-by: Christoph Lameter <cl@linux.com>

Subject: Re: [PATCH v3 08/16] slab: allow enable_cpu_cache to use preset values for its tunables
Posted by Christoph Lameter on Tue, 18 Sep 2012 15:25:54 GMT

View Forum Message <> Reply to Message

On Tue, 18 Sep 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.

Again the same is true for SLUB. Some generic way of preserving tuning
parameters would be appreciated.

Subject: Re: [PATCH v3 09/16] sl[au]b: always get the cache from its page in kfree
Posted by Christoph Lameter on Tue, 18 Sep 2012 15:28:19 GMT

View Forum Message <> Reply to Message

On Tue, 18 Sep 2012, Glauber Costa wrote:

> index f2d760c..18de3f6 100644

```
> --- a/mm/slab.c
> +++ b/mm/slab.c
> @@ -3938,9 +3938,12 @@ EXPORT_SYMBOL(__kmalloc);
>   * Free an object which was previously allocated from this
>   * cache.
>   */
> -void kmem_cache_free(struct kmem_cache *cachep, void *objp)
> +void kmem_cache_free(struct kmem_cache *s, void *objp)
>  {
>   unsigned long flags;
> + struct kmem_cache *cachep = virt_to_cache(objp);
> +
> + VM_BUG_ON(!slab_equal_or_parent(cachep, s));
>
```

This is an extremely hot path of the kernel and you are adding significant
processing. Check how the benchmarks are influenced by this change.
virt_to_cache can be a bit expensive.

```
> diff --git a/mm/slub.c b/mm/slub.c
> index 4778548..a045dfc 100644
> --- a/mm/slub.c
> +++ b/mm/slub.c
> @@ -2604,7 +2604,9 @@ void kmem_cache_free(struct kmem_cache *s, void *x)
>
>   page = virt_to_head_page(x);
>
> - slab_free(s, page, x, _RET_IP_);
> + VM_BUG_ON(!slab_equal_or_parent(page->slab, s));
> +
> + slab_free(page->slab, page, x, _RET_IP_);
>
```

Less of a problem here but you are eroding one advantage that slab has had
in the past over slub in terms of freeing objects.

Subject: Re: [PATCH v3 13/16] slab: slab-specific propagation changes.
Posted by Christoph Lameter on Tue, 18 Sep 2012 17:00:20 GMT

View Forum Message <> Reply to Message

On Tue, 18 Sep 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.

One of the todo list items for the common stuff is to have actually a
common kmem_cache structure. If we add a common callback then we could put

that also into the core.

Subject: Re: [PATCH v3 15/16] memcg/sl[au]b: shrink dead caches
Posted by Christoph Lameter on Tue, 18 Sep 2012 17:02:19 GMT
View Forum Message <> Reply to Message

Why doesnt slab need that too? It keeps a number of free pages on the per
node lists until shrink is called.

Subject: Re: [PATCH v3 15/16] memcg/sl[au]b: shrink dead caches
Posted by Glauber Costa on Wed, 19 Sep 2012 07:40:26 GMT
View Forum Message <> Reply to Message

On 09/18/2012 09:02 PM, Christoph Lameter wrote:
> Why doesnt slab need that too? It keeps a number of free pages on the per
> node lists until shrink is called.
>
You have already given me this feedback, and I forgot to include it
here. I am sorry for this slip. It was my intention to this for the slab
as well.

Thanks for the eyes!

Subject: Re: [PATCH v3 13/16] slab: slab-specific propagation changes.
Posted by Glauber Costa on Wed, 19 Sep 2012 07:41:30 GMT
View Forum Message <> Reply to Message

On 09/18/2012 09:00 PM, Christoph Lameter wrote:
> On Tue, 18 Sep 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.
>
> One of the todo list items for the common stuff is to have actually a
> common kmem_cache structure. If we add a common callback then we could put
> that also into the core.
>

At this point, I'd like to keep it this way if possible. I agree that it
would be a lot better living in a common location, and I commit to
helping you to move in that direction.

But I see no reason to have this commonization to happen first, as a
pre-requisite for this.

Subject: Re: [PATCH v3 09/16] sl[au]b: always get the cache from its page in kfree
Posted by Glauber Costa on Wed, 19 Sep 2012 07:42:54 GMT
View Forum Message <> Reply to Message

On 09/18/2012 07:28 PM, Christoph Lameter wrote:

```
> On Tue, 18 Sep 2012, Glauber Costa wrote:
>
>> index f2d760c..18de3f6 100644
>> --- a/mm/slab.c
>> +++ b/mm/slab.c
>> @@ -3938,9 +3938,12 @@ EXPORT_SYMBOL(__kmalloc);
>>   * Free an object which was previously allocated from this
>>   * cache.
>>   */
>> -void kmem_cache_free(struct kmem_cache *cachep, void *objp)
>> +void kmem_cache_free(struct kmem_cache *s, void *objp)
>>  {
>>   unsigned long flags;
>> + struct kmem_cache *cachep = virt_to_cache(objp);
>> +
>> + VM_BUG_ON(!slab_equal_or_parent(cachep, s));
>>
>
> This is an extremely hot path of the kernel and you are adding significant
> processing. Check how the benchmarks are influenced by this change.
> virt_to_cache can be a bit expensive.
>
```

Would it be enough for you to have a separate code path for
!CONFIG_MEMCG_KMEM?

I don't really see another way to do it, aside from deriving the cache
from the object in our case. I am open to suggestions if you do.

```
>> diff --git a/mm/slub.c b/mm/slub.c
>> index 4778548..a045dfc 100644
>> --- a/mm/slub.c
>> +++ b/mm/slub.c
>> @@ -2604,7 +2604,9 @@ void kmem_cache_free(struct kmem_cache *s, void *x)
>>
>>   page = virt_to_head_page(x);
>>
>> - slab_free(s, page, x, _RET_IP_);
>> + VM_BUG_ON(!slab_equal_or_parent(page->slab, s));
>> +
>> + slab_free(page->slab, page, x, _RET_IP_);
>>
>
> Less of a problem here but you are eroding one advantage that slab has had
```

> in the past over slub in terms of freeing objects.
>
likewise.

Subject: Re: [PATCH v3 08/16] slab: allow enable_cpu_cache to use preset values for its tunables
Posted by Glauber Costa on Wed, 19 Sep 2012 07:44:24 GMT

View Forum Message <> Reply to Message

On 09/18/2012 07:25 PM, Christoph Lameter wrote:
> On Tue, 18 Sep 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.
>
> Again the same is true for SLUB. Some generic way of preserving tuning
> parameters would be appreciated.

So you would like me to extend "slub: slub-specific propagation changes"
to also allow for pre-set values, right?

Subject: Re: [PATCH v3 09/16] sl[au]b: always get the cache from its page in kfree
Posted by Christoph Lameter on Wed, 19 Sep 2012 14:14:23 GMT

View Forum Message <> Reply to Message

On Wed, 19 Sep 2012, Glauber Costa wrote:

> > This is an extremely hot path of the kernel and you are adding significant
> > processing. Check how the benchmarks are influenced by this change.
> > virt_to_cache can be a bit expensive.
> Would it be enough for you to have a separate code path for
> !CONFIG_MEMCG_KMEM?

Yes, at least add an #ifdef around this.

> I don't really see another way to do it, aside from deriving the cache
> from the object in our case. I am open to suggestions if you do.

Rethink the whole memcg approach and find some other way to do it? This
whole scheme is very intrusive and is likely to increase instability in
the VM given the explosion of lru lists, duplication of slab caches and
significantly more complex processing throughout the VM.

Subject: Re: [PATCH v3 15/16] memcg/sl[au]b: shrink dead caches
Posted by JoonSoo Kim on Fri, 21 Sep 2012 04:48:30 GMT

View Forum Message <> Reply to Message

Hi Glauber.

2012/9/18 Glauber Costa <glommer@parallels.com>:

```
> diff --git a/mm/slub.c b/mm/slub.c
> index 0b68d15..9d79216 100644
> --- a/mm/slub.c
> +++ b/mm/slub.c
> @@ -2602,6 +2602,7 @@ redo:
>        } else
>                __slab_free(s, page, x, addr);
>
> +       kmem_cache_verify_dead(s);
>  }
```

As far as u know, I am not a expert and don't know anything about memcg.
IMHO, this implementation may hurt system performance in some case.

In case of memcg is destoried, remained kmem_cache is marked "dead".
After it is marked,
every free operation to this "dead" kmem_cache call
kmem_cache_verify_dead() and finally call kmem_cache_shrink().
kmem_cache_shrink() do invoking kmalloc and flush_all() and taking a
lock for online node and invoking kfree.
Especially, flush_all() may hurt performance largely, because it call
has_cpu_slab() against all the cpus.

And I know some other case it can hurt system performance.
But, I don't mention it, because above case is sufficient to worry.

And, I found one case that destroying memcg's kmem_cache don't works properly.
If we destroy memcg after all object is freed, current implementation
doesn't destroy kmem_cache.
kmem_cache_destroy_work_func() check "cachep->memcg_params.nr_pages == 0",
but in this case, it return false, because kmem_cache may have
cpu_slab, and cpu_partials_slabs.
As we already free all objects, kmem_cache_verify_dead() is not invoked forever.
I think that we need another kmem_cache_shrink() in
kmem_cache_destroy_work_func().

I don't convince that I am right, so think carefully my humble opinion.

Thanks.

Subject: Re: [PATCH v3 15/16] memcg/sl[au]b: shrink dead caches
Posted by Glauber Costa on Fri, 21 Sep 2012 08:40:04 GMT
View Forum Message <> Reply to Message

On 09/21/2012 08:48 AM, JoonSoo Kim wrote:
> Hi Glauber.
>
Hi

> 2012/9/18 Glauber Costa <glommer@parallels.com>:
```
>> diff --git a/mm/slub.c b/mm/slub.c
>> index 0b68d15..9d79216 100644
>> --- a/mm/slub.c
>> +++ b/mm/slub.c
>> @@ -2602,6 +2602,7 @@ redo:
>>         } else
>>                 __slab_free(s, page, x, addr);
>>
>> +       kmem_cache_verify_dead(s);
>>  }
```
>
> As far as u know, I am not a expert and don't know anything about memcg.
> IMHO, this implementation may hurt system performance in some case.
>
> In case of memcg is destoried, remained kmem_cache is marked "dead".
> After it is marked,
> every free operation to this "dead" kmem_cache call
> kmem_cache_verify_dead() and finally call kmem_cache_shrink().

As long as it is restricted to that cache, this is a non issue.
dead caches are exactly what they name imply: dead.

Means that we actively want them to go away, and just don't kill them
right away because they have some inflight objects - which we expect not
to be too much.

> kmem_cache_shrink() do invoking kmalloc and flush_all() and taking a
> lock for online node and invoking kfree.
> Especially, flush_all() may hurt performance largely, because it call
> has_cpu_slab() against all the cpus.

Again, this is all right, but being a dead cache, it shouldn't be on any
hot path.

>
> And, I found one case that destroying memcg's kmem_cache don't works properly.
> If we destroy memcg after all object is freed, current implementation
> doesn't destroy kmem_cache.
> kmem_cache_destroy_work_func() check "cachep->memcg_params.nr_pages == 0",

> but in this case, it return false, because kmem_cache may have
> cpu_slab, and cpu_partials_slabs.
> As we already free all objects, kmem_cache_verify_dead() is not invoked forever.
> I think that we need another kmem_cache_shrink() in
> kmem_cache_destroy_work_func().

I'll take a look here. What you describe makes sense, and can
potentially happen. I tried to handle this case with care in
destroy_all_caches, but I may have always made a mistake...

Did you see this actively happening, or are you just assuming this can
happen from your read of the code?

Subject: Re: [PATCH v3 15/16] memcg/sl[au]b: shrink dead caches
Posted by JoonSoo Kim on Fri, 21 Sep 2012 09:28:02 GMT

View Forum Message <> Reply to Message

Hi, Glauber.

```
>> 2012/9/18 Glauber Costa <glommer@parallels.com>:
>>> diff --git a/mm/slub.c b/mm/slub.c
>>> index 0b68d15..9d79216 100644
>>> --- a/mm/slub.c
>>> +++ b/mm/slub.c
>>> @@ -2602,6 +2602,7 @@ redo:
>>>         } else
>>>                 __slab_free(s, page, x, addr);
>>>
>>> +       kmem_cache_verify_dead(s);
>>>  }
>>
>> As far as u know, I am not a expert and don't know anything about memcg.
>> IMHO, this implementation may hurt system performance in some case.
>>
>> In case of memcg is destoried, remained kmem_cache is marked "dead".
>> After it is marked,
>> every free operation to this "dead" kmem_cache call
>> kmem_cache_verify_dead() and finally call kmem_cache_shrink().
>
> As long as it is restricted to that cache, this is a non issue.
> dead caches are exactly what they name imply: dead.
>
> Means that we actively want them to go away, and just don't kill them
> right away because they have some inflight objects - which we expect not
> to be too much.
```

Hmm.. I don't think so.

We can destroy memcg whenever we want, is it right?
If it is right, there is many inflight objects when we destory memcg.
If there is so many inflight objects, performance of these processes
can be hurt too much.

>> And, I found one case that destroying memcg's kmem_cache don't works properly.
>> If we destroy memcg after all object is freed, current implementation
>> doesn't destroy kmem_cache.
>> kmem_cache_destroy_work_func() check "cachep->memcg_params.nr_pages == 0",
>> but in this case, it return false, because kmem_cache may have
>> cpu_slab, and cpu_partials_slabs.
>> As we already free all objects, kmem_cache_verify_dead() is not invoked forever.
>> I think that we need another kmem_cache_shrink() in
>> kmem_cache_destroy_work_func().
>
> I'll take a look here. What you describe makes sense, and can
> potentially happen. I tried to handle this case with care in
> destroy_all_caches, but I may have always made a mistake...
>
> Did you see this actively happening, or are you just assuming this can
> happen from your read of the code?

Just read of the code.

Thanks.

Subject: Re: [PATCH v3 08/16] slab: allow enable_cpu_cache to use preset values for its tunables

Posted by Pekka Enberg on Fri, 21 Sep 2012 09:29:20 GMT

View Forum Message <> Reply to Message

On Tue, Sep 18, 2012 at 5:12 PM, Glauber Costa <glommer@parallels.com> wrote:

```
> diff --git a/mm/slab.c b/mm/slab.c
> index e2cf984..f2d760c 100644
> --- a/mm/slab.c
> +++ b/mm/slab.c
> @@ -4141,8 +4141,19 @@ 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;
```

Style nit: please introduce a variable for
"cachep->memcg_params.parent" to make this human-readable.

Subject: Re: [PATCH v3 09/16] sl[au]b: always get the cache from its page in kfree
Posted by Glauber Costa on Fri, 21 Sep 2012 09:30:59 GMT
View Forum Message <> Reply to Message

On 09/21/2012 01:33 PM, Pekka Enberg wrote:

```
> On Wed, Sep 19, 2012 at 10:42 AM, Glauber Costa <glommer@parallels.com> wrote:
>>>> index f2d760c..18de3f6 100644
>>>> --- a/mm/slab.c
>>>> +++ b/mm/slab.c
>>>> @@ -3938,9 +3938,12 @@ EXPORT_SYMBOL(__kmalloc);
>>>>   * Free an object which was previously allocated from this
>>>>   * cache.
>>>>   */
>>>> -void kmem_cache_free(struct kmem_cache *cachep, void *objp)
>>>> +void kmem_cache_free(struct kmem_cache *s, void *objp)
>>>>  {
>>>>     unsigned long flags;
>>>> +    struct kmem_cache *cachep = virt_to_cache(objp);
>>>> +
>>>> +    VM_BUG_ON(!slab_equal_or_parent(cachep, s));
>>>
>>> This is an extremely hot path of the kernel and you are adding significant
>>> processing. Check how the benchmarks are influenced by this change.
>>> virt_to_cache can be a bit expensive.
>>
>> Would it be enough for you to have a separate code path for
>> !CONFIG_MEMCG_KMEM?
>>
>> I don't really see another way to do it, aside from deriving the cache
>> from the object in our case. I am open to suggestions if you do.
>
> We should assume that most distributions enable CONFIG_MEMCG_KMEM,
> right? Therfore, any performance impact should be dependent on whether
> or not kmem memcg is *enabled* at runtime or not.
>
> Can we use the "static key" thingy introduced by tracing folks for this?
>
```

Yes.

I am already using static keys extensively in this patchset, and that is
how I intend to handle this particular case.

Subject: Re: [PATCH v3 15/16] memcg/sl[au]b: shrink dead caches
Posted by Glauber Costa on Fri, 21 Sep 2012 09:31:55 GMT

View Forum Message <> Reply to Message

On 09/21/2012 01:28 PM, JoonSoo Kim wrote:
> Hi, Glauber.
>
>>> 2012/9/18 Glauber Costa <glommer@parallels.com>:
>>>> diff --git a/mm/slub.c b/mm/slub.c
>>>> index 0b68d15..9d79216 100644
>>>> --- a/mm/slub.c
>>>> +++ b/mm/slub.c
>>>> @@ -2602,6 +2602,7 @@ redo:
>>>>   } else
>>>>   __slab_free(s, page, x, addr);
>>>>
>>>> +   kmem_cache_verify_dead(s);
>>>> }
>>>
>>> As far as u know, I am not a expert and don't know anything about memcg.
>>> IMHO, this implementation may hurt system performance in some case.
>>>
>>> In case of memcg is destoried, remained kmem_cache is marked "dead".
>>> After it is marked,
>>> every free operation to this "dead" kmem_cache call
>>> kmem_cache_verify_dead() and finally call kmem_cache_shrink().
>>
>> As long as it is restricted to that cache, this is a non issue.
>> dead caches are exactly what they name imply: dead.
>>
>> Means that we actively want them to go away, and just don't kill them
>> right away because they have some inflight objects - which we expect not
>> to be too much.
>
> Hmm.. I don't think so.
> We can destroy memcg whenever we want, is it right?

wrong.

it is impossible to track objects to a task, so when tasks are moved,
objects stay. Which means that some objects are still referenced by the
cache.
> If it is right, there is many inflight objects when we destory memcg.
> If there is so many inflight objects, performance of these processes

> can be hurt too much.
>
There are in-flight objects. It is not expected to have "many inflight
objects", which is a different statement.

We are assuming all the time that the workloads that goes into a cgroup
are mostly nature. When it goes away, most of the objects that were
referenced are expected to go away. Because we can't guarantee all of
them will, the references stay.

When we are able to call shrink_slab() directly on a memcg slab, which
is WIP, we'll be able make those objects even rarer.

>>> And, I found one case that destroying memcg's kmem_cache don't works properly.
>>> If we destroy memcg after all object is freed, current implementation
>>> doesn't destroy kmem_cache.
>>> kmem_cache_destroy_work_func() check "cachep->memcg_params.nr_pages == 0",
>>> but in this case, it return false, because kmem_cache may have
>>> cpu_slab, and cpu_partials_slabs.
>>> As we already free all objects, kmem_cache_verify_dead() is not invoked forever.
>>> I think that we need another kmem_cache_shrink() in
>>> kmem_cache_destroy_work_func().
>>
>> I'll take a look here. What you describe makes sense, and can
>> potentially happen. I tried to handle this case with care in
>> destroy_all_caches, but I may have always made a mistake...
>>
>> Did you see this actively happening, or are you just assuming this can
>> happen from your read of the code?
>
> Just read of the code.
>

I will go through it again, just in case. This is indeed subtle and it
is in my best interest to sort out those issues early.

Subject: Re: [PATCH v3 09/16] sl[au]b: always get the cache from its page in kfree
Posted by Pekka Enberg on Fri, 21 Sep 2012 09:33:23 GMT

View Forum Message <> Reply to Message

On Wed, Sep 19, 2012 at 10:42 AM, Glauber Costa <glommer@parallels.com> wrote:
>>> index f2d760c..18de3f6 100644
>>> --- a/mm/slab.c
>>> +++ b/mm/slab.c
>>> @@ -3938,9 +3938,12 @@ EXPORT_SYMBOL(__kmalloc);
>>>   * Free an object which was previously allocated from this
>>>   * cache.

>>>   */
>>> -void kmem_cache_free(struct kmem_cache *cachep, void *objp)
>>> +void kmem_cache_free(struct kmem_cache *s, void *objp)
>>> {
>>>   unsigned long flags;
>>> +   struct kmem_cache *cachep = virt_to_cache(objp);
>>> +
>>> +   VM_BUG_ON(!slab_equal_or_parent(cachep, s));
>>
>> This is an extremely hot path of the kernel and you are adding significant
>> processing. Check how the benchmarks are influenced by this change.
>> virt_to_cache can be a bit expensive.
>
> Would it be enough for you to have a separate code path for
> !CONFIG_MEMCG_KMEM?
>
> I don't really see another way to do it, aside from deriving the cache
> from the object in our case. I am open to suggestions if you do.

We should assume that most distributions enable CONFIG_MEMCG_KMEM,
right? Therfore, any performance impact should be dependent on whether
or not kmem memcg is *enabled* at runtime or not.

Can we use the "static key" thingy introduced by tracing folks for this?

Subject: Re: [PATCH v3 00/16] slab accounting for memcg
Posted by Pekka Enberg on Fri, 21 Sep 2012 09:40:15 GMT

View Forum Message <> Reply to Message

Hi Glauber,

On Tue, Sep 18, 2012 at 5:11 PM, Glauber Costa <glommer@parallels.com> wrote:
> This is a followup to the previous kmem series. I divided them logically
> so it gets easier for reviewers. But I believe they are ready to be merged
> together (although we can do a two-pass merge if people would prefer)
>
> Throwaway git tree found at:
>
>   git://git.kernel.org/pub/scm/linux/kernel/git/glommer/memcg. git kmemcg-slab
>
> There are mostly bugfixes since last submission.

Overall, I like this series a lot. However, I don't really see this as a
v3.7 material because we already have largeish pending updates to the
slab allocators. I also haven't seen any performance numbers for this
which is a problem.

So what I'd really like to see is this series being merged early in the
v3.8 development cycle to maximize the number of people eyeballing the
code and looking at performance impact.

Does this sound reasonable to you Glauber?

Pekka

Subject: Re: [PATCH v3 09/16] sl[au]b: always get the cache from its page in kfree
Posted by Pekka Enberg on Fri, 21 Sep 2012 09:41:52 GMT

View Forum Message <> Reply to Message

On Fri, 21 Sep 2012, Glauber Costa wrote:
> > We should assume that most distributions enable CONFIG_MEMCG_KMEM,
> > right? Therfore, any performance impact should be dependent on whether
> > or not kmem memcg is *enabled* at runtime or not.
> >
> > Can we use the "static key" thingy introduced by tracing folks for this?
>
> Yes.
>
> I am already using static keys extensively in this patchset, and that is
> how I intend to handle this particular case.

Cool.

The key point here is that !CONFIG_MEMCG_KMEM should have exactly *zero*
performance impact and CONFIG_MEMCG_KMEM disabled at runtime should have
absolute minimal impact.

Pekka

Subject: Re: [PATCH v3 00/16] slab accounting for memcg
Posted by Glauber Costa on Fri, 21 Sep 2012 09:43:48 GMT

View Forum Message <> Reply to Message

On 09/21/2012 01:40 PM, Pekka Enberg wrote:
> Hi Glauber,
>
> On Tue, Sep 18, 2012 at 5:11 PM, Glauber Costa <glommer@parallels.com> wrote:
>> This is a followup to the previous kmem series. I divided them logically
>> so it gets easier for reviewers. But I believe they are ready to be merged
>> together (although we can do a two-pass merge if people would prefer)
>>
>> Throwaway git tree found at:

>>
>>   git://git.kernel.org/pub/scm/linux/kernel/git/glommer/memcg. git kmemcg-slab
>>
>> There are mostly bugfixes since last submission.
>
> Overall, I like this series a lot. However, I don't really see this as a
> v3.7 material because we already have largeish pending updates to the
> slab allocators. I also haven't seen any performance numbers for this
> which is a problem.
>
> So what I'd really like to see is this series being merged early in the
> v3.8 development cycle to maximize the number of people eyeballing the
> code and looking at performance impact.
>
> Does this sound reasonable to you Glauber?

Absolutely.

As I've stated before, I actually believe the kmemcg-stack and
kmemcg-slab (this one) portions should be merged separately. (So we can
sort out issues more easily, and point to the right place)

The first one is a lot more stable and got a lot more love. The goal of
this one is to get it reviewed so we can merge as soon as we can - but
not sooner.

early v3.8 sounds perfect to me.

Subject: Re: [PATCH v3 03/16] slab: Ignore the cflgs bit in cache creation
Posted by Tejun Heo on Fri, 21 Sep 2012 17:33:52 GMT

View Forum Message <> Reply to Message

On Tue, Sep 18, 2012 at 06:11:57PM +0400, Glauber Costa wrote:
> No cache should ever pass that as a creation flag, since this bit is
> used to mark an internal decision of the slab about object placement. We
> can just ignore this bit if it happens to be passed (such as when
> duplicating a cache in the kmem memcg patches)
>
> Signed-off-by: Glauber Costa <glommer@parallels.com>
> CC: Christoph Lameter <cl@linux.com>
> CC: Pekka Enberg <penberg@cs.helsinki.fi>
> CC: David Rientjes <rientjes@google.com>
> ---
> mm/slab.c | 1 +
> 1 file changed, 1 insertion(+)
>
> diff --git a/mm/slab.c b/mm/slab.c

```
> index a7ed60f..ccf496c 100644
> --- a/mm/slab.c
> +++ b/mm/slab.c
> @@ -2373,6 +2373,7 @@ __kmem_cache_create (struct kmem_cache *cachep, unsigned long
flags)
>   int err;
>   size_t size = cachep->size;
>
> + flags &= ~CFLGS_OFF_SLAB;
```

A comment explaining why this is necessary wouldn't hurt.

Thanks.

--
tejun

Subject: Re: [PATCH v3 05/16] consider a memcg parameter in kmem_create_cache
Posted by Tejun Heo on Fri, 21 Sep 2012 18:14:58 GMT

View Forum Message <> Reply to Message

Hello, Glauber.

On Tue, Sep 18, 2012 at 06:11:59PM +0400, Glauber Costa wrote:

```
> +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;
> +}
```

I'm a bit confused. Why is id allocated only when memcg is NULL?

Also, how would the per-memcg slab/slubs appear in slabinfo? If they
appear separately it might be better to give them readable cgroup
names.

Thanks.

--
tejun

Subject: Re: [PATCH v3 06/16] memcg: infrastructure to match an allocation to the right cache

Posted by Tejun Heo on Fri, 21 Sep 2012 18:32:17 GMT

View Forum Message <> Reply to Message

On Tue, Sep 18, 2012 at 06:12:00PM +0400, Glauber Costa wrote:

```
> diff --git a/mm/memcontrol.c b/mm/memcontrol.c
> index 04851bb..1cce5c3 100644
> --- a/mm/memcontrol.c
> +++ b/mm/memcontrol.c
> @@ -339,6 +339,11 @@ 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
```

Bah, 400 entry array in struct mem_cgroup. Can't we do something a
bit more flexible?

```
> +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);
```

Maybe including full path is better, I don't know.

```
> + return name;
> +}
...
>  void __init memcg_init_kmem_cache(void)
> @@ -665,6 +704,170 @@ 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);
```

Blank line missing. Or if it's used inside memcg_create_kmem_cache()
only move it inside the function?

```
> +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_can_account_kmem(memcg));
```

WARN_ON_ONCE() generally preferred.

```
> + idx = cachep->memcg_params.id;
```

Ah, okay so the id is assigned to the "base" cache. Maybe explain it
somewhere?

```
> + 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)
> +{
...
> +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);
> +}
```

Why create your own worklist and flush mechanism? Just embed a work item in create_work and use a dedicated workqueue for flushing.

```
> +/*
> + * 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_can_account_kmem(memcg))
> +  return cachep;
> +
> + if (memcg->slabs[idx] == NULL) {
> +  memcg_create_cache_enqueue(memcg, cachep);
```

Do we want to wait for the work item if @gfp allows?

Thanks.

--
tejun

Subject: Re: [PATCH v3 07/16] memcg: skip memcg kmem allocations in specified code regions

Posted by Tejun Heo on Fri, 21 Sep 2012 19:59:29 GMT

View Forum Message <> Reply to Message

Hello,

On Tue, Sep 18, 2012 at 06:12:01PM +0400, Glauber Costa wrote:
> +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--;
> +}

I can't say I'm a big fan of this approach. If there are enough
users, maybe but can't we just annotate the affected allocations
explicitly? Is this gonna have many more users?

Also, in general, can we please add some comments? I know memcg code
is dearth of comments but let's please not keep it that way.

Thanks.

--
tejun

Subject: Re: [PATCH v3 09/16] sl[au]b: always get the cache from its page in kfree
Posted by Tejun Heo on Fri, 21 Sep 2012 20:07:33 GMT

View Forum Message <> Reply to Message

Hello,

On Fri, Sep 21, 2012 at 12:41:52PM +0300, Pekka Enberg wrote:
> > I am already using static keys extensively in this patchset, and that is
> > how I intend to handle this particular case.
>
> Cool.
>
> The key point here is that !CONFIG_MEMCG_KMEM should have exactly *zero*

> performance impact and CONFIG_MEMCG_KMEM disabled at runtime should have
> absolute minimal impact.

Not necessarily disagreeing, but I don't think it's helpful to set the
bar impossibly high. Even static_key doesn't have "exactly *zero*"
impact. Let's stick to as minimal as possible when not in use and
reasonable in use.

And, yeah, this one can be easily solved by using static_key.

Thanks.

--
tejun

Subject: Re: [PATCH v3 09/16] sl[au]b: always get the cache from its page in kfree
Posted by Pekka Enberg on Fri, 21 Sep 2012 20:14:45 GMT
View Forum Message <> Reply to Message

On Fri, Sep 21, 2012 at 11:07 PM, Tejun Heo <tj@kernel.org> wrote:
> Not necessarily disagreeing, but I don't think it's helpful to set the
> bar impossibly high. Even static_key doesn't have "exactly *zero*"
> impact. Let's stick to as minimal as possible when not in use and
> reasonable in use.

For !CONFIG_MEMCG_KMEM, it should be exactly zero. No need to play
games with static_key.

Subject: Re: [PATCH v3 09/16] sl[au]b: always get the cache from its page in kfree
Posted by Tejun Heo on Fri, 21 Sep 2012 20:16:02 GMT
View Forum Message <> Reply to Message

On Fri, Sep 21, 2012 at 11:14:45PM +0300, Pekka Enberg wrote:
> On Fri, Sep 21, 2012 at 11:07 PM, Tejun Heo <tj@kernel.org> wrote:
> > Not necessarily disagreeing, but I don't think it's helpful to set the
> > bar impossibly high. Even static_key doesn't have "exactly *zero*"
> > impact. Let's stick to as minimal as possible when not in use and
> > reasonable in use.
>
> For !CONFIG_MEMCG_KMEM, it should be exactly zero. No need to play
> games with static_key.

Ah, misread, though you were talking about MEMCG_KMEM && not in use.
Yeap, if turned off in Kconfig, it shouldn't have any performance
overhead.

Thanks.

--
tejun

Subject: Re: [PATCH v3 11/16] memcg: destroy memcg caches
Posted by Tejun Heo on Fri, 21 Sep 2012 20:22:20 GMT
View Forum Message <> Reply to Message

Hello, Glauber.

On Tue, Sep 18, 2012 at 06:12:05PM +0400, Glauber Costa wrote:

```
> +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);
```

Again, please don't build your own worklist. Just embed a work item
into mem_cgroup_cache_params and manipulate them. No need to
duplicate what workqueue already implements.

Thanks.

--

tejun

Subject: Re: [PATCH v3 12/16] memcg/sl[au]b Track all the memcg children of a kmem_cache.
Posted by Tejun Heo on Fri, 21 Sep 2012 20:31:01 GMT
View Forum Message <> Reply to Message

Hello, Glauber.

On Tue, Sep 18, 2012 at 06:12:06PM +0400, Glauber Costa wrote:
> 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.

I find the terms parent / child / sibling a bit confusing. It usually
implies proper tree structure. Maybe we can use better terms which
reflect the single layer structure better?

And, again, in general, please add some comments. If someone tries to
understand this for the first time and takes a look at
mem_cgroup_cache_params, there's almost nothing to guide that person.
What's the struct for? What does each field do? What are the
synchronization rules?

```
> @@ -626,6 +630,9 @@ 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);
> +
```

list_del_init() please.

Thanks.

--
tejun

Subject: Re: [PATCH v3 15/16] memcg/sl[au]b: shrink dead caches
Posted by Tejun Heo on Fri, 21 Sep 2012 20:40:35 GMT
View Forum Message <> Reply to Message

Hello, Glauber.

On Tue, Sep 18, 2012 at 06:12:09PM +0400, Glauber Costa wrote:

```
> @@ -764,10 +777,21 @@ 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';
```

This is kinda nasty. Do we really need to do this? How long would a
dead cache stick around?

```
> diff --git a/mm/slab.c b/mm/slab.c
> index bd9928f..6cb4abf 100644
> --- a/mm/slab.c
> +++ b/mm/slab.c
> @@ -3785,6 +3785,8 @@ static inline void __cache_free(struct kmem_cache *cachep, void
*objp,
>   }
>
>   ac_put_obj(cachep, ac, objp);
> +
> + kmem_cache_verify_dead(cachep);
```

Reaping dead caches doesn't exactly sound like a high priority thing
and adding a branch to hot path for that might not be the best way to
do it. Why not schedule an extremely lazy deferrable delayed_work
which polls for emptiness, say, every miniute or whatever?

Thanks.

--
tejun

Subject: Re: [PATCH v3 00/16] slab accounting for memcg
Posted by Tejun Heo on Fri, 21 Sep 2012 20:46:19 GMT

View Forum Message <> Reply to Message

Hello,

On Tue, Sep 18, 2012 at 06:11:54PM +0400, Glauber Costa wrote:
> This is a followup to the previous kmem series. I divided them logically
> so it gets easier for reviewers. But I believe they are ready to be merged
> together (although we can do a two-pass merge if people would prefer)
>
> Throwaway git tree found at:
>
>   git://git.kernel.org/pub/scm/linux/kernel/git/glommer/memcg. git kmemcg-slab
>
> There are mostly bugfixes since last submission.
>
> For a detailed explanation about this series, please refer to my previous post
> (Subj: [PATCH v3 00/13] kmem controller for memcg.)

In general, things look good to me. I think the basic approach is
manageable and does a decent job of avoiding introducing complications
on the usual code paths.

Pekka seems generally happy with the approach too. Christoph, what do
you think?

Thanks.

--
tejun

Subject: Re: [PATCH v3 00/16] slab accounting for memcg
Posted by Tejun Heo on Fri, 21 Sep 2012 20:47:27 GMT

View Forum Message <> Reply to Message

On Fri, Sep 21, 2012 at 01:46:19PM -0700, Tejun Heo wrote:
> In general, things look good to me. I think the basic approach is
> manageable and does a decent job of avoiding introducing complications
> on the usual code paths.
>
> Pekka seems generally happy with the approach too. Christoph, what do
> you think?

Ooh, also, why aren't Andrew and Michal not cc'd?

--
tejun

Subject: Re: [PATCH v3 06/16] memcg: infrastructure to match an allocation to the right cache

Posted by Tejun Heo on Fri, 21 Sep 2012 20:52:36 GMT
View Forum Message <> Reply to Message

Missed some stuff.

On Tue, Sep 18, 2012 at 06:12:00PM +0400, Glauber Costa wrote:

```
> +static struct kmem_cache *memcg_create_kmem_cache(struct mem_cgroup *memcg,
> +       struct kmem_cache *cachep)
> +{
...
> + memcg->slabs[idx] = new_cachep;
...
> +struct kmem_cache *__memcg_kmem_get_cache(struct kmem_cache *cachep,
> +      gfp_t gfp)
> +{
...
> + return memcg->slabs[idx];
```

I think you need memory barriers for the above pair.

Thanks.

--
tejun

Subject: Re: [PATCH v3 05/16] consider a memcg parameter in kmem_create_cache

Posted by Glauber Costa on Mon, 24 Sep 2012 08:12:23 GMT
View Forum Message <> Reply to Message

On 09/21/2012 10:14 PM, Tejun Heo wrote:

```
> Hello, Glauber.
>
> On Tue, Sep 18, 2012 at 06:11:59PM +0400, Glauber Costa wrote:
>> +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;
>> +}
>
> I'm a bit confused.  Why is id allocated only when memcg is NULL?
>
```

I think you figured that out already from your answer in another patch,
right? But I'll add a comment here since it seems to be a a natural
search point for people, explaining the mechanism.

> Also, how would the per-memcg slab/slubs appear in slabinfo? If they
> appear separately it might be better to give them readable cgroup
> names.
>

The new caches will appear under /proc/slabinfo with the rest, with a
string appended that identifies the group.

> Thanks.
>

Subject: Re: [PATCH v3 00/16] slab accounting for memcg
Posted by Glauber Costa on Mon, 24 Sep 2012 08:15:51 GMT

View Forum Message <> Reply to Message

On 09/22/2012 12:46 AM, Tejun Heo wrote:
> Hello,
>
> On Tue, Sep 18, 2012 at 06:11:54PM +0400, Glauber Costa wrote:
>> This is a followup to the previous kmem series. I divided them logically
>> so it gets easier for reviewers. But I believe they are ready to be merged
>> together (although we can do a two-pass merge if people would prefer)
>>
>> Throwaway git tree found at:
>>
>>   git://git.kernel.org/pub/scm/linux/kernel/git/glommer/memcg. git kmemcg-slab
>>
>> There are mostly bugfixes since last submission.
>>
>> For a detailed explanation about this series, please refer to my previous post
>> (Subj: [PATCH v3 00/13] kmem controller for memcg.)
>
> In general, things look good to me. I think the basic approach is
> manageable and does a decent job of avoiding introducing complications
> on the usual code paths.
>
> Pekka seems generally happy with the approach too. Christoph, what do
> you think?
>
> Thanks.
>
I myself think that while the approach is okay, this would need one or
two more versions for us to sort out some issues that are still remaining.

I'd very much like to have the kmemcg-stack series seeing its way
forward first. Mel Gorman said he would try his best to review it his
week, and I don't plan to resubmit anything before that.

After that, I plan to rebase at least this one again.

Subject: Re: [PATCH v3 06/16] memcg: infrastructure to match an allocation to the right cache
Posted by Glauber Costa on Mon, 24 Sep 2012 08:17:37 GMT
View Forum Message <> Reply to Message

On 09/22/2012 12:52 AM, Tejun Heo wrote:

```
> Missed some stuff.
>
> On Tue, Sep 18, 2012 at 06:12:00PM +0400, Glauber Costa wrote:
>> +static struct kmem_cache *memcg_create_kmem_cache(struct mem_cgroup *memcg,
>> +         struct kmem_cache *cachep)
>> +{
> ...
>> + memcg->slabs[idx] = new_cachep;
> ...
>> +struct kmem_cache *__memcg_kmem_get_cache(struct kmem_cache *cachep,
>> +        gfp_t gfp)
>> +{
> ...
>> + return memcg->slabs[idx];
>
> I think you need memory barriers for the above pair.
>
> Thanks.
>
```

Why is that?

We'll either see a value, or NULL. If we see NULL, we assume the cache
is not yet created. Not a big deal.

Subject: Re: [PATCH v3 15/16] memcg/sl[au]b: shrink dead caches
Posted by Glauber Costa on Mon, 24 Sep 2012 08:25:00 GMT
View Forum Message <> Reply to Message

On 09/22/2012 12:40 AM, Tejun Heo wrote:

```
> Hello, Glauber.
>
```

> On Tue, Sep 18, 2012 at 06:12:09PM +0400, Glauber Costa wrote:
>> @@ -764,10 +777,21 @@ 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';
>
> This is kinda nasty. Do we really need to do this? How long would a
> dead cache stick around?

Without targeted shrinking, until all objects are manually freed, which
may need to wait global reclaim to kick in.

In general, if we agree with duplicating the caches, the problem that
they may stick around for some time will not be avoidable. If you have
any suggestions about alternative ways for it, I'm all ears.

>
>> diff --git a/mm/slab.c b/mm/slab.c
>> index bd9928f..6cb4abf 100644
>> --- a/mm/slab.c
>> +++ b/mm/slab.c
>> @@ -3785,6 +3785,8 @@ static inline void __cache_free(struct kmem_cache *cachep, void *objp,
>>   }
>>
>>   ac_put_obj(cachep, ac, objp);
>> +
>> + kmem_cache_verify_dead(cachep);
>
> Reaping dead caches doesn't exactly sound like a high priority thing
> and adding a branch to hot path for that might not be the best way to
> do it. Why not schedule an extremely lazy deferrable delayed_work
> which polls for emptiness, say, every miniute or whatever?
>

Because this branch is marked as unlikely, I would expect it not to be a
big problem. It will be not taken most of the time, and becomes a very
cheap branch. I considered this to be simpler than a deferred work
mechanism.

If even then, you guys believe this is still too high, I can resort to that.

> Thanks.
>

Subject: Re: [PATCH v3 06/16] memcg: infrastructure to match an allocation to the right cache
Posted by Glauber Costa on Mon, 24 Sep 2012 08:46:35 GMT
View Forum Message <> Reply to Message

On 09/21/2012 10:32 PM, Tejun Heo wrote:

```
> On Tue, Sep 18, 2012 at 06:12:00PM +0400, Glauber Costa wrote:
>> diff --git a/mm/memcontrol.c b/mm/memcontrol.c
>> index 04851bb..1cce5c3 100644
>> --- a/mm/memcontrol.c
>> +++ b/mm/memcontrol.c
>> @@ -339,6 +339,11 @@ 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
>
> Bah, 400 entry array in struct mem_cgroup.  Can't we do something a
> bit more flexible?
>
```

I guess. I still would like it to be an array, so we can easily access
its fields. There are two ways around this:

1) Do like the events mechanism and allocate this in a separate
structure. Add a pointer chase in the access, and I don't think it helps
much because it gets allocated anyway. But we could at least
defer it to the time when we limit the cache.

2) The indexes are only assigned after the slab is FULL. At that time, a
lot of the caches are already initialized. We can, for instance, allow
for "twice the number we have in the system", which already provides
room for a couple of more appearing. Combining with the 1st approach, we
can defer it to limit-time, and then allow for, say, 50 % more caches
than we already have. The pointer chasing may very well be worth it...

```
>> +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);
>
```

> Maybe including full path is better, I don't know.

It can get way too big.

```
>> +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);
>> +}
>
```

> Why create your own worklist and flush mechanism? Just embed a work
> item in create_work and use a dedicated workqueue for flushing.

I'll take a look at this.

```
>
>> +/*
>> + * 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_can_account_kmem(memcg))
>> +  return cachep;
>> +
>> + if (memcg->slabs[idx] == NULL) {
>> +  memcg_create_cache_enqueue(memcg, cachep);
>
```

> Do we want to wait for the work item if @gfp allows?
>

I tried this once, and it got complicated enough that I deemed as "not
worth it". I honestly don't remember much of the details now, it was one
of the first things I tried, and a bunch of time has passed. If you
think it is absolutely worth it, I can try it again. But at the very
best, I view this as an optimization.

Subject: Re: [PATCH v3 07/16] memcg: skip memcg kmem allocations in specified code regions
Posted by Glauber Costa on Mon, 24 Sep 2012 09:09:23 GMT

View Forum Message <> Reply to Message

On 09/21/2012 11:59 PM, Tejun Heo wrote:

```
> Hello,
>
> On Tue, Sep 18, 2012 at 06:12:01PM +0400, Glauber Costa wrote:
>> +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--;
>> +}
>
> I can't say I'm a big fan of this approach.  If there are enough
> users, maybe but can't we just annotate the affected allocations
> explicitly?  Is this gonna have many more users?
>
```

What exactly do you mean by annotating the affected allocations?

There are currently two users of this. In both places, we are interested
in disallowing recursion, because cache creation will trigger new cache
allocations that will bring us back here.

We can't rely on unsetting the GFP flag we're using for this, because
that affects only the page allocation, not the metadata allocation for

the cache.

> Also, in general, can we please add some comments? I know memcg code
> is dearth of comments but let's please not keep it that way.
>
All right here.

Subject: Re: [PATCH v3 05/16] consider a memcg parameter in kmem_create_cache
Posted by christoph on Mon, 24 Sep 2012 12:41:26 GMT

View Forum Message <> Reply to Message

On Sep 24, 2012, at 3:12, Glauber Costa <glommer@parallels.com> wrote:

> On 09/21/2012 10:14 PM, Tejun Heo wrote:
>
> The new caches will appear under /proc/slabinfo with the rest, with a
> string appended that identifies the group.

There are f.e. meminfo files in the per node directories in sysfs. It would make sense to have a slabinfo file there (if you want to keep that around). Then the format would be the same.

Subject: Re: [PATCH v3 05/16] consider a memcg parameter in kmem_create_cache
Posted by Glauber Costa on Mon, 24 Sep 2012 12:41:36 GMT

View Forum Message <> Reply to Message

On 09/24/2012 04:41 PM, Christoph wrote:
>
> On Sep 24, 2012, at 3:12, Glauber Costa <glommer@parallels.com> wrote:
>
>> On 09/21/2012 10:14 PM, Tejun Heo wrote:
>>
>> The new caches will appear under /proc/slabinfo with the rest, with a
>> string appended that identifies the group.
>
> There are f.e. meminfo files in the per node directories in sysfs. It would make sense to have a slabinfo file there (if you want to keep that around). Then the format would be the same.
>

But that is orthogonal, isn't it? People will still expect to see it in
the old slabinfo file.

Subject: Re: [PATCH v3 05/16] consider a memcg parameter in kmem_create_cache

Posted by Christoph Lameter on Mon, 24 Sep 2012 13:42:05 GMT

View Forum Message <> Reply to Message

On Mon, 24 Sep 2012, Glauber Costa wrote:

> But that is orthogonal, isn't it? People will still expect to see it in
> the old slabinfo file.

The current scheme for memory statistics is

/proc/meminfo contains global counters

/sys/devices/system/node/nodeX/meminfo

contains node specific counters.

The cgroups directory already contains various files. Adding a slabinfo file would make sense and it could be found easily.

The sysfs hierachy /sys/kernel/slab could also show up there as a "slab" directory under which all the details of the various caches would be available and tunable. Very much in sync with the way the cgroups directories operate.

Subject: Re: [PATCH v3 05/16] consider a memcg parameter in kmem_create_cache

Posted by Glauber Costa on Mon, 24 Sep 2012 13:44:24 GMT

View Forum Message <> Reply to Message

On 09/24/2012 05:42 PM, Christoph Lameter wrote:
> On Mon, 24 Sep 2012, Glauber Costa wrote:
>
>> But that is orthogonal, isn't it? People will still expect to see it in
>> the old slabinfo file.
>
> The current scheme for memory statistics is
>
> /proc/meminfo contains global counters
>
> /sys/devices/system/node/nodeX/meminfo
>
> contains node specific counters.
>
> The cgroups directory already contains various files. Adding a slabinfo
> file would make sense and it could be found easily.

>
> The sysfs hierachy /sys/kernel/slab could also show up there as a "slab"
> directory under which all the details of the various caches would be
> available and tunable. Very much in sync with the way the cgroups
> directories operate.
>

The reason I say it is orthogonal, is that people will still want to see
their caches in /proc/slabinfo, regardless of wherever else they'll be.
It was a requirement from Pekka in one of the first times I posted this,
IIRC.

Subject: Re: [PATCH v3 05/16] consider a memcg parameter in kmem_create_cache
Posted by Christoph Lameter on Mon, 24 Sep 2012 13:56:32 GMT

View Forum Message <> Reply to Message

On Mon, 24 Sep 2012, Glauber Costa wrote:

> The reason I say it is orthogonal, is that people will still want to see
> their caches in /proc/slabinfo, regardless of wherever else they'll be.
> It was a requirement from Pekka in one of the first times I posted this,
> IIRC.

They want to see total counts there true. But as I said we already have a
duplication of the statistics otherwise. We have never done the scheme
that you propose. That is unexpected. I would not expect the numbers to be
there.

Subject: Re: [PATCH v3 05/16] consider a memcg parameter in kmem_create_cache
Posted by Glauber Costa on Mon, 24 Sep 2012 13:57:55 GMT

View Forum Message <> Reply to Message

On 09/24/2012 05:56 PM, Christoph Lameter wrote:
> On Mon, 24 Sep 2012, Glauber Costa wrote:
>
>> The reason I say it is orthogonal, is that people will still want to see
>> their caches in /proc/slabinfo, regardless of wherever else they'll be.
>> It was a requirement from Pekka in one of the first times I posted this,
>> IIRC.
>
> They want to see total counts there true. But as I said we already have a
> duplication of the statistics otherwise. We have never done the scheme
> that you propose. That is unexpected. I would not expect the numbers to be

> there.
>

I myself personally believe it can potentially clutter slabinfo, and
won't put my energy in defending the current implementation. What I
don't want is to keep switching between implementations.

Pekka, Tejun, what do you guys say here?

Subject: Re: [PATCH v3 05/16] consider a memcg parameter in kmem_create_cache
Posted by Glauber Costa on Mon, 24 Sep 2012 15:36:44 GMT
View Forum Message <> Reply to Message

On 09/24/2012 07:38 PM, Pekka Enberg wrote:
> On 09/24/2012 05:56 PM, Christoph Lameter wrote:
>>> On Mon, 24 Sep 2012, Glauber Costa wrote:
>>>
>>>> The reason I say it is orthogonal, is that people will still want to see
>>>> their caches in /proc/slabinfo, regardless of wherever else they'll be.
>>>> It was a requirement from Pekka in one of the first times I posted this,
>>>> IIRC.
>>>
>>> They want to see total counts there true. But as I said we already have a
>>> duplication of the statistics otherwise. We have never done the scheme
>>> that you propose. That is unexpected. I would not expect the numbers to be
>>> there.
>
> On Mon, Sep 24, 2012 at 4:57 PM, Glauber Costa <glommer@parallels.com> wrote:
>> I myself personally believe it can potentially clutter slabinfo, and
>> won't put my energy in defending the current implementation. What I
>> don't want is to keep switching between implementations.
>>
>> Pekka, Tejun, what do you guys say here?
>
> So Christoph is proposing that the new caches appear somewhere under
> the cgroups directory and /proc/slabinfo includes aggregated counts,
> right? I'm certainly OK with that.
>
Just for clarification, I am not sure about the aggregate counts -
although it surely makes sense.

Christoph, is that what you're proposing ?

Subject: Re: [PATCH v3 05/16] consider a memcg parameter in

kmem_create_cache
Posted by Pekka Enberg on Mon, 24 Sep 2012 15:38:48 GMT
View Forum Message <> Reply to Message

On 09/24/2012 05:56 PM, Christoph Lameter wrote:
>> On Mon, 24 Sep 2012, Glauber Costa wrote:
>>
>>> The reason I say it is orthogonal, is that people will still want to see
>>> their caches in /proc/slabinfo, regardless of wherever else they'll be.
>>> It was a requirement from Pekka in one of the first times I posted this,
>>> IIRC.
>>
>> They want to see total counts there true. But as I said we already have a
>> duplication of the statistics otherwise. We have never done the scheme
>> that you propose. That is unexpected. I would not expect the numbers to be
>> there.

On Mon, Sep 24, 2012 at 4:57 PM, Glauber Costa <glommer@parallels.com> wrote:
> I myself personally believe it can potentially clutter slabinfo, and
> won't put my energy in defending the current implementation. What I
> don't want is to keep switching between implementations.
>
> Pekka, Tejun, what do you guys say here?

So Christoph is proposing that the new caches appear somewhere under
the cgroups directory and /proc/slabinfo includes aggregated counts,
right? I'm certainly OK with that.

Subject: Re: [PATCH v3 05/16] consider a memcg parameter in kmem_create_cache
Posted by Christoph Lameter on Mon, 24 Sep 2012 17:38:17 GMT
View Forum Message <> Reply to Message

On Mon, 24 Sep 2012, Glauber Costa wrote:

> > So Christoph is proposing that the new caches appear somewhere under
> > the cgroups directory and /proc/slabinfo includes aggregated counts,
> > right? I'm certainly OK with that.
> >
> Just for clarification, I am not sure about the aggregate counts -
> although it surely makes sense.
>
> Christoph, is that what you're proposing ?

Yes. Make it similar to the way /proc/meminfo is handled.

Subject: Re: [PATCH v3 05/16] consider a memcg parameter in kmem_create_cache
Posted by Christoph Lameter on Mon, 24 Sep 2012 17:39:49 GMT

View Forum Message <> Reply to Message

On Mon, 24 Sep 2012, Pekka Enberg wrote:

> So Christoph is proposing that the new caches appear somewhere under
> the cgroups directory and /proc/slabinfo includes aggregated counts,
> right? I'm certainly OK with that.

Caches would appear either in cgroup/slabinfo (which would have the same
format as /proc/slabinfo) or in cgroup/slab/<slabname>/<fieldname>
(similar to /sys/kernel/slab/...)

Subject: Re: [PATCH v3 15/16] memcg/sl[au]b: shrink dead caches
Posted by Tejun Heo on Mon, 24 Sep 2012 17:43:46 GMT

View Forum Message <> Reply to Message

Hello,

On Mon, Sep 24, 2012 at 12:25:00PM +0400, Glauber Costa wrote:
> > This is kinda nasty. Do we really need to do this? How long would a
> > dead cache stick around?
>
> Without targeted shrinking, until all objects are manually freed, which
> may need to wait global reclaim to kick in.
>
> In general, if we agree with duplicating the caches, the problem that
> they may stick around for some time will not be avoidable. If you have
> any suggestions about alternative ways for it, I'm all ears.

I don't have much problem with caches sticking around waiting to be
reaped. I'm just wondering whether renaming trick is really
necessary.

> > Reaping dead caches doesn't exactly sound like a high priority thing
> > and adding a branch to hot path for that might not be the best way to
> > do it. Why not schedule an extremely lazy deferrable delayed_work
> > which polls for emptiness, say, every miniute or whatever?
> >
>
> Because this branch is marked as unlikely, I would expect it not to be a
> big problem. It will be not taken most of the time, and becomes a very
> cheap branch. I considered this to be simpler than a deferred work
> mechanism.
>

> If even then, you guys believe this is still too high, I can resort to that.

It's still an otherwise unnecessary branch on a very hot path. If you
can remove it, there's no reason not to.

Thanks.

--
tejun

Subject: Re: [PATCH v3 07/16] memcg: skip memcg kmem allocations in specified code regions
Posted by Tejun Heo on Mon, 24 Sep 2012 17:47:42 GMT

View Forum Message <> Reply to Message

Hello,

On Mon, Sep 24, 2012 at 01:09:23PM +0400, Glauber Costa wrote:
> > I can't say I'm a big fan of this approach. If there are enough
> > users, maybe but can't we just annotate the affected allocations
> > explicitly? Is this gonna have many more users?
>
> What exactly do you mean by annotating the affected allocations?

IOW, can't you just pass down an extra argument / flag / whatever
instead?

Thanks.

--
tejun

Subject: Re: [PATCH v3 06/16] memcg: infrastructure to match an allocation to the right cache
Posted by Tejun Heo on Mon, 24 Sep 2012 17:56:19 GMT

View Forum Message <> Reply to Message

Hello, Glauber.

On Mon, Sep 24, 2012 at 12:46:35PM +0400, Glauber Costa wrote:
> >> +#ifdef CONFIG_MEMCG_KMEM
> >> + /* Slab accounting */
> >> + struct kmem_cache *slabs[MAX_KMEM_CACHE_TYPES];
> >> +#endif
> >

> > Bah, 400 entry array in struct mem_cgroup. Can't we do something a
> > bit more flexible?
> >
>
> I guess. I still would like it to be an array, so we can easily access
> its fields. There are two ways around this:
>
> 1) Do like the events mechanism and allocate this in a separate
> structure. Add a pointer chase in the access, and I don't think it helps
> much because it gets allocated anyway. But we could at least
> defer it to the time when we limit the cache.

Start at some reasonable size and then double it as usage grows? How
many kmem_caches do we typically end up using?

> >> + if (memcg->slabs[idx] == NULL) {
> >> + memcg_create_cache_enqueue(memcg, cachep);
> >
> > Do we want to wait for the work item if @gfp allows?
> >
>
> I tried this once, and it got complicated enough that I deemed as "not
> worth it". I honestly don't remember much of the details now, it was one
> of the first things I tried, and a bunch of time has passed. If you
> think it is absolutely worth it, I can try it again. But at the very
> best, I view this as an optimization.

I don't know. It seems like a logical thing to try and depends on how
complex it gets. I don't think it's a must. The whole thing is
somewhat opportunistic after all.

Thanks.

--
tejun

Subject: Re: [PATCH v3 06/16] memcg: infrastructure to match an allocation to the right cache

Posted by Tejun Heo on Mon, 24 Sep 2012 17:58:01 GMT

View Forum Message <> Reply to Message

On Mon, Sep 24, 2012 at 12:17:37PM +0400, Glauber Costa wrote:
> On 09/22/2012 12:52 AM, Tejun Heo wrote:
> > Missed some stuff.
> >
> > On Tue, Sep 18, 2012 at 06:12:00PM +0400, Glauber Costa wrote:
> >> +static struct kmem_cache *memcg_create_kmem_cache(struct mem_cgroup *memcg,

> >> +   struct kmem_cache *cachep)
> >> +{
> > ...
> >> + memcg->slabs[idx] = new_cachep;
> > ...
> >> +struct kmem_cache *__memcg_kmem_get_cache(struct kmem_cache *cachep,
> >> +   gfp_t gfp)
> >> +{
> > ...
> >> + return memcg->slabs[idx];
> >
> > I think you need memory barriers for the above pair.
> >
> > Thanks.
> >
>
> Why is that?
>
> We'll either see a value, or NULL. If we see NULL, we assume the cache
> is not yet created. Not a big deal.

Because when you see !NULL cache pointer you want to be able to see
the cache fully initialized. You need wmb between cache creation and
pointer assignment and at least read_barrier_depends() between
fetching the cache pointer and dereferencing it.

Thanks.

--
tejun

Subject: Re: [PATCH v3 06/16] memcg: infrastructure to match an allocation to the right cache
Posted by Glauber Costa on Tue, 25 Sep 2012 13:57:38 GMT

View Forum Message <> Reply to Message

On 09/24/2012 09:56 PM, Tejun Heo wrote:
> Hello, Glauber.
>
> On Mon, Sep 24, 2012 at 12:46:35PM +0400, Glauber Costa wrote:
>>>> +#ifdef CONFIG_MEMCG_KMEM
>>>> + /* Slab accounting */
>>>> + struct kmem_cache *slabs[MAX_KMEM_CACHE_TYPES];
>>>> +#endif
>>>
>>> Bah, 400 entry array in struct mem_cgroup. Can't we do something a
>>> bit more flexible?

>>>
>>
>> I guess. I still would like it to be an array, so we can easily access
>> its fields. There are two ways around this:
>>
>> 1) Do like the events mechanism and allocate this in a separate
>> structure. Add a pointer chase in the access, and I don't think it helps
>> much because it gets allocated anyway. But we could at least
>> defer it to the time when we limit the cache.
>
> Start at some reasonable size and then double it as usage grows? How
> many kmem_caches do we typically end up using?
>

So my Fedora box here, recently booted on a Fedora kernel, will have 111
caches. How would 150 sound to you?

Subject: Re: [PATCH v3 06/16] memcg: infrastructure to match an allocation to the right cache
Posted by Christoph Lameter on Tue, 25 Sep 2012 16:28:11 GMT

View Forum Message <> Reply to Message

On Tue, 25 Sep 2012, Glauber Costa wrote:

> >> 1) Do like the events mechanism and allocate this in a separate
> >> structure. Add a pointer chase in the access, and I don't think it helps
> >> much because it gets allocated anyway. But we could at least
> >> defer it to the time when we limit the cache.
> >
> > Start at some reasonable size and then double it as usage grows? How
> > many kmem_caches do we typically end up using?
> >
>
> So my Fedora box here, recently booted on a Fedora kernel, will have 111
> caches. How would 150 sound to you?

Some drivers/subsystems can dynamically create slabs as needed for new
devices or instances of metadata. You cannot use a fixed size
array and cannot establish an upper boundary for the number of slabs on
the system.

Subject: Re: [PATCH v3 05/16] consider a memcg parameter in kmem_create_cache
Posted by Michal Hocko on Tue, 02 Oct 2012 14:46:10 GMT

View Forum Message <> Reply to Message

On Tue 18-09-12 18:11:59, Glauber Costa wrote:
> 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.

It would be nice to describe what is memcg_params.id intended for. There
is no usage in this patch (except for create_unique_id in slub).
I guess that by root caches you mean all default caches with
memcg==NULL, right?

[...]
--
Michal Hocko
SUSE Labs