Subject: [PATCH v4 00/19] slab accounting for memcg
Posted by Glauber Costa on Fri, 12 Oct 2012 13:40:54 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

I've bundled the following important changes since last submission:
* no more messing with the cache name after destruction: aggregated figures
are shown in /proc/slabinfo.
* memory.kmem.slabinfo file with memcg-specific cache information during its
lifespan.
* full slub attribute propagation.
* reusing the standard workqueue mechanism.
* cache-side indexing, instead of memcg-side indexing. The memcg css_id serves
as an index, and we don't need extra indexes for that.
* struct memcg_cache_params no longer bundled in struct kmem_cache: We now will
have only a pointer in the struct, allowing memory consumption when disable to
fall down ever further.

Patches need to be adjusted to cope with those changes, but other than that,
look the same - just a lot simpler.

I also put quite some effort to overcome my writing disability and get some
decent changelogs in place.

For a detailed explanation about this whole effort, please refer to my previous
post (https://lkml.org/lkml/2012/10/8/119)

*** BLURB HERE ***

Glauber Costa (19):
slab: Ignore internal flags in cache creation
move slabinfo processing to slab_common.c
move print_slabinfo_header to slab_common.c
sl[au]b: process slabinfo_show in common code
slab: don't preemptively remove element from list in cache destroy
slab/slub: struct memcg_params
consider a memcg parameter in kmem_create_cache
Allocate memory for memcg caches whenever a new memcg appears
memcg: infrastructure to match an allocation to the right cache
memcg: skip memcg kmem allocations in specified code regions

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.
memcg/sl[au]b: shrink dead caches
Aggregate memcg cache values in slabinfo
slab: propagate tunables values
slub: slub-specific propagation changes.
Add slab-specific documentation about the kmem controller

```
Documentation/cgroups/memory.txt |   7 +
include/linux/memcontrol.h       |  88 ++++++
include/linux/sched.h            |   1 +
include/linux/slab.h             |  47 +++
include/linux/slab_def.h         |   3 +
include/linux/slub_def.h         |  19 +-
init/Kconfig                     |   2 +-
mm/memcontrol.c                  | 599 ++++++++++++++++++++++++++++++++++++++++++--
mm/slab.c                        | 210 ++++++--------
mm/slab.h                        | 157 +++++++++-
mm/slab_common.c                 | 224 +++++++++++++++-
mm/slub.c                        | 193 ++++++++-----
12 files changed, 1311 insertions(+), 239 deletions(-)
```

--
1.7.11.4

Subject: [PATCH v4 01/19] slab: Ignore internal flags in cache creation
Posted by Glauber Costa on Fri, 12 Oct 2012 13:40:55 GMT
View Forum Message <> Reply to Message

Some flags are used internally by the allocators for management
purposes. One example of that is the CFLGS_OFF_SLAB flag that slab uses
to mark that the metadata for that cache is stored outside of the slab.

No cache should ever pass those as a creation flags. We can just ignore
this bit if it happens to be passed (such as when duplicating a cache in
the kmem memcg patches).

Because such flags can vary from allocator to allocator, we allow them
to make their own decisions on that, defining SLAB_AVAILABLE_FLAGS with
all flags that are valid at creation time. Allocators that doesn't have
any specific flag requirement should define that to mean all flags.

Common code will mask out all flags not belonging to that set.

[v2: leave the mask out decision up to the allocators]

[v3: define flags for all allocators]
[v4: move all definitions to slab.h]

Signed-off-by: Glauber Costa <glommer@parallels.com>
CC: Christoph Lameter <cl@linux.com>
CC: David Rientjes <rientjes@google.com>
CC: Pekka Enberg <penberg@cs.helsinki.fi>
---
 mm/slab.c | 22 ----------------------
 mm/slab.h | 25 +++++++++++++++++++++++++
 mm/slab_common.c | 7 +++++++
 mm/slub.c | 3 ---
 4 files changed, 32 insertions(+), 25 deletions(-)

diff --git a/mm/slab.c b/mm/slab.c
index 87569af..eafef58 100644
--- a/mm/slab.c
+++ b/mm/slab.c
@@ -162,23 +162,6 @@
  */
 static bool pfmemalloc_active __read_mostly;

-/* Legal flag mask for kmem_cache_create(). */
-#if DEBUG
-# define CREATE_MASK (SLAB_RED_ZONE | \
-    SLAB_POISON | SLAB_HWCACHE_ALIGN | \
-    SLAB_CACHE_DMA | \
-    SLAB_STORE_USER | \
-    SLAB_RECLAIM_ACCOUNT | SLAB_PANIC | \
-    SLAB_DESTROY_BY_RCU | SLAB_MEM_SPREAD | \
-    SLAB_DEBUG_OBJECTS | SLAB_NOLEAKTRACE | SLAB_NOTRACK)
-#else
-# define CREATE_MASK (SLAB_HWCACHE_ALIGN | \
-    SLAB_CACHE_DMA | \
-    SLAB_RECLAIM_ACCOUNT | SLAB_PANIC | \
-    SLAB_DESTROY_BY_RCU | SLAB_MEM_SPREAD | \
-    SLAB_DEBUG_OBJECTS | SLAB_NOLEAKTRACE | SLAB_NOTRACK)
-#endif
-
 /*
  * kmem_bufctl_t:
  *
@@ -2385,11 +2368,6 @@ __kmem_cache_create (struct kmem_cache *cachep, unsigned long flags)
  if (flags & SLAB_DESTROY_BY_RCU)
   BUG_ON(flags & SLAB_POISON);
 #endif
- /*

```
-  * Always checks flags, a caller might be expecting debug support which
-  * isn't available.
-  */
- BUG_ON(flags & ~CREATE_MASK);

  /*
   * Check that size is in terms of words.  This is needed to avoid
diff --git a/mm/slab.h b/mm/slab.h
index 7deeb44..35b60b7 100644
--- a/mm/slab.h
+++ b/mm/slab.h
@@ -45,6 +45,31 @@ static inline struct kmem_cache *__kmem_cache_alias(const char *name,
size_t siz
 #endif


+/* Legal flag mask for kmem_cache_create(), for various configurations */
+#define SLAB_CORE_FLAGS (SLAB_HWCACHE_ALIGN | SLAB_CACHE_DMA |
SLAB_PANIC | \
+    SLAB_DESTROY_BY_RCU | SLAB_DEBUG_OBJECTS )
+
+#if defined(CONFIG_DEBUG_SLAB)
+#define SLAB_DEBUG_FLAGS (SLAB_RED_ZONE | SLAB_POISON | SLAB_STORE_USER)
+#elif defined(CONFIG_SLUB_DEBUG)
+#define SLAB_DEBUG_FLAGS (SLAB_RED_ZONE | SLAB_POISON | SLAB_STORE_USER | \
+     SLAB_TRACE | SLAB_DEBUG_FREE)
+#else
+#define SLAB_DEBUG_FLAGS (0)
+#endif
+
+#if defined(CONFIG_SLAB)
+#define SLAB_CACHE_FLAGS (SLAB_MEM_SPREAD | SLAB_NOLEAKTRACE | \
+     SLAB_RECLAIM_ACCOUNT | SLAB_TEMPORARY | SLAB_NOTRACK)
+#elif defined(CONFIG_SLUB)
+#define SLAB_CACHE_FLAGS (SLAB_NOLEAKTRACE | SLAB_RECLAIM_ACCOUNT | \
+     SLAB_TEMPORARY | SLAB_NOTRACK)
+#else
+#define SLAB_CACHE_FLAGS (0)
+#endif
+
+#define CACHE_CREATE_MASK (SLAB_CORE_FLAGS | SLAB_DEBUG_FLAGS |
SLAB_CACHE_FLAGS)
+
 int __kmem_cache_shutdown(struct kmem_cache *);

 #endif
diff --git a/mm/slab_common.c b/mm/slab_common.c
index 9c21725..0e2b8e3 100644
```

```
--- a/mm/slab_common.c
+++ b/mm/slab_common.c
@@ -107,6 +107,13 @@ struct kmem_cache *kmem_cache_create(const char *name, size_t
size, size_t align
  if (!kmem_cache_sanity_check(name, size) == 0)
   goto out_locked;

+ /*
+  * Some allocators will constraint the set of valid flags to a subset
+  * of all flags. We expect them to define CACHE_CREATE_MASK in this
+  * case, and we'll just provide them with a sanitized version of the
+  * passed flags.
+  */
+ flags &= CACHE_CREATE_MASK;

  s = __kmem_cache_alias(name, size, align, flags, ctor);
  if (s)
diff --git a/mm/slub.c b/mm/slub.c
index 628a261..f50c5b2 100644
--- a/mm/slub.c
+++ b/mm/slub.c
@@ -112,9 +112,6 @@
  *    the fast path and disables lockless freelists.
  */

-#define SLAB_DEBUG_FLAGS (SLAB_RED_ZONE | SLAB_POISON | SLAB_STORE_USER | \
-  SLAB_TRACE | SLAB_DEBUG_FREE)
-
 static inline int kmem_cache_debug(struct kmem_cache *s)
 {
 #ifdef CONFIG_SLUB_DEBUG
--
1.7.11.4
```

Subject: [PATCH v4 02/19] move slabinfo processing to slab_common.c
Posted by Glauber Costa on Fri, 12 Oct 2012 13:40:56 GMT
View Forum Message <> Reply to Message

This patch moves all the common machinery to slabinfo processing
to slab_common.c. We can do better by noticing that the output is
heavily common, and having the allocators to just provide finished
information about this. But after this first step, this can be done
easier.

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

```
CC: David Rientjes <rientjes@google.com>
---
 mm/slab.c        | 72 ++++++++++-----------------------------------------
 mm/slab.h        |  8 +++++++
 mm/slab_common.c | 70 ++++++++++++++++++++++++++++++++++++++++++++++++++++++
 mm/slub.c        | 51 ++++----------------------------------
 4 files changed, 96 insertions(+), 105 deletions(-)

diff --git a/mm/slab.c b/mm/slab.c
index eafef58..e35970a 100644
--- a/mm/slab.c
+++ b/mm/slab.c
@@ -4263,7 +4263,7 @@ out:

 #ifdef CONFIG_SLABINFO

-static void print_slabinfo_header(struct seq_file *m)
+void print_slabinfo_header(struct seq_file *m)
 {
  /*
   * Output format version, so at least we can change it
@@ -4286,28 +4286,7 @@ static void print_slabinfo_header(struct seq_file *m)
  seq_putc(m, '\n');
 }

-static void *s_start(struct seq_file *m, loff_t *pos)
-{
- loff_t n = *pos;
-
- mutex_lock(&slab_mutex);
- if (!n)
-  print_slabinfo_header(m);
-
- return seq_list_start(&slab_caches, *pos);
-}
-
-static void *s_next(struct seq_file *m, void *p, loff_t *pos)
-{
- return seq_list_next(p, &slab_caches, pos);
-}
-
-static void s_stop(struct seq_file *m, void *p)
-{
- mutex_unlock(&slab_mutex);
-}
-
-static int s_show(struct seq_file *m, void *p)
+int slabinfo_show(struct seq_file *m, void *p)
```

```
 {
  struct kmem_cache *cachep = list_entry(p, struct kmem_cache, list);
  struct slab *slabp;
@@ -4404,27 +4383,6 @@ static int s_show(struct seq_file *m, void *p)
  return 0;
 }

-/*
- * slabinfo_op - iterator that generates /proc/slabinfo
- *
- * Output layout:
- * cache-name
- * num-active-objs
- * total-objs
- * object size
- * num-active-slabs
- * total-slabs
- * num-pages-per-slab
- * + further values on SMP and with statistics enabled
- */
-
-static const struct seq_operations slabinfo_op = {
- .start = s_start,
- .next = s_next,
- .stop = s_stop,
- .show = s_show,
-};
-
 #define MAX_SLABINFO_WRITE 128
 /**
  * slabinfo_write - Tuning for the slab allocator
@@ -4433,7 +4391,7 @@ static const struct seq_operations slabinfo_op = {
  * @count: data length
  * @ppos: unused
  */
-static ssize_t slabinfo_write(struct file *file, const char __user *buffer,
+ssize_t slabinfo_write(struct file *file, const char __user *buffer,
         size_t count, loff_t *ppos)
 {
  char kbuf[MAX_SLABINFO_WRITE + 1], *tmp;
@@ -4476,19 +4434,6 @@ static ssize_t slabinfo_write(struct file *file, const char __user *buffer,
  return res;
 }

-static int slabinfo_open(struct inode *inode, struct file *file)
-{
- return seq_open(file, &slabinfo_op);
-}
```

```
-
-static const struct file_operations proc_slabinfo_operations = {
- .open  = slabinfo_open,
- .read  = seq_read,
- .write  = slabinfo_write,
- .llseek  = seq_lseek,
- .release = seq_release,
-};
-
 #ifdef CONFIG_DEBUG_SLAB_LEAK

 static void *leaks_start(struct seq_file *m, loff_t *pos)
@@ -4617,6 +4562,16 @@ static int leaks_show(struct seq_file *m, void *p)
  return 0;
 }

+static void *s_next(struct seq_file *m, void *p, loff_t *pos)
+{
+ return seq_list_next(p, &slab_caches, pos);
+}
+
+static void s_stop(struct seq_file *m, void *p)
+{
+ mutex_unlock(&slab_mutex);
+}
+
 static const struct seq_operations slabstats_op = {
  .start = leaks_start,
  .next = s_next,
@@ -4651,7 +4606,6 @@ static const struct file_operations proc_slabstats_operations = {

 static int __init slab_proc_init(void)
 {
- proc_create("slabinfo",S_IWUSR|S_IRUSR,NULL,&proc_slabinfo_operations);
 #ifdef CONFIG_DEBUG_SLAB_LEAK
  proc_create("slab_allocators", 0, NULL, &proc_slabstats_operations);
 #endif
diff --git a/mm/slab.h b/mm/slab.h
index 35b60b7..4156d21 100644
--- a/mm/slab.h
+++ b/mm/slab.h
@@ -72,4 +72,12 @@ static inline struct kmem_cache *__kmem_cache_alias(const char *name,
size_t siz

 int __kmem_cache_shutdown(struct kmem_cache *);

+struct seq_file;
+struct file;
```

```
+void print_slabinfo_header(struct seq_file *m);
+
+int slabinfo_show(struct seq_file *m, void *p);
+
+ssize_t slabinfo_write(struct file *file, const char __user *buffer,
+         size_t count, loff_t *ppos);
 #endif
diff --git a/mm/slab_common.c b/mm/slab_common.c
index 0e2b8e3..11ecab4 100644
--- a/mm/slab_common.c
+++ b/mm/slab_common.c
@@ -13,6 +13,8 @@
 #include <linux/module.h>
 #include <linux/cpu.h>
 #include <linux/uaccess.h>
+#include <linux/seq_file.h>
+#include <linux/proc_fs.h>
 #include <asm/cacheflush.h>
 #include <asm/tlbflush.h>
 #include <asm/page.h>
@@ -196,3 +198,71 @@ int slab_is_available(void)
 {
  return slab_state >= UP;
 }
+
+#ifdef CONFIG_SLABINFO
+static void *s_start(struct seq_file *m, loff_t *pos)
+{
+ loff_t n = *pos;
+
+ mutex_lock(&slab_mutex);
+ if (!n)
+  print_slabinfo_header(m);
+
+ return seq_list_start(&slab_caches, *pos);
+}
+
+static void *s_next(struct seq_file *m, void *p, loff_t *pos)
+{
+ return seq_list_next(p, &slab_caches, pos);
+}
+
+static void s_stop(struct seq_file *m, void *p)
+{
+ mutex_unlock(&slab_mutex);
+}
+
+static int s_show(struct seq_file *m, void *p)
```

```
+{
+ return slabinfo_show(m, p);
+}
+
+/*
+ * slabinfo_op - iterator that generates /proc/slabinfo
+ *
+ * Output layout:
+ * cache-name
+ * num-active-objs
+ * total-objs
+ * object size
+ * num-active-slabs
+ * total-slabs
+ * num-pages-per-slab
+ * + further values on SMP and with statistics enabled
+ */
+static const struct seq_operations slabinfo_op = {
+ .start = s_start,
+ .next = s_next,
+ .stop = s_stop,
+ .show = s_show,
+};
+
+static int slabinfo_open(struct inode *inode, struct file *file)
+{
+ return seq_open(file, &slabinfo_op);
+}
+
+static const struct file_operations proc_slabinfo_operations = {
+ .open  = slabinfo_open,
+ .read  = seq_read,
+ .write         = slabinfo_write,
+ .llseek  = seq_lseek,
+ .release = seq_release,
+};
+
+static int __init slab_proc_init(void)
+{
+ proc_create("slabinfo", S_IRUSR, NULL, &proc_slabinfo_operations);
+ return 0;
+}
+module_init(slab_proc_init);
+#endif /* CONFIG_SLABINFO */
diff --git a/mm/slub.c b/mm/slub.c
index f50c5b2..55304ed 100644
--- a/mm/slub.c
+++ b/mm/slub.c
```

```
@@ -5394,7 +5394,7 @@ __initcall(slab_sysfs_init);
  * The /proc/slabinfo ABI
  */
 #ifdef CONFIG_SLABINFO
-static void print_slabinfo_header(struct seq_file *m)
+void print_slabinfo_header(struct seq_file *m)
 {
  seq_puts(m, "slabinfo - version: 2.1\n");
  seq_puts(m, "# name            <active_objs> <num_objs> <object_size> "
@@ -5404,28 +5404,7 @@ static void print_slabinfo_header(struct seq_file *m)
  seq_putc(m, '\n');
 }

-static void *s_start(struct seq_file *m, loff_t *pos)
-{
- loff_t n = *pos;
-
- mutex_lock(&slab_mutex);
- if (!n)
-  print_slabinfo_header(m);
-
- return seq_list_start(&slab_caches, *pos);
-}
-
-static void *s_next(struct seq_file *m, void *p, loff_t *pos)
-{
- return seq_list_next(p, &slab_caches, pos);
-}
-
-static void s_stop(struct seq_file *m, void *p)
-{
- mutex_unlock(&slab_mutex);
-}
-
-static int s_show(struct seq_file *m, void *p)
+int slabinfo_show(struct seq_file *m, void *p)
 {
  unsigned long nr_partials = 0;
  unsigned long nr_slabs = 0;
@@ -5461,29 +5440,9 @@ static int s_show(struct seq_file *m, void *p)
  return 0;
 }

-static const struct seq_operations slabinfo_op = {
- .start = s_start,
- .next = s_next,
- .stop = s_stop,
- .show = s_show,
```

```
-};
-
-static int slabinfo_open(struct inode *inode, struct file *file)
-{
- return seq_open(file, &slabinfo_op);
-}
-
-static const struct file_operations proc_slabinfo_operations = {
- .open  = slabinfo_open,
- .read  = seq_read,
- .llseek  = seq_lseek,
- .release = seq_release,
-};
-
-static int __init slab_proc_init(void)
+ssize_t slabinfo_write(struct file *file, const char __user *buffer,
+         size_t count, loff_t *ppos)
 {
- proc_create("slabinfo", S_IRUSR, NULL, &proc_slabinfo_operations);
- return 0;
+ return -EIO;
 }
-module_init(slab_proc_init);
 #endif /* CONFIG_SLABINFO */
--
1.7.11.4
```

Subject: [PATCH v4 03/19] move print_slabinfo_header to slab_common.c
Posted by Glauber Costa on Fri, 12 Oct 2012 13:40:57 GMT
View Forum Message <> Reply to Message

The header format is highly similar between slab and slub. The main
difference lays in the fact that slab may optionally have statistics
added here in case of CONFIG_SLAB_DEBUG, while the slub will stick them
somewhere else.

By making sure that information conditionally lives inside a
globally-visible CONFIG_DEBUG_SLAB switch, we can move the header
printing to a common location.

Signed-off-by: Glauber Costa <glommer@parallels.com>
Acked-by: Christoph Lameter <cl@linux.com>
CC: Pekka Enberg <penberg@cs.helsinki.fi>
CC: David Rientjes <rientjes@google.com>
---
 mm/slab.c | 24 ------------------------
 mm/slab.h | 2 --

```
 mm/slab_common.c | 23 ++++++++++++++++++++++
 mm/slub.c        | 10 ----------
 4 files changed, 23 insertions(+), 36 deletions(-)

diff --git a/mm/slab.c b/mm/slab.c
index e35970a..864a9e9 100644
--- a/mm/slab.c
+++ b/mm/slab.c
@@ -4262,30 +4262,6 @@ out:
 }

 #ifdef CONFIG_SLABINFO
-
-void print_slabinfo_header(struct seq_file *m)
-{
- /*
-  * Output format version, so at least we can change it
-  * without _too_ many complaints.
-  */
-#if STATS
- seq_puts(m, "slabinfo - version: 2.1 (statistics)\n");
-#else
- seq_puts(m, "slabinfo - version: 2.1\n");
-#endif
- seq_puts(m, "# name            <active_objs> <num_objs> <objsize> "
-   "<objperslab> <pagesperslab>");
- seq_puts(m, " : tunables <limit> <batchcount> <sharedfactor>");
- seq_puts(m, " : slabdata <active_slabs> <num_slabs> <sharedavail>");
-#if STATS
- seq_puts(m, " : globalstat <listallocs> <maxobjs> <grown> <reaped> "
-   "<error> <maxfreeable> <nodeallocs> <remotefrees> <alienoverflow>");
- seq_puts(m, " : cpustat <allochit> <allocmiss> <freehit> <freemiss>");
-#endif
- seq_putc(m, '\n');
-}
-
 int slabinfo_show(struct seq_file *m, void *p)
 {
  struct kmem_cache *cachep = list_entry(p, struct kmem_cache, list);
diff --git a/mm/slab.h b/mm/slab.h
index 4156d21..e9ba23f 100644
--- a/mm/slab.h
+++ b/mm/slab.h
@@ -74,8 +74,6 @@ int __kmem_cache_shutdown(struct kmem_cache *);

 struct seq_file;
 struct file;
-void print_slabinfo_header(struct seq_file *m);
```

```
-
 int slabinfo_show(struct seq_file *m, void *p);

 ssize_t slabinfo_write(struct file *file, const char __user *buffer,
diff --git a/mm/slab_common.c b/mm/slab_common.c
index 11ecab4..bb4d751 100644
--- a/mm/slab_common.c
+++ b/mm/slab_common.c
@@ -200,6 +200,29 @@ int slab_is_available(void)
 }

 #ifdef CONFIG_SLABINFO
+static void print_slabinfo_header(struct seq_file *m)
+{
+ /*
+  * Output format version, so at least we can change it
+  * without _too_ many complaints.
+  */
+#ifdef CONFIG_DEBUG_SLAB
+ seq_puts(m, "slabinfo - version: 2.1 (statistics)\n");
+#else
+ seq_puts(m, "slabinfo - version: 2.1\n");
+#endif
+ seq_puts(m, "# name            <active_objs> <num_objs> <objsize> "
+   "<objperslab> <pagesperslab>");
+ seq_puts(m, " : tunables <limit> <batchcount> <sharedfactor>");
+ seq_puts(m, " : slabdata <active_slabs> <num_slabs> <sharedavail>");
+#ifdef CONFIG_DEBUG_SLAB
+ seq_puts(m, " : globalstat <listallocs> <maxobjs> <grown> <reaped> "
+   "<error> <maxfreeable> <nodeallocs> <remotefrees> <alienoverflow>");
+ seq_puts(m, " : cpustat <allochit> <allocmiss> <freehit> <freemiss>");
+#endif
+ seq_putc(m, '\n');
+}
+
 static void *s_start(struct seq_file *m, loff_t *pos)
 {
  loff_t n = *pos;
diff --git a/mm/slub.c b/mm/slub.c
index 55304ed..91e1f3b 100644
--- a/mm/slub.c
+++ b/mm/slub.c
@@ -5394,16 +5394,6 @@ __initcall(slab_sysfs_init);
  * The /proc/slabinfo ABI
  */
 #ifdef CONFIG_SLABINFO
-void print_slabinfo_header(struct seq_file *m)
-{
```

```
- seq_puts(m, "slabinfo - version: 2.1\n");
- seq_puts(m, "# name            <active_objs> <num_objs> <object_size> "
-   "<objperslab> <pagesperslab>");
- seq_puts(m, " : tunables <limit> <batchcount> <sharedfactor>");
- seq_puts(m, " : slabdata <active_slabs> <num_slabs> <sharedavail>");
- seq_putc(m, '\n');
-}
-
 int slabinfo_show(struct seq_file *m, void *p)
 {
  unsigned long nr_partials = 0;
--
1.7.11.4
```

Subject: [PATCH v4 04/19] sl[au]b: process slabinfo_show in common code
Posted by Glauber Costa on Fri, 12 Oct 2012 13:40:58 GMT
View Forum Message <> Reply to Message

With all the infrastructure in place, we can now have slabinfo_show
done from slab_common.c. A cache-specific function is called to grab
information about the cache itself, since that is still heavily
dependent on the implementation. But with the values produced by it, all
the printing and handling is done from common code.

[v2: moved objects_per_slab and cache_order to slabinfo]

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 | 26 ++++++++++++++++++----------
 mm/slab.h | 16 ++++++++++++++-
 mm/slab_common.c | 18 ++++++++++++++++++
 mm/slub.c | 24 ++++++++++++++----------
 4 files changed, 57 insertions(+), 27 deletions(-)

diff --git a/mm/slab.c b/mm/slab.c
index 864a9e9..98b3460 100644
--- a/mm/slab.c
+++ b/mm/slab.c
@@ -4262,9 +4262,8 @@ out:
 }

 #ifdef CONFIG_SLABINFO
-int slabinfo_show(struct seq_file *m, void *p)
+void get_slabinfo(struct kmem_cache *cachep, struct slabinfo *sinfo)

```
 {
- struct kmem_cache *cachep = list_entry(p, struct kmem_cache, list);
  struct slab *slabp;
  unsigned long active_objs;
  unsigned long num_objs;
@@ -4319,13 +4318,20 @@ int slabinfo_show(struct seq_file *m, void *p)
  if (error)
   printk(KERN_ERR "slab: cache %s error: %s\n", name, error);

- seq_printf(m, "%-17s %6lu %6lu %6u %4u %4d",
-     name, active_objs, num_objs, cachep->size,
-     cachep->num, (1 << cachep->gfporder));
- seq_printf(m, " : tunables %4u %4u %4u",
-     cachep->limit, cachep->batchcount, cachep->shared);
- seq_printf(m, " : slabdata %6lu %6lu %6lu",
-     active_slabs, num_slabs, shared_avail);
+ sinfo->active_objs = active_objs;
+ sinfo->num_objs = num_objs;
+ sinfo->active_slabs = active_slabs;
+ sinfo->num_slabs = num_slabs;
+ sinfo->shared_avail = shared_avail;
+ sinfo->limit = cachep->limit;
+ sinfo->batchcount = cachep->batchcount;
+ sinfo->shared = cachep->shared;
+ sinfo->objects_per_slab = cachep->num;
+ sinfo->cache_order = cachep->gfporder;
+}
+
+void slabinfo_show_stats(struct seq_file *m, struct kmem_cache *cachep)
+{
 #if STATS
  {   /* list3 stats */
   unsigned long high = cachep->high_mark;
@@ -4355,8 +4361,6 @@ int slabinfo_show(struct seq_file *m, void *p)
       allochit, allocmiss, freehit, freemiss);
  }
 #endif
- seq_putc(m, '\n');
- return 0;
 }

 #define MAX_SLABINFO_WRITE 128
diff --git a/mm/slab.h b/mm/slab.h
index e9ba23f..66a62d3 100644
--- a/mm/slab.h
+++ b/mm/slab.h
@@ -74,8 +74,22 @@ int __kmem_cache_shutdown(struct kmem_cache *);
```

```
 struct seq_file;
 struct file;
-int slabinfo_show(struct seq_file *m, void *p);

+struct slabinfo {
+ unsigned long active_objs;
+ unsigned long num_objs;
+ unsigned long active_slabs;
+ unsigned long num_slabs;
+ unsigned long shared_avail;
+ unsigned int limit;
+ unsigned int batchcount;
+ unsigned int shared;
+ unsigned int objects_per_slab;
+ unsigned int cache_order;
+};
+
+void get_slabinfo(struct kmem_cache *s, struct slabinfo *sinfo);
+void slabinfo_show_stats(struct seq_file *m, struct kmem_cache *s);
 ssize_t slabinfo_write(struct file *file, const char __user *buffer,
         size_t count, loff_t *ppos);
 #endif
diff --git a/mm/slab_common.c b/mm/slab_common.c
index bb4d751..1ee1d6f 100644
--- a/mm/slab_common.c
+++ b/mm/slab_common.c
@@ -246,7 +246,23 @@ static void s_stop(struct seq_file *m, void *p)

 static int s_show(struct seq_file *m, void *p)
 {
- return slabinfo_show(m, p);
+ struct kmem_cache *s = list_entry(p, struct kmem_cache, list);
+ struct slabinfo sinfo;
+
+ memset(&sinfo, 0, sizeof(sinfo));
+ get_slabinfo(s, &sinfo);
+
+ seq_printf(m, "%-17s %6lu %6lu %6u %4u %4d",
+     s->name, sinfo.active_objs, sinfo.num_objs, s->size,
+     sinfo.objects_per_slab, (1 << sinfo.cache_order));
+
+ seq_printf(m, " : tunables %4u %4u %4u",
+     sinfo.limit, sinfo.batchcount, sinfo.shared);
+ seq_printf(m, " : slabdata %6lu %6lu %6lu",
+     sinfo.active_slabs, sinfo.num_slabs, sinfo.shared_avail);
+ slabinfo_show_stats(m, s);
+ seq_putc(m, '\n');
+ return 0;
```

```
 }

 /*
diff --git a/mm/slub.c b/mm/slub.c
index 91e1f3b..a34548e 100644
--- a/mm/slub.c
+++ b/mm/slub.c
@@ -5394,18 +5394,14 @@ __initcall(slab_sysfs_init);
  * The /proc/slabinfo ABI
  */
 #ifdef CONFIG_SLABINFO
-int slabinfo_show(struct seq_file *m, void *p)
+void get_slabinfo(struct kmem_cache *s, struct slabinfo *sinfo)
 {
  unsigned long nr_partials = 0;
  unsigned long nr_slabs = 0;
- unsigned long nr_inuse = 0;
  unsigned long nr_objs = 0;
  unsigned long nr_free = 0;
- struct kmem_cache *s;
  int node;

- s = list_entry(p, struct kmem_cache, list);
-
  for_each_online_node(node) {
   struct kmem_cache_node *n = get_node(s, node);

@@ -5418,16 +5414,16 @@ int slabinfo_show(struct seq_file *m, void *p)
   nr_free += count_partial(n, count_free);
  }

- nr_inuse = nr_objs - nr_free;
+ sinfo->active_objs = nr_objs - nr_free;
+ sinfo->num_objs = nr_objs;
+ sinfo->active_slabs = nr_slabs;
+ sinfo->num_slabs = nr_slabs;
+ sinfo->objects_per_slab = oo_objects(s->oo);
+ sinfo->cache_order = oo_order(s->oo);
+}

- seq_printf(m, "%-17s %6lu %6lu %6u %4u %4d", s->name, nr_inuse,
-     nr_objs, s->size, oo_objects(s->oo),
-     (1 << oo_order(s->oo)));
- seq_printf(m, " : tunables %4u %4u %4u", 0, 0, 0);
- seq_printf(m, " : slabdata %6lu %6lu %6lu", nr_slabs, nr_slabs,
-     0UL);
- seq_putc(m, '\n');
- return 0;
```

```
+void slabinfo_show_stats(struct seq_file *m, struct kmem_cache *s)
+{
 }

 ssize_t slabinfo_write(struct file *file, const char __user *buffer,
--
1.7.11.4
```

Subject: [PATCH v4 05/19] slab: don't preemptively remove element from list in cache destroy
Posted by Glauber Costa on Fri, 12 Oct 2012 13:40:59 GMT
View Forum Message <> Reply to Message

After the slab/slub/slob merge, we are deleting the element from the slab_cache lists, and then if the destruction fail, we add it back again. This behavior was present in some caches, but not in others, if my memory doesn't fail me.

I, however, see no reason why we need to do so, since we are now locked during the whole deletion (which wasn't necessarily true before). I propose a simplification in which we delete it only when there is no more going back, so we don't need to add it again.

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>
CC: Tejun Heo <tj@kernel.org>
---
 mm/slab_common.c | 5 ++---
 1 file changed, 2 insertions(+), 3 deletions(-)

```
diff --git a/mm/slab_common.c b/mm/slab_common.c
index 1ee1d6f..bf4b4f1 100644
--- a/mm/slab_common.c
+++ b/mm/slab_common.c
@@ -174,16 +174,15 @@ void kmem_cache_destroy(struct kmem_cache *s)
  mutex_lock(&slab_mutex);
  s->refcount--;
  if (!s->refcount) {
-  list_del(&s->list);
-
   if (!__kmem_cache_shutdown(s)) {
    if (s->flags & SLAB_DESTROY_BY_RCU)
```

```
     rcu_barrier();

+    list_del(&s->list);
+
     kfree(s->name);
     kmem_cache_free(kmem_cache, s);
   } else {
-    list_add(&s->list, &slab_caches);
     printk(KERN_ERR "kmem_cache_destroy %s: Slab cache still has objects\n",
      s->name);
     dump_stack();
--
1.7.11.4
```

Subject: [PATCH v4 06/19] slab/slub: struct memcg_params
Posted by Glauber Costa on Fri, 12 Oct 2012 13:41:00 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>
CC: Tejun Heo <tj@kernel.org>

```
 include/linux/slab.h     | 25 +++++++++++++++++++++++
 include/linux/slab_def.h |  3 +++
 include/linux/slub_def.h |  3 +++
 mm/slab.h                | 13 +++++++++++++
 4 files changed, 44 insertions(+)

diff --git a/include/linux/slab.h b/include/linux/slab.h
index 0dd2dfa..e4ea48a 100644
--- a/include/linux/slab.h
+++ b/include/linux/slab.h
@@ -177,6 +177,31 @@ unsigned int kmem_cache_size(struct kmem_cache *);
 #define ARCH_SLAB_MINALIGN __alignof__(unsigned long long)
 #endif

+#include <linux/workqueue.h>
+/*
+ * This is the main placeholder for memcg-related information in kmem caches.
```

```
+ * struct kmem_cache will hold a pointer to it, so the memory cost while
+ * disabled is 1 pointer. The runtime cost while enabled, gets bigger than it
+ * would otherwise be if that would be bundled in kmem_cache: we'll need an
+ * extra pointer chase. But the trade off clearly lays in favor of not
+ * penalizing non-users.
+ *
+ * Both the root cache and the child caches will have it. For the root cache,
+ * this will hold a dynamically allocated array large enough to hold
+ * information about the currently limited memcgs in the system.
+ *
+ * Child caches will hold extra metadata needed for its operation. Fields are:
+ *
+ * @memcg: pointer to the memcg this cache belongs to
+ */
+struct memcg_cache_params {
+ bool is_root_cache;
+ union {
+  struct kmem_cache *memcg_caches[0];
+  struct mem_cgroup *memcg;
+ };
+};
+
 /*
  * Common kmalloc functions provided by all allocators
  */
diff --git a/include/linux/slab_def.h b/include/linux/slab_def.h
index 36d7031..665afa4 100644
--- a/include/linux/slab_def.h
+++ b/include/linux/slab_def.h
@@ -81,6 +81,9 @@ struct kmem_cache {
   */
  int obj_offset;
 #endif /* CONFIG_DEBUG_SLAB */
+#ifdef CONFIG_MEMCG_KMEM
+ struct memcg_cache_params *memcg_params;
+#endif

 /* 6) per-cpu/per-node data, touched during every alloc/free */
  /*
diff --git a/include/linux/slub_def.h b/include/linux/slub_def.h
index df448ad..961e72e 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 memcg_cache_params *memcg_params;
+#endif

 #ifdef CONFIG_NUMA
  /*
diff --git a/mm/slab.h b/mm/slab.h
index 66a62d3..5ee1851 100644
--- a/mm/slab.h
+++ b/mm/slab.h
@@ -92,4 +92,17 @@ void get_slabinfo(struct kmem_cache *s, struct slabinfo *sinfo);
 void slabinfo_show_stats(struct seq_file *m, struct kmem_cache *s);
 ssize_t slabinfo_write(struct file *file, const char __user *buffer,
         size_t count, loff_t *ppos);
+
+#ifdef CONFIG_MEMCG_KMEM
+static inline bool is_root_cache(struct kmem_cache *s)
+{
+ return !s->memcg_params || s->memcg_params->is_root_cache;
+}
+#else
+static inline bool is_root_cache(struct kmem_cache *s)
+{
+ return true;
+}
+
+#endif
 #endif
--
1.7.11.4
```

Subject: [PATCH v4 07/19] consider a memcg parameter in kmem_create_cache
Posted by Glauber Costa on Fri, 12 Oct 2012 13:41:01 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.

A helper is provided, 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.

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

```
[ v4: no longer using the index mechanism ]

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>
CC: Tejun Heo <tj@kernel.org>
---
 include/linux/memcontrol.h | 22 ++++++++++++++++++++++
 include/linux/slab.h       | 12 ++++++++++++-
 mm/memcontrol.c            | 49 +++++++++++++++++++++++++++++++++++++++++++++++++
 mm/slab.h                  | 23 +++++++++++++++++++----
 mm/slab_common.c           | 40 ++++++++++++++++++++++++++++++----------
 mm/slub.c                  | 16 ++++++++++++----
 6 files changed, 143 insertions(+), 19 deletions(-)

diff --git a/include/linux/memcontrol.h b/include/linux/memcontrol.h
index 34e96cf..957da60 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 {
@@ -414,6 +415,11 @@ void __memcg_kmem_commit_charge(struct page *page,
          struct mem_cgroup *memcg, int order);
 void __memcg_kmem_uncharge_page(struct page *page, int order);

+int memcg_css_id(struct mem_cgroup *memcg);
+int memcg_register_cache(struct mem_cgroup *memcg, struct kmem_cache *s);
+void memcg_release_cache(struct kmem_cache *cachep);
+void memcg_cache_list_add(struct mem_cgroup *memcg, struct kmem_cache *cachep);
+
 /**
  * memcg_kmem_newpage_charge: verify if a new kmem allocation is allowed.
  * @gfp: the gfp allocation flags.
@@ -506,6 +512,22 @@ static inline void
 memcg_kmem_commit_charge(struct page *page, struct mem_cgroup *memcg, int order)
 {
 }
+
```

```
+static inline int memcg_register_cache(struct mem_cgroup *memcg,
+           struct kmem_cache *s)
+{
+ return 0;
+}
+
+static inline void memcg_release_cache(struct kmem_cache *cachep)
+{
+}
+
+static inline void memcg_cache_list_add(struct mem_cgroup *memcg,
+     struct kmem_cache *s)
+{
+ BUG();
+}
 #endif /* CONFIG_MEMCG_KMEM */
 #endif /* _LINUX_MEMCONTROL_H */

diff --git a/include/linux/slab.h b/include/linux/slab.h
index e4ea48a..b22a158 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 *);
@@ -193,15 +197,21 @@ unsigned int kmem_cache_size(struct kmem_cache *);
  * Child caches will hold extra metadata needed for its operation. Fields are:
  *
  * @memcg: pointer to the memcg this cache belongs to
+ * @root_cache: pointer to the global, root cache, this cache was derived from
  */
 struct memcg_cache_params {
  bool is_root_cache;
```

```
  union {
   struct kmem_cache *memcg_caches[0];
-  struct mem_cgroup *memcg;
+  struct {
+   struct mem_cgroup *memcg;
+   struct kmem_cache *root_cache;
+  };
  };
 };

+int memcg_update_all_caches(int num_memcgs);
+
 /*
  * Common kmalloc functions provided by all allocators
  */
diff --git a/mm/memcontrol.c b/mm/memcontrol.c
index b3a22a6..fff089e 100644
--- a/mm/memcontrol.c
+++ b/mm/memcontrol.c
@@ -339,6 +339,12 @@ struct mem_cgroup {
 #if defined(CONFIG_MEMCG_KMEM) && defined(CONFIG_INET)
  struct tcp_memcontrol tcp_mem;
 #endif
+#if defined(CONFIG_MEMCG_KMEM)
+ /* analogous to slab_common's slab_caches list. per-memcg */
+ struct list_head memcg_slab_caches;
+ /* Not a spinlock, we can take a lot of time walking the list */
+ struct mutex slab_caches_mutex;
+#endif
 };

 /* internal only representation about the status of kmem accounting. */
@@ -2754,6 +2760,49 @@ static void memcg_uncharge_kmem(struct mem_cgroup *memcg,
u64 size)
   mem_cgroup_put(memcg);
 }

+void memcg_cache_list_add(struct mem_cgroup *memcg, struct kmem_cache *cachep)
+{
+ mutex_lock(&memcg->slab_caches_mutex);
+ list_add(&cachep->list, &memcg->memcg_slab_caches);
+ mutex_unlock(&memcg->slab_caches_mutex);
+}
+
+/*
+ * helper for acessing a memcg's index. It will be used as an index in the
+ * child cache array in kmem_cache, and also to derive its name.
+ *
```

```
+ * According to kernel/cgroup.c, needs to be rcu protected.
+ */
+int memcg_css_id(struct mem_cgroup *memcg)
+{
+ int id;
+ rcu_read_lock();
+ id = css_id(&memcg->css);
+ rcu_read_unlock();
+ return id;
+}
+
+int memcg_register_cache(struct mem_cgroup *memcg, struct kmem_cache *s)
+{
+ size_t size = sizeof(struct memcg_cache_params);
+
+ if (!memcg_kmem_enabled())
+  return 0;
+
+ s->memcg_params = kzalloc(size, GFP_KERNEL);
+ if (!s->memcg_params)
+  return -ENOMEM;
+
+ if (memcg)
+  s->memcg_params->memcg = memcg;
+ return 0;
+}
+
+void memcg_release_cache(struct kmem_cache *s)
+{
+ kfree(s->memcg_params);
+}
+
 /*
  * 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.h b/mm/slab.h
index 5ee1851..c35ecce 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

@@ -98,11 +101,23 @@ static inline bool is_root_cache(struct kmem_cache *s)
 {
  return !s->memcg_params || s->memcg_params->is_root_cache;
 }
+
+static inline bool cache_match_memcg(struct kmem_cache *cachep,
+         struct mem_cgroup *memcg)
+{
+ return (is_root_cache(cachep) && !memcg) ||
+  (cachep->memcg_params->memcg == memcg);
+}
 #else
 static inline bool is_root_cache(struct kmem_cache *s)
 {
  return true;
 }

+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 bf4b4f1..f97f7b8 100644
--- a/mm/slab_common.c
+++ b/mm/slab_common.c
@@ -18,6 +18,7 @@
 #include <asm/cacheflush.h>
 #include <asm/tlbflush.h>
 #include <asm/page.h>
+#include <linux/memcontrol.h>

 #include "slab.h"

@@ -27,7 +28,8 @@ 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;

@@ -53,7 +55,7 @@ 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();
@@ -66,7 +68,8 @@ 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;
 }
@@ -97,8 +100,9 @@ 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;
@@ -106,7 +110,7 @@ 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;
```

```
  /*
@@ -117,7 +121,7 @@ struct kmem_cache *kmem_cache_create(const char *name, size_t size,
size_t align
   */
  flags &= CACHE_CREATE_MASK;

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

@@ -126,6 +130,13 @@ 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;
+
+  if (memcg_register_cache(memcg, s)) {
+   kmem_cache_free(kmem_cache, s);
+   err = -ENOMEM;
+   goto out_locked;
+  }
+
   s->name = kstrdup(name, GFP_KERNEL);
   if (!s->name) {
    kmem_cache_free(kmem_cache, s);
@@ -135,10 +146,11 @@ 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);
-
+   if (!memcg)
+    list_add(&s->list, &slab_caches);
+   else
+    memcg_cache_list_add(memcg, s);
   } else {
    kfree(s->name);
    kmem_cache_free(kmem_cache, s);
@@ -166,6 +178,13 @@ 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)
@@ -180,6 +199,7 @@ void kmem_cache_destroy(struct kmem_cache *s)

    list_del(&s->list);

+   memcg_release_cache(s);
    kfree(s->name);
    kmem_cache_free(kmem_cache, s);
   } else {
diff --git a/mm/slub.c b/mm/slub.c
index a34548e..05aefe2 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>

@@ -3880,7 +3881,7 @@ 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 *))
 {
@@ -3916,17 +3917,20 @@ 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++;
   /*
@@ -5246,6 +5250,10 @@ 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 (!is_root_cache(s))
+  p += sprintf(p, "-%08d", memcg_css_id(s->memcg_params->memcg));
+#endif
  BUG_ON(p > name + ID_STR_LENGTH - 1);
  return name;
 }
--
1.7.11.4
```

Subject: [PATCH v4 08/19] Allocate memory for memcg caches whenever a new memcg appears
Posted by Glauber Costa on Fri, 12 Oct 2012 13:41:02 GMT
View Forum Message <> Reply to Message

Every cache that is considered a root cache (basically the "original" caches,
tied to the root memcg/no-memcg) will have an array that should be large enough
to store a cache pointer per each memcg in the system.

Theoreticaly, this is as high as 1 << sizeof(css_id), which is currently in the
64k pointers range. Most of the time, we won't be using that much.

What goes in this patch, is a simple scheme to dynamically allocate such an
array, in order to minimize memory usage for memcg caches. Because we would
also like to avoid allocations all the time, at least for now, the array will
only grow. It will tend to be big enough to hold the maximum number of
kmem-limited memcgs ever achieved.

We'll allocate it to be a minimum of 64 kmem-limited memcgs. When we have more

than that, we'll start doubling the size of this array every time the limit is
reached.

Because we are only considering kmem limited memcgs, a natural point for this
to happen is when we write to the limit. At that point, we already have
set_limit_mutex held, so that will become our natural synchronization
mechanism.

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>
CC: Tejun Heo <tj@kernel.org>
---
 include/linux/memcontrol.h | 2 +
 mm/memcontrol.c | 171 ++++++++++++++++++++++++++++++++++++++-----
 mm/slab_common.c | 25 +++++++
 3 files changed, 181 insertions(+), 17 deletions(-)

diff --git a/include/linux/memcontrol.h b/include/linux/memcontrol.h
index 957da60..e8d0571 100644
--- a/include/linux/memcontrol.h
+++ b/include/linux/memcontrol.h
@@ -420,6 +420,8 @@ int memcg_register_cache(struct mem_cgroup *memcg, struct
kmem_cache *s);
 void memcg_release_cache(struct kmem_cache *cachep);
 void memcg_cache_list_add(struct mem_cgroup *memcg, struct kmem_cache *cachep);

+int memcg_update_cache_size(struct kmem_cache *s, int num_groups);
+void memcg_update_array_size(int num_groups);
 /**
  * memcg_kmem_newpage_charge: verify if a new kmem allocation is allowed.
  * @gfp: the gfp allocation flags.
diff --git a/mm/memcontrol.c b/mm/memcontrol.c
index fff089e..8c5c570 100644
--- a/mm/memcontrol.c
+++ b/mm/memcontrol.c
@@ -371,9 +371,15 @@ static bool memcg_kmem_is_active(struct mem_cgroup *memcg)
  return test_bit(KMEM_ACCOUNTED_ACTIVE, &memcg->kmem_accounted);
 }

-static void memcg_kmem_set_activated(struct mem_cgroup *memcg)
+static bool memcg_kmem_set_activated(struct mem_cgroup *memcg)
 {
- set_bit(KMEM_ACCOUNTED_ACTIVATED, &memcg->kmem_accounted);

```
+ return !test_and_set_bit(KMEM_ACCOUNTED_ACTIVATED,
+     &memcg->kmem_accounted);
+}
+
+static void memcg_kmem_clear_activated(struct mem_cgroup *memcg)
+{
+ clear_bit(KMEM_ACCOUNTED_ACTIVATED, &memcg->kmem_accounted);
 }

 static void memcg_kmem_mark_dead(struct mem_cgroup *memcg)
@@ -546,6 +552,17 @@ static void disarm_sock_keys(struct mem_cgroup *memcg)
 #endif

 #ifdef CONFIG_MEMCG_KMEM
+static int memcg_limited_groups_array_size;
+#define MEMCG_CACHES_MIN_SIZE 64
+/*
+ * MAX_SIZE should be as large as the number of css_ids. Ideally, we could get
+ * this constant directly from cgroup, but it is understandable that this is
+ * better kept as an internal representation in cgroup.c
+ *
+ * As of right now, this should be enough.
+ */
+#define MEMCG_CACHES_MAX_SIZE 65535
+
 struct static_key memcg_kmem_enabled_key;

 static void disarm_kmem_keys(struct mem_cgroup *memcg)
@@ -2782,6 +2799,115 @@ int memcg_css_id(struct mem_cgroup *memcg)
  return id;
 }

+/*
+ * This ends up being protected by the set_limit mutex, during normal
+ * operation, because that is its main call site.
+ *
+ * But when we create a new cache, we can call this as well if its parent
+ * is kmem-limited. That will have to hold set_limit_mutex as well.
+ */
+int memcg_update_cache_sizes(struct mem_cgroup *memcg)
+{
+ int num, ret;
+ /*
+  * After this point, kmem_accounted (that we test atomically in
+  * the beginning of this conditional), is no longer 0. This
+  * guarantees only one process will set the following boolean
+  * to true. We don't need test_and_set because we're protected
+  * by the set_limit_mutex anyway.
```

```
+  */
+ if (!memcg_kmem_set_activated(memcg))
+  return 0;
+
+ num = memcg_css_id(memcg);
+ ret = memcg_update_all_caches(num);
+ if (ret) {
+  memcg_kmem_clear_activated(memcg);
+  return ret;
+ }
+
+ INIT_LIST_HEAD(&memcg->memcg_slab_caches);
+ mutex_init(&memcg->slab_caches_mutex);
+ return 0;
+}
+
+static size_t memcg_caches_array_size(int num_groups)
+{
+ ssize_t size;
+ if (num_groups <= 0)
+  return 0;
+
+ size = 2 * num_groups;
+ if (size < MEMCG_CACHES_MIN_SIZE)
+  size = MEMCG_CACHES_MIN_SIZE;
+ else if (size > MEMCG_CACHES_MAX_SIZE)
+  size = MEMCG_CACHES_MAX_SIZE;
+
+ return size;
+}
+
+/*
+ * We should update the current array size iff all caches updates succeed. This
+ * can only be done from the slab side. The slab mutex needs to be held when
+ * calling this.
+ */
+void memcg_update_array_size(int num)
+{
+ if (num > memcg_limited_groups_array_size)
+  memcg_limited_groups_array_size = memcg_caches_array_size(num);
+}
+
+int memcg_update_cache_size(struct kmem_cache *s, int num_groups)
+{
+ struct memcg_cache_params *cur_params = s->memcg_params;
+
+ VM_BUG_ON(s->memcg_params && !s->memcg_params->is_root_cache);
+
```

```
+ if (num_groups > memcg_limited_groups_array_size) {
+  int i;
+  ssize_t size = memcg_caches_array_size(num_groups);
+
+  size *= sizeof(void *);
+  size += sizeof(struct memcg_cache_params);
+
+  s->memcg_params = kzalloc(size ,GFP_KERNEL);
+  if (!s->memcg_params) {
+   s->memcg_params = cur_params;
+   return -ENOMEM;
+  }
+
+  s->memcg_params->is_root_cache = true;
+
+  /*
+   * There is the chance it will be bigger than
+   * memcg_limited_groups_array_size, if we failed an allocation
+   * in a cache, in which case all caches updated before it, will
+   * have a bigger array.
+   *
+   * But if that is the case, the data after
+   * memcg_limited_groups_array_size is certainly unused
+   */
+  for (i = 0; memcg_limited_groups_array_size; i++) {
+   if (!cur_params->memcg_caches[i])
+    continue;
+   s->memcg_params->memcg_caches[i] =
+       cur_params->memcg_caches[i];
+  }
+
+  /*
+   * Ideally, we would wait until all caches succeed, and only
+   * then free the old one. But this is not worth the extra
+   * pointer per-cache we'd have to have for this.
+   *
+   * It is not a big deal if some caches are left with a size
+   * bigger than the others. And all updates will reset this
+   * anyway.
+   */
+  kfree(cur_params);
+ }
+ return 0;
+}
+
 int memcg_register_cache(struct mem_cgroup *memcg, struct kmem_cache *s)
 {
  size_t size = sizeof(struct memcg_cache_params);
```

```
@@ -2789,6 +2915,9 @@ int memcg_register_cache(struct mem_cgroup *memcg, struct
kmem_cache *s)
  if (!memcg_kmem_enabled())
   return 0;

+ if (!memcg)
+  size =+ memcg_limited_groups_array_size * sizeof(void *);
+
  s->memcg_params = kzalloc(size, GFP_KERNEL);
  if (!s->memcg_params)
   return -ENOMEM;
@@ -4291,14 +4420,11 @@ static int memcg_update_kmem_limit(struct cgroup *cont, u64 val)
   ret = res_counter_set_limit(&memcg->kmem, val);
   VM_BUG_ON(ret);

-  /*
-   * After this point, kmem_accounted (that we test atomically in
-   * the beginning of this conditional), is no longer 0. This
-   * guarantees only one process will set the following boolean
-   * to true. We don't need test_and_set because we're protected
-   * by the set_limit_mutex anyway.
-   */
-  memcg_kmem_set_activated(memcg);
+  ret = memcg_update_cache_sizes(memcg);
+  if (ret) {
+   res_counter_set_limit(&memcg->kmem, RESOURCE_MAX);
+   goto out;
+  }
   must_inc_static_branch = true;
   /*
    * kmem charges can outlive the cgroup. In the case of slab
@@ -4337,9 +4463,10 @@ out:
  return ret;
 }

-static void memcg_propagate_kmem(struct mem_cgroup *memcg,
-     struct mem_cgroup *parent)
+static int memcg_propagate_kmem(struct mem_cgroup *memcg,
+    struct mem_cgroup *parent)
 {
+ int ret = 0;
  memcg->kmem_accounted = parent->kmem_accounted;
 #ifdef CONFIG_MEMCG_KMEM
  /*
@@ -4352,11 +4479,19 @@ static void memcg_propagate_kmem(struct mem_cgroup *memcg,
   * It is a lot simpler just to do static_key_slow_inc() on every child
   * that is accounted.
   */
```

```
- if (memcg_kmem_is_active(memcg)) {
-  mem_cgroup_get(memcg);
-  static_key_slow_inc(&memcg_kmem_enabled_key);
- }
+ if (!memcg_kmem_is_active(memcg))
+  return ret;
+
+ mutex_lock(&set_limit_mutex);
+ ret = memcg_update_cache_sizes(memcg);
+ mutex_unlock(&set_limit_mutex);
+ if (ret)
+  return ret;
+
+ mem_cgroup_get(memcg);
+ static_key_slow_inc(&memcg_kmem_enabled_key);
 #endif
+ return ret;
 }

 /*
@@ -5441,8 +5576,10 @@ mem_cgroup_create(struct cgroup *cont)
    * This refcnt will be decremented when freeing this
    * mem_cgroup(see mem_cgroup_put).
    */
+  error = memcg_propagate_kmem(memcg, parent);
+  if (error)
+   goto free_out;
   mem_cgroup_get(parent);
-  memcg_propagate_kmem(memcg, parent);
  } else {
   res_counter_init(&memcg->res, NULL);
   res_counter_init(&memcg->memsw, NULL);
diff --git a/mm/slab_common.c b/mm/slab_common.c
index f97f7b8..fcf59d7 100644
--- a/mm/slab_common.c
+++ b/mm/slab_common.c
@@ -75,6 +75,31 @@ static inline int kmem_cache_sanity_check(struct mem_cgroup *memcg,
 }
 #endif

+#ifdef CONFIG_MEMCG_KMEM
+int memcg_update_all_caches(int num_memcgs)
+{
+ struct kmem_cache *s;
+ int ret = 0;
+ mutex_lock(&slab_mutex);
+
+ list_for_each_entry(s, &slab_caches, list) {
```

```
+  ret = memcg_update_cache_size(s, num_memcgs);
+  /*
+   * See comment in memcontrol.c, memcg_update_cache_size:
+   * Instead of freeing the memory, we'll just leave the caches
+   * up to this point in an updated state.
+   */
+  if (ret)
+   goto out;
+ }
+
+ memcg_update_array_size(num_memcgs);
+out:
+ mutex_unlock(&slab_mutex);
+ return ret;
+}
+#endif
+
 /*
  * kmem_cache_create - Create a cache.
  * @name: A string which is used in /proc/slabinfo to identify this cache.
--
1.7.11.4
```

Subject: [PATCH v4 09/19] memcg: infrastructure to match an allocation to the right cache

Posted by Glauber Costa on Fri, 12 Oct 2012 13:41:03 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.

[v4: use a standard workqueue mechanism, create right away if
  possible, index from cache side]

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>
CC: Tejun Heo <tj@kernel.org>
---
 include/linux/memcontrol.h | 42 ++++++++++
 init/Kconfig | 2 +-
 mm/memcontrol.c | 188 ++
 3 files changed, 231 insertions(+), 1 deletion(-)

diff --git a/include/linux/memcontrol.h b/include/linux/memcontrol.h
index e8d0571..c7886aa 100644
--- a/include/linux/memcontrol.h
+++ b/include/linux/memcontrol.h
@@ -422,6 +422,10 @@ void memcg_cache_list_add(struct mem_cgroup *memcg, struct
kmem_cache *cachep);

 int memcg_update_cache_size(struct kmem_cache *s, int num_groups);
 void memcg_update_array_size(int num_groups);
+
+struct kmem_cache *
+__memcg_kmem_get_cache(struct kmem_cache *cachep, gfp_t gfp);
+
 /**
  * memcg_kmem_newpage_charge: verify if a new kmem allocation is allowed.
  * @gfp: the gfp allocation flags.
@@ -492,6 +496,38 @@ memcg_kmem_commit_charge(struct page *page, struct mem_cgroup
*memcg, int order)
   __memcg_kmem_commit_charge(page, memcg, order);
 }

+/**
+ * memcg_kmem_get_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;
+        if (unlikely(test_thread_flag(TIF_MEMDIE)
+                     || fatal_signal_pending(current)))
+  return cachep;
+
+ return __memcg_kmem_get_cache(cachep, gfp);
+}
 #else
 static inline void sock_update_memcg(struct sock *sk)
 {
@@ -530,6 +566,12 @@ static inline void memcg_cache_list_add(struct mem_cgroup *memcg,
 {
  BUG();
 }
+
+static inline struct kmem_cache *
+memcg_kmem_get_cache(struct kmem_cache *cachep, gfp_t gfp)
+{
+ return cachep;
+}
 #endif /* CONFIG_MEMCG_KMEM */
 #endif /* _LINUX_MEMCONTROL_H */

diff --git a/init/Kconfig b/init/Kconfig
index af6c7f8..62b1f28 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 8c5c570..148baad 100644
--- a/mm/memcontrol.c
+++ b/mm/memcontrol.c
@@ -563,7 +563,14 @@ static int memcg_limited_groups_array_size;
  */
 #define MEMCG_CACHES_MAX_SIZE 65535

+/*
+ * 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 void disarm_kmem_keys(struct mem_cgroup *memcg)
 {
@@ -2929,9 +2936,190 @@ int memcg_register_cache(struct mem_cgroup *memcg, struct
kmem_cache *s)

 void memcg_release_cache(struct kmem_cache *s)
 {
+ struct kmem_cache *root;
+ int id = memcg_css_id(s->memcg_params->memcg);
+
+ if (s->memcg_params->is_root_cache)
+  goto out;
+
+ root = s->memcg_params->root_cache;
+ root->memcg_params->memcg_caches[id] = NULL;
+ mem_cgroup_put(s->memcg_params->memcg);
+out:
  kfree(s->memcg_params);
 }

+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;
+}
+
+/*
+ * This lock protects updaters, not readers. We want readers to be as fast as
+ * they can, and they will either see NULL or a valid cache value. Our model
+ * allow them to see NULL, in which case the root memcg will be selected.
+ *
+ * We need this lock because multiple allocations to the same cache from a non
+ * GFP_WAIT area will span more than one worker. Only one of them can create
+ * the cache.
+ */
+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 = memcg_css_id(memcg);
+
```

```
+ mutex_lock(&memcg_cache_mutex);
+ new_cachep = cachep->memcg_params->memcg_caches[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);
+ cachep->memcg_params->memcg_caches[idx] = new_cachep;
+ wmb(); /* the readers won't lock, make sure everybody sees it */
+ new_cachep->memcg_params->memcg = memcg;
+ new_cachep->memcg_params->root_cache = cachep;
+out:
+ mutex_unlock(&memcg_cache_mutex);
+ return new_cachep;
+}
+
+struct create_work {
+ struct mem_cgroup *memcg;
+ struct kmem_cache *cachep;
+ struct work_struct work;
+};
+
+static void memcg_create_cache_work_func(struct work_struct *w)
+{
+ struct create_work *cw;
+
+ cw = container_of(w, struct create_work, work);
+ memcg_create_kmem_cache(cw->memcg, cw->cachep);
+ /* Drop the reference gotten when we enqueued. */
+ css_put(&cw->memcg->css);
+ kfree(cw);
+}
+
+/*
+ * 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;
+
```

```
+ cw = kmalloc(sizeof(struct create_work), GFP_NOWAIT);
+ if (cw == NULL)
+  return;
+
+ /* The corresponding put will be done in the workqueue. */
+ if (!css_tryget(&memcg->css))
+  return;
+
+ cw->memcg = memcg;
+ cw->cachep = cachep;
+
+ INIT_WORK(&cw->work, memcg_create_cache_work_func);
+ schedule_work(&cw->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;
+
+ if (cachep->memcg_params && cachep->memcg_params->memcg)
+  return cachep;
+
+ rcu_read_lock();
+ memcg = mem_cgroup_from_task(rcu_dereference(current->mm->owner));
+ rcu_read_unlock();
+
+ if (!memcg_can_account_kmem(memcg))
+  return cachep;
+
+ idx = memcg_css_id(memcg);
+ VM_BUG_ON(idx == -1);
+
+ if (cachep->memcg_params->memcg_caches[idx] == NULL) {
```

```
+  /*
+   * If we are in a safe context, better to be predictable and
+   * return right away. This guarantees that the allocation being
+   * performed already belongs in the new cache. When we enqueue,
+   * we can't do that, and at least the current allocation will
+   * be relayed to the root cache while our cache is being
+   * created.
+   */
+  if ((gfp & __GFP_WAIT) && !in_interrupt())
+   return memcg_create_kmem_cache(memcg, cachep);
+  else {
+   memcg_create_cache_enqueue(memcg, cachep);
+   return cachep;
+  }
+  return cachep;
+ }
+
+ return cachep->memcg_params->memcg_caches[idx];
+}
+EXPORT_SYMBOL(__memcg_kmem_get_cache);
+
 /*
  * 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
--
1.7.11.4
```

Subject: [PATCH v4 10/19] memcg: skip memcg kmem allocations in specified code regions

Posted by Glauber Costa on Fri, 12 Oct 2012 13:41:04 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.

[v2: wrap the whole enqueue process, INIT_WORK can alloc memory]

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>
CC: Tejun Heo <tj@kernel.org>

```
 include/linux/sched.h |  1 +
 mm/memcontrol.c       | 63 +++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++--
 2 files changed, 62 insertions(+), 2 deletions(-)

diff --git a/include/linux/sched.h b/include/linux/sched.h
index 0d907e1..9fad6c1 100644
--- a/include/linux/sched.h
+++ b/include/linux/sched.h
@@ -1581,6 +1581,7 @@ 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 148baad..96916bd 100644
--- a/mm/memcontrol.c
+++ b/mm/memcontrol.c
@@ -2949,6 +2949,41 @@ out:
  kfree(s->memcg_params);
 }

+/*
+ * During the creation a new cache, we need to disable our accounting mechanism
+ * altogether. This is true even if we are not creating, but rather just
+ * enqueing new caches to be created.
+ *
+ * This is because that process will trigger allocations; some visible, like
+ * explicit kmallocs to auxiliary data structures, name strings and internal
+ * cache structures; some well concealed, like INIT_WORK() that can allocate
+ * objects during debug.
+ *
+ * If any allocation happens during memcg_kmem_get_cache, we will recurse back
+ * to it. This may not be a bounded recursion: since the first cache creation
+ * failed to complete (waiting on the allocation), we'll just try to create the
+ * cache again, failing at the same point.
+ *
+ * memcg_kmem_get_cache is prepared to abort after seeing a positive count of
+ * memcg_kmem_skip_account. So we enclose anything that might allocate memory
+ * inside the following two functions.
+ */
+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;
@@ -3008,7 +3043,10 @@ static struct kmem_cache *memcg_create_kmem_cache(struct
mem_cgroup *memcg,
  if (new_cachep)
   goto out;

+ /* Don't block progress to enqueue caches for internal infrastructure */
+ memcg_stop_kmem_account();
  new_cachep = kmem_cache_dup(memcg, cachep);
+ memcg_resume_kmem_account();

  if (new_cachep == NULL) {
   new_cachep = cachep;
@@ -3046,8 +3084,8 @@ static void memcg_create_cache_work_func(struct work_struct *w)
  * 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)
+static void __memcg_create_cache_enqueue(struct mem_cgroup *memcg,
+      struct kmem_cache *cachep)
 {
  struct create_work *cw;

@@ -3066,6 +3104,24 @@ static void memcg_create_cache_enqueue(struct mem_cgroup
*memcg,
  schedule_work(&cw->work);
 }

+static void memcg_create_cache_enqueue(struct mem_cgroup *memcg,
+          struct kmem_cache *cachep)
```

```
+{
+ /*
+  * We need to stop accounting when we kmalloc, because if the
+  * corresponding kmalloc cache is not yet created, the first allocation
+  * in __memcg_create_cache_enqueue will recurse.
+  *
+  * However, it is better to enclose the whole function. Depending on
+  * the debugging options enabled, INIT_WORK(), for instance, can
+  * trigger an allocation. This too, will make us recurse. Because at
+  * this point we can't allow ourselves back into memcg_kmem_get_cache,
+  * the safest choice is to do it like this, wrapping the whole function.
+  */
+ memcg_stop_kmem_account();
+ __memcg_create_cache_enqueue(memcg, cachep);
+ memcg_resume_kmem_account();
+}
 /*
  * 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.
@@ -3085,6 +3141,9 @@ struct kmem_cache *__memcg_kmem_get_cache(struct kmem_cache
*cachep,
  struct mem_cgroup *memcg;
  int idx;

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

--
1.7.11.4
```

Subject: [PATCH v4 11/19] sl[au]b: always get the cache from its page in kfree
Posted by Glauber Costa on Fri, 12 Oct 2012 13:41:05 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]
[v4: make it faster when kmemcg not in use]

```
Signed-off-by: Glauber Costa <glommer@parallels.com>
CC: Christoph Lameter <cl@linux.com>
CC: Pekka Enberg <penberg@cs.helsinki.fi>
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: Tejun Heo <tj@kernel.org>
---
 include/linux/memcontrol.h |  4 ++++
 mm/slab.c                  | 17 +++++++++++++++-
 mm/slab.h                  | 13 +++++++++++++
 mm/slub.c                  | 14 +++++++++++--
 4 files changed, 45 insertions(+), 3 deletions(-)

diff --git a/include/linux/memcontrol.h b/include/linux/memcontrol.h
index c7886aa..4c94182 100644
--- a/include/linux/memcontrol.h
+++ b/include/linux/memcontrol.h
@@ -536,6 +536,10 @@ static inline void sock_release_memcg(struct sock *sk)
 {
 }

+static inline bool memcg_kmem_enabled(void)
+{
+ return false;
+}
 static inline bool
 memcg_kmem_newpage_charge(gfp_t gfp, struct mem_cgroup **memcg, int order)
 {
diff --git a/mm/slab.c b/mm/slab.c
index 98b3460..6f22067 100644
--- a/mm/slab.c
+++ b/mm/slab.c
@@ -3911,9 +3911,24 @@ 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;
+
+ /*
```

```
+  * When kmemcg is not being used, both assignments should return the
+  * same value. but we don't want to pay the assignment price in that
+  * case. If it is not compiled in, the compiler should be smart enough
+  * to not do even the assignment. In that case, slab_equal_or_root
+  * will also be a constant.
+  */
+ if (memcg_kmem_enabled()) {
+  cachep = virt_to_cache(objp);
+  VM_BUG_ON(!slab_equal_or_root(cachep, s));
+ } else
+  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 c35ecce..b9b5f1f 100644
--- a/mm/slab.h
+++ b/mm/slab.h
@@ -108,6 +108,13 @@ static inline bool cache_match_memcg(struct kmem_cache *cachep,
  return (is_root_cache(cachep) && !memcg) ||
   (cachep->memcg_params->memcg == memcg);
 }
+
+static inline bool slab_equal_or_root(struct kmem_cache *s,
+     struct kmem_cache *p)
+{
+ return (p == s) ||
+  (s->memcg_params && (p == s->memcg_params->root_cache));
+}
 #else
 static inline bool is_root_cache(struct kmem_cache *s)
 {
@@ -119,5 +126,11 @@ static inline bool cache_match_memcg(struct kmem_cache *cachep,
 {
  return true;
 }
+
+static inline bool slab_equal_or_root(struct kmem_cache *s,
+     struct kmem_cache *p)
+{
+ return true;
+}
 #endif
 #endif
diff --git a/mm/slub.c b/mm/slub.c
index 05aefe2..6e1a90f 100644
--- a/mm/slub.c
```

```
+++ b/mm/slub.c
@@ -2609,9 +2609,19 @@ redo:

 void kmem_cache_free(struct kmem_cache *s, void *x)
 {
- struct page *page;
+ struct page *page = virt_to_head_page(x);

- page = virt_to_head_page(x);
+ /*
+  * When kmemcg is not being used, both assignments should return the
+  * same value. but we don't want to pay the assignment price in that
+  * case. If it is not compiled in, the compiler should be smart enough
+  * to not do even the assignment. In that case, slab_equal_or_root
+  * will also be a constant.
+  */
+ if (memcg_kmem_enabled()) {
+  VM_BUG_ON(!slab_equal_or_root(page->slab, s));
+  s = page->slab;
+ }

  if (kmem_cache_debug(s) && page->slab != s) {
   pr_err("kmem_cache_free: Wrong slab cache. %s but object"
--
1.7.11.4
```

Subject: [PATCH v4 12/19] sl[au]b: Allocate objects from memcg cache
Posted by Glauber Costa on Fri, 12 Oct 2012 13:41:06 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>
CC: Tejun Heo <tj@kernel.org>

```
 include/linux/slub_def.h | 15 ++++++++++-----
 mm/memcontrol.c          |  3 +++
 mm/slab.c                |  6 +++++-
 mm/slub.c                |  5 +++--
 4 files changed, 21 insertions(+), 8 deletions(-)

diff --git a/include/linux/slub_def.h b/include/linux/slub_def.h
index 961e72e..ed330df 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 96916bd..4d2a01f 100644
--- a/mm/memcontrol.c
+++ b/mm/memcontrol.c
@@ -3014,6 +3014,9 @@ 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);

+ if (new)
+  new->allocflags |= __GFP_KMEMCG;
+
  kfree(name);
  return new;
 }
diff --git a/mm/slab.c b/mm/slab.c
index 6f22067..03952c4 100644
--- a/mm/slab.c
+++ b/mm/slab.c
@@ -1949,7 +1949,7 @@ 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)
@@ -3514,6 +3514,8 @@ __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);

@@ -3599,6 +3601,8 @@ __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 6e1a90f..257e130 100644
--- a/mm/slub.c
+++ b/mm/slub.c
@@ -1405,7 +1405,7 @@ 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      \
@@ -2321,6 +2321,7 @@ 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:

  /*
@@ -3478,7 +3479,7 @@ 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 v4 13/19] memcg: destroy memcg caches
Posted by Glauber Costa on Fri, 12 Oct 2012 13:41:07 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>
CC: Tejun Heo <tj@kernel.org>
---
 include/linux/memcontrol.h | 2 ++
 include/linux/slab.h | 9 +++++++++
 mm/memcontrol.c | 47 +++
 mm/slab.c | 3 +++
 mm/slab.h | 24 ++++++++++++++++++++++++
 mm/slub.c | 7 ++++++-
 6 files changed, 91 insertions(+), 1 deletion(-)

diff --git a/include/linux/memcontrol.h b/include/linux/memcontrol.h
index 4c94182..9ac12cb 100644
--- a/include/linux/memcontrol.h
+++ b/include/linux/memcontrol.h
@@ -426,6 +426,8 @@ void memcg_update_array_size(int num_groups);
 struct kmem_cache *
 __memcg_kmem_get_cache(struct kmem_cache *cachep, gfp_t gfp);

+void mem_cgroup_destroy_cache(struct kmem_cache *cachep);
+
 /**
  * memcg_kmem_newpage_charge: verify if a new kmem allocation is allowed.
  * @gfp: the gfp allocation flags.
diff --git a/include/linux/slab.h b/include/linux/slab.h

```
index b22a158..e17d348 100644
--- a/include/linux/slab.h
+++ b/include/linux/slab.h
@@ -198,6 +198,11 @@ unsigned int kmem_cache_size(struct kmem_cache *);
  *
  * @memcg: pointer to the memcg this cache belongs to
  * @root_cache: pointer to the global, root cache, this cache was derived from
+ * @cachep: backpointer to the kmem_cache structure that hold us.
+ * @dead: set to true after the memcg dies; the cache may still be around.
+ * @nr_pages: number of pages that belongs to this cache.
+ * @destroy: worker to be called whenever we are ready, or believe we may be
+ *           ready, to destroy this cache.
  */
 struct memcg_cache_params {
  bool is_root_cache;
@@ -206,6 +211,10 @@ struct memcg_cache_params {
   struct {
    struct mem_cgroup *memcg;
    struct kmem_cache *root_cache;
+   struct kmem_cache *cachep;
+   bool dead;
+   atomic_t nr_pages;
+   struct work_struct destroy;
   };
  };
 };
diff --git a/mm/memcontrol.c b/mm/memcontrol.c
index 4d2a01f..f744305 100644
--- a/mm/memcontrol.c
+++ b/mm/memcontrol.c
@@ -2949,6 +2949,31 @@ out:
  kfree(s->memcg_params);
 }

+static void kmem_cache_destroy_work_func(struct work_struct *w)
+{
+ struct kmem_cache *cachep;
+ struct memcg_cache_params *p;
+
+ p = container_of(w, struct memcg_cache_params, destroy);
+ cachep = p->cachep;
+
+ if (!atomic_read(&cachep->memcg_params->nr_pages))
+  kmem_cache_destroy(cachep);
+}
+static DECLARE_WORK(kmem_cache_destroy_work, kmem_cache_destroy_work_func);
+
+void mem_cgroup_destroy_cache(struct kmem_cache *cachep)
```

```
+{
+ 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.
+  */
+ schedule_work(&cachep->memcg_params->destroy);
+}
+
 /*
  * During the creation a new cache, we need to disable our accounting mechanism
  * altogether. This is true even if we are not creating, but rather just
@@ -3061,6 +3086,8 @@ static struct kmem_cache *memcg_create_kmem_cache(struct
mem_cgroup *memcg,
  wmb(); /* the readers won't lock, make sure everybody sees it */
  new_cachep->memcg_params->memcg = memcg;
  new_cachep->memcg_params->root_cache = cachep;
+ new_cachep->memcg_params->cachep = new_cachep;
+ atomic_set(&new_cachep->memcg_params->nr_pages , 0);
 out:
  mutex_unlock(&memcg_cache_mutex);
  return new_cachep;
@@ -3072,6 +3099,21 @@ struct create_work {
  struct work_struct work;
 };

+static void mem_cgroup_destroy_all_caches(struct mem_cgroup *memcg)
+{
+ struct kmem_cache *cachep;
+
+ mutex_lock(&memcg->slab_caches_mutex);
+ list_for_each_entry(cachep, &memcg->memcg_slab_caches, list) {
+
+  cachep->memcg_params->dead = true;
+  INIT_WORK(&cachep->memcg_params->destroy,
+      kmem_cache_destroy_work_func);
+  schedule_work(&cachep->memcg_params->destroy);
+ }
+ mutex_unlock(&memcg->slab_caches_mutex);
+}
+
 static void memcg_create_cache_work_func(struct work_struct *w)
 {
  struct create_work *cw;
@@ -3277,6 +3319,10 @@ void __memcg_kmem_uncharge_page(struct page *page, int order)
  VM_BUG_ON(mem_cgroup_is_root(memcg));
```

```
  memcg_uncharge_kmem(memcg, PAGE_SIZE << order);
 }
+#else
+static inline void mem_cgroup_destroy_all_caches(struct mem_cgroup *memcg)
+{
+}
 #endif /* CONFIG_MEMCG_KMEM */

 #ifdef CONFIG_TRANSPARENT_HUGEPAGE
@@ -5872,6 +5918,7 @@ 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 03952c4..39127f6 100644
--- a/mm/slab.c
+++ b/mm/slab.c
@@ -1911,6 +1911,7 @@ static void *kmem_getpages(struct kmem_cache *cachep, gfp_t flags,
int nodeid)
    if (page->pfmemalloc)
     SetPageSlabPfmemalloc(page + i);
  }
+ memcg_bind_pages(cachep, cachep->gfporder);

  if (kmemcheck_enabled && !(cachep->flags & SLAB_NOTRACK)) {
   kmemcheck_alloc_shadow(page, cachep->gfporder, flags, nodeid);
@@ -1947,6 +1948,8 @@ static void kmem_freepages(struct kmem_cache *cachep, void *addr)
   __ClearPageSlab(page);
   page++;
  }
+
+ memcg_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 b9b5f1f..ab57462 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
  */
@@ -109,6 +110,21 @@ static inline bool cache_match_memcg(struct kmem_cache *cachep,
   (cachep->memcg_params->memcg == memcg);
 }

+static inline void memcg_bind_pages(struct kmem_cache *s, int order)
+{
+ if (!is_root_cache(s))
+  atomic_add(1 << order, &s->memcg_params->nr_pages);
+}
+
+static inline void memcg_release_pages(struct kmem_cache *s, int order)
+{
+ if (is_root_cache(s))
+  return;
+
+ if (atomic_sub_and_test((1 << order), &s->memcg_params->nr_pages))
+  mem_cgroup_destroy_cache(s);
+}
+
 static inline bool slab_equal_or_root(struct kmem_cache *s,
       struct kmem_cache *p)
 {
@@ -127,6 +143,14 @@ static inline bool cache_match_memcg(struct kmem_cache *cachep,
  return true;
 }

+static inline void memcg_bind_pages(struct kmem_cache *s, int order)
+{
+}
+
+static inline void memcg_release_pages(struct kmem_cache *s, int order)
+{
+}
+
 static inline bool slab_equal_or_root(struct kmem_cache *s,
       struct kmem_cache *p)
 {
diff --git a/mm/slub.c b/mm/slub.c
index 257e130..e98fdf0 100644
--- a/mm/slub.c
+++ b/mm/slub.c
@@ -1344,6 +1344,7 @@ static struct page *new_slab(struct kmem_cache *s, gfp_t flags, int
node)
  void *start;
  void *last;
  void *p;
```

```
+ int order;

  BUG_ON(flags & GFP_SLAB_BUG_MASK);

@@ -1352,7 +1353,9 @@ 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);
+ memcg_bind_pages(s, order);
  page->slab = s;
  __SetPageSlab(page);
  if (page->pfmemalloc)
@@ -1361,7 +1364,7 @@ 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) {
@@ -1402,6 +1405,8 @@ static void __free_slab(struct kmem_cache *s, struct page *page)

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

Subject: [PATCH v4 14/19] memcg/sl[au]b Track all the memcg children of a kmem_cache.

Posted by Glauber Costa on Fri, 12 Oct 2012 13:41:08 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>
CC: Tejun Heo <tj@kernel.org>
---
 include/linux/memcontrol.h |  6 ++++++
 mm/memcontrol.c            | 32 ++++++++++++++++++++++++++++++--
 mm/slab_common.c           |  3 +++
 3 files changed, 39 insertions(+), 2 deletions(-)

diff --git a/include/linux/memcontrol.h b/include/linux/memcontrol.h
index 9ac12cb..6c8156b 100644
--- a/include/linux/memcontrol.h
+++ b/include/linux/memcontrol.h
@@ -427,6 +427,7 @@ struct kmem_cache *
 __memcg_kmem_get_cache(struct kmem_cache *cachep, gfp_t gfp);

 void mem_cgroup_destroy_cache(struct kmem_cache *cachep);
+void kmem_cache_destroy_memcg_children(struct kmem_cache *s);

 /**
  * memcg_kmem_newpage_charge: verify if a new kmem allocation is allowed.
@@ -578,6 +579,11 @@ memcg_kmem_get_cache(struct kmem_cache *cachep, gfp_t gfp)
 {
  return cachep;
 }
+
+
+static inline void kmem_cache_destroy_memcg_children(struct kmem_cache *s)
+{
+}
 #endif /* CONFIG_MEMCG_KMEM */
 #endif /* _LINUX_MEMCONTROL_H */

diff --git a/mm/memcontrol.c b/mm/memcontrol.c
index f744305..8cf8b4d 100644
--- a/mm/memcontrol.c
+++ b/mm/memcontrol.c
@@ -2716,6 +2716,8 @@ static void __mem_cgroup_commit_charge(struct mem_cgroup
*memcg,
  memcg_check_events(memcg, page);
 }

+static DEFINE_MUTEX(set_limit_mutex);
```

```
+
 #ifdef CONFIG_MEMCG_KMEM
 static inline bool memcg_can_account_kmem(struct mem_cgroup *memcg)
 {
@@ -3093,6 +3095,34 @@ out:
  return new_cachep;
 }

+void kmem_cache_destroy_memcg_children(struct kmem_cache *s)
+{
+ struct kmem_cache *c;
+ int i;
+
+ if (!s->memcg_params)
+  return;
+ if (!s->memcg_params->is_root_cache)
+  return;
+
+ /*
+  * If the cache is being destroyed, we trust that there is no one else
+  * requesting objects from it. Even if there are, the sanity checks in
+  * kmem_cache_destroy should caught this ill-case.
+  *
+  * Still, we don't want anyone else freeing memcg_caches under our
+  * noses, which can happen if a new memcg comes to life. As usual,
+  * we'll take the set_limit_mutex to protect ourselves against this.
+  */
+ mutex_lock(&set_limit_mutex);
+ for (i = 0; i < memcg_limited_groups_array_size; i++) {
+  c = s->memcg_params->memcg_caches[i];
+  if (c)
+   kmem_cache_destroy(c);
+ }
+ mutex_unlock(&set_limit_mutex);
+}
+
 struct create_work {
  struct mem_cgroup *memcg;
  struct kmem_cache *cachep;
@@ -4189,8 +4219,6 @@ void mem_cgroup_print_bad_page(struct page *page)
 }
 #endif

-static DEFINE_MUTEX(set_limit_mutex);
-
 static int mem_cgroup_resize_limit(struct mem_cgroup *memcg,
     unsigned long long val)
 {
```

```
diff --git a/mm/slab_common.c b/mm/slab_common.c
index fcf59d7..c02faf5 100644
--- a/mm/slab_common.c
+++ b/mm/slab_common.c
@@ -214,6 +214,9 @@ EXPORT_SYMBOL(kmem_cache_create);

 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 v4 15/19] memcg/sl[au]b: shrink dead caches
Posted by Glauber Costa on Fri, 12 Oct 2012 13:41:09 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]
[v3: use delayed_work to avoid calling verify_dead at every free]

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>
CC: Tejun Heo <tj@kernel.org>
---
 include/linux/slab.h |  2 +-
 mm/memcontrol.c      | 28 ++++++++++++++++++++++------
 2 files changed, 23 insertions(+), 7 deletions(-)

diff --git a/include/linux/slab.h b/include/linux/slab.h
index e17d348..9ed1a98 100644
--- a/include/linux/slab.h
+++ b/include/linux/slab.h
@@ -214,7 +214,7 @ struct memcg_cache_params {
     struct kmem_cache *cachep;
     bool dead;
     atomic_t nr_pages;
-    struct work_struct destroy;
+    struct delayed_work destroy;
   };
  };
 };
diff --git a/mm/memcontrol.c b/mm/memcontrol.c
index 8cf8b4d..b52e6b9 100644
--- a/mm/memcontrol.c
+++ b/mm/memcontrol.c
@@ -2955,11 +2955,27 @ static void kmem_cache_destroy_work_func(struct work_struct *w)
 {
  struct kmem_cache *cachep;
  struct memcg_cache_params *p;
+ struct delayed_work *dw = to_delayed_work(w);

- p = container_of(w, struct memcg_cache_params, destroy);
+ p = container_of(dw, struct memcg_cache_params, destroy);
  cachep = p->cachep;

- if (!atomic_read(&cachep->memcg_params->nr_pages))
+ /*
+  * If we get down to 0 after shrink, we could delete right away.
+  * However, memcg_release_pages() already puts us back in the workqueue
+  * in that case. If we proceed deleting, we'll get a dangling
+  * reference, and removing the object from the workqueue in that case is
+  * unnecessary complication. We are not a fast path.
+  *
+  * If we aren't down to zero, we'll schedule a slow worker and try again
+  */
```

```
+ if (atomic_read(&cachep->memcg_params->nr_pages) != 0) {
+  kmem_cache_shrink(cachep);
+  if (atomic_read(&cachep->memcg_params->nr_pages) == 0)
+   return;
+  /* Once per minute should be good enough. */
+  schedule_delayed_work(&cachep->memcg_params->destroy, 60 * HZ);
+ } else
   kmem_cache_destroy(cachep);
 }
 static DECLARE_WORK(kmem_cache_destroy_work, kmem_cache_destroy_work_func);
@@ -2973,7 +2989,7 @@ void mem_cgroup_destroy_cache(struct kmem_cache *cachep)
   * We have to defer the actual destroying to a workqueue, because
   * we might currently be in a context that cannot sleep.
   */
- schedule_work(&cachep->memcg_params->destroy);
+ schedule_delayed_work(&cachep->memcg_params->destroy, 0);
 }

 /*
@@ -3137,9 +3153,9 @@ static void mem_cgroup_destroy_all_caches(struct mem_cgroup
*memcg)
  list_for_each_entry(cachep, &memcg->memcg_slab_caches, list) {

   cachep->memcg_params->dead = true;
-  INIT_WORK(&cachep->memcg_params->destroy,
-    kmem_cache_destroy_work_func);
-  schedule_work(&cachep->memcg_params->destroy);
+  INIT_DELAYED_WORK(&cachep->memcg_params->destroy,
+      kmem_cache_destroy_work_func);
+  schedule_delayed_work(&cachep->memcg_params->destroy, 0);
  }
  mutex_unlock(&memcg->slab_caches_mutex);
 }
--
1.7.11.4
```

Subject: [PATCH v4 16/19] Aggregate memcg cache values in slabinfo
Posted by Glauber Costa on Fri, 12 Oct 2012 13:41:10 GMT
View Forum Message <> Reply to Message

When we create caches in memcgs, we need to display their usage
information somewhere. We'll adopt a scheme similar to /proc/meminfo,
with aggregate totals shown in the global file, and per-group
information stored in the group itself.

For the time being, only reads are allowed in the per-group cache.

```
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>
CC: Tejun Heo <tj@kernel.org>
---
 include/linux/memcontrol.h |  8 ++++++++
 include/linux/slab.h       |  3 +++
 mm/memcontrol.c            | 34 ++++++++++++++++++++++++++++++++-
 mm/slab.h                  | 27 +++++++++++++++++++++++++++
 mm/slab_common.c           | 42 ++++++++++++++++++++++++++++++++++++++----
 5 files changed, 109 insertions(+), 5 deletions(-)

diff --git a/include/linux/memcontrol.h b/include/linux/memcontrol.h
index 6c8156b..5dd366e 100644
--- a/include/linux/memcontrol.h
+++ b/include/linux/memcontrol.h
@@ -404,6 +404,11 @@ void sock_update_memcg(struct sock *sk);
 void sock_release_memcg(struct sock *sk);

 extern struct static_key memcg_kmem_enabled_key;
+
+extern int memcg_limited_groups_array_size;
+#define for_each_memcg_cache_index(_idx) \
+ for ((_idx) = 0; i < memcg_limited_groups_array_size; (_idx)++)
+
 static inline bool memcg_kmem_enabled(void)
 {
  return static_key_false(&memcg_kmem_enabled_key);
@@ -539,6 +544,9 @@ static inline void sock_release_memcg(struct sock *sk)
 {
 }

+#define for_each_memcg_cache_index(_idx) \
+ for (; NULL; )
+
 static inline bool memcg_kmem_enabled(void)
 {
  return false;
diff --git a/include/linux/slab.h b/include/linux/slab.h
index 9ed1a98..b7eea06 100644
--- a/include/linux/slab.h
+++ b/include/linux/slab.h
@@ -221,6 +221,9 @@ struct memcg_cache_params {
```

```
 int memcg_update_all_caches(int num_memcgs);

+int cache_show(struct kmem_cache *s, struct seq_file *m);
+void print_slabinfo_header(struct seq_file *m);
+
 /*
  * Common kmalloc functions provided by all allocators
  */
diff --git a/mm/memcontrol.c b/mm/memcontrol.c
index b52e6b9..da39f47 100644
--- a/mm/memcontrol.c
+++ b/mm/memcontrol.c
@@ -552,7 +552,7 @@ static void disarm_sock_keys(struct mem_cgroup *memcg)
 #endif

 #ifdef CONFIG_MEMCG_KMEM
-static int memcg_limited_groups_array_size;
+int memcg_limited_groups_array_size;
 #define MEMCG_CACHES_MIN_SIZE 64
 /*
  * MAX_SIZE should be as large as the number of css_ids. Ideally, we could get
@@ -2725,6 +2725,33 @@ static inline bool memcg_can_account_kmem(struct mem_cgroup
*memcg)
   (memcg->kmem_accounted & KMEM_ACCOUNTED_MASK);
 }

+
+static int mem_cgroup_slabinfo_write(struct cgroup *cont, struct cftype *cft,
+        const char *buffer)
+{
+ return -EIO;
+}
+
+static int mem_cgroup_slabinfo_read(struct cgroup *cont, struct cftype *cft,
+     struct seq_file *m)
+{
+ struct mem_cgroup *memcg = mem_cgroup_from_cont(cont);
+ struct kmem_cache *s;
+
+ if (!memcg_can_account_kmem(memcg))
+  return -EIO;
+
+ print_slabinfo_header(m);
+
+ mutex_lock(&memcg->slab_caches_mutex);
+ list_for_each_entry(s, &memcg->memcg_slab_caches, list)
+  cache_show(s, m);
+
```

```
+ mutex_unlock(&memcg->slab_caches_mutex);
+
+ return 0;
+}
+
 static int memcg_charge_kmem(struct mem_cgroup *memcg, gfp_t gfp, u64 size)
 {
  struct res_counter *fail_res;
@@ -5661,6 +5688,11 @@ static struct cftype mem_cgroup_files[] = {
   .trigger = mem_cgroup_reset,
   .read = mem_cgroup_read,
  },
+ {
+  .name = "kmem.slabinfo",
+  .write_string = mem_cgroup_slabinfo_write,
+  .read_seq_string = mem_cgroup_slabinfo_read,
+ },
 #endif
  { }, /* terminate */
 };
diff --git a/mm/slab.h b/mm/slab.h
index ab57462..c60f649 100644
--- a/mm/slab.h
+++ b/mm/slab.h
@@ -131,6 +131,23 @@ static inline bool slab_equal_or_root(struct kmem_cache *s,
  return (p == s) ||
   (s->memcg_params && (p == s->memcg_params->root_cache));
 }
+
+/*
+ * We use suffixes to the name in memcg because we can't have caches
+ * created in the system with the same name. But when we print them
+ * locally, better refer to them with the base name
+ */
+static inline const char *cache_name(struct kmem_cache *s)
+{
+ if (!is_root_cache(s))
+  return s->memcg_params->root_cache->name;
+ return s->name;
+}
+
+static inline struct kmem_cache *cache_from_memcg(struct kmem_cache *s, int idx)
+{
+ return s->memcg_params->memcg_caches[idx];
+}
 #else
 static inline bool is_root_cache(struct kmem_cache *s)
 {
```

```
@@ -156,5 +173,15 @@ static inline bool slab_equal_or_root(struct kmem_cache *s,
 {
  return true;
 }
+
+static inline const char *cache_name(struct kmem_cache *s)
+{
+ return s->name;
+}
+
+static inline struct kmem_cache *cache_from_memcg(struct kmem_cache *s, int idx)
+{
+ return NULL;
+}
 #endif
 #endif
diff --git a/mm/slab_common.c b/mm/slab_common.c
index c02faf5..7e6d503 100644
--- a/mm/slab_common.c
+++ b/mm/slab_common.c
@@ -247,7 +247,7 @@ int slab_is_available(void)
 }

 #ifdef CONFIG_SLABINFO
-static void print_slabinfo_header(struct seq_file *m)
+void print_slabinfo_header(struct seq_file *m)
 {
  /*
   * Output format version, so at least we can change it
@@ -291,16 +291,43 @@ static void s_stop(struct seq_file *m, void *p)
  mutex_unlock(&slab_mutex);
 }

-static int s_show(struct seq_file *m, void *p)
+static void
+memcg_accumulate_slabinfo(struct kmem_cache *s, struct slabinfo *info)
+{
+ struct kmem_cache *c;
+ struct slabinfo sinfo;
+ int i;
+
+ if (!is_root_cache(s))
+  return;
+
+ for_each_memcg_cache_index(i) {
+  c = cache_from_memcg(s, i);
+  if (!c)
+   continue;
```

```
+
+  memset(&sinfo, 0, sizeof(sinfo));
+  get_slabinfo(c, &sinfo);
+
+  info->active_slabs += sinfo.active_slabs;
+  info->num_slabs += sinfo.num_slabs;
+  info->shared_avail += sinfo.shared_avail;
+  info->active_objs += sinfo.active_objs;
+  info->num_objs += sinfo.num_objs;
+ }
+}
+
+int cache_show(struct kmem_cache *s, struct seq_file *m)
 {
- struct kmem_cache *s = list_entry(p, struct kmem_cache, list);
  struct slabinfo sinfo;

  memset(&sinfo, 0, sizeof(sinfo));
  get_slabinfo(s, &sinfo);

+ memcg_accumulate_slabinfo(s, &sinfo);
+
  seq_printf(m, "%-17s %6lu %6lu %6u %4u %4d",
-     s->name, sinfo.active_objs, sinfo.num_objs, s->size,
+     cache_name(s), sinfo.active_objs, sinfo.num_objs, s->size,
      sinfo.objects_per_slab, (1 << sinfo.cache_order));

  seq_printf(m, " : tunables %4u %4u %4u",
@@ -312,6 +339,13 @@ static int s_show(struct seq_file *m, void *p)
  return 0;
 }

+static int s_show(struct seq_file *m, void *p)
+{
+ struct kmem_cache *s = list_entry(p, struct kmem_cache, list);
+
+ return cache_show(s, m);
+}
+
 /*
  * slabinfo_op - iterator that generates /proc/slabinfo
  *
--
1.7.11.4
```

Subject: [PATCH v4 17/19] slab: propagate tunables values

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.

It can also happen that a root cache has its tunables updated during
normal system operation. In this case, we will propagate the change to
all caches that are already active.

This change will require us to move the assignment of root_cache in
memcg_params a bit earlier. We need this to be already set - which
memcg_kmem_register_cache will do - when we reach __kmem_cache_create()

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>
CC: Tejun Heo <tj@kernel.org>
---
 include/linux/memcontrol.h | 8 +++++---
 include/linux/slab.h | 2 +-
 mm/memcontrol.c | 10 ++++++----
 mm/slab.c | 44 ++---
 mm/slab.h | 12 ++++++++++++
 mm/slab_common.c | 7 +++++---
 6 files changed, 69 insertions(+), 14 deletions(-)

diff --git a/include/linux/memcontrol.h b/include/linux/memcontrol.h
index 5dd366e..09c2917 100644
--- a/include/linux/memcontrol.h
+++ b/include/linux/memcontrol.h
@@ -421,7 +421,8 @@ void __memcg_kmem_commit_charge(struct page *page,
 void __memcg_kmem_uncharge_page(struct page *page, int order);

 int memcg_css_id(struct mem_cgroup *memcg);

```
-int memcg_register_cache(struct mem_cgroup *memcg, struct kmem_cache *s);
+int memcg_register_cache(struct mem_cgroup *memcg, struct kmem_cache *s,
+    struct kmem_cache *root_cache);
 void memcg_release_cache(struct kmem_cache *cachep);
 void memcg_cache_list_add(struct mem_cgroup *memcg, struct kmem_cache *cachep);

@@ -566,8 +567,9 @@ memcg_kmem_commit_charge(struct page *page, struct mem_cgroup
*memcg, int order)
 {
 }

-static inline int memcg_register_cache(struct mem_cgroup *memcg,
-           struct kmem_cache *s)
+static inline int
+memcg_register_cache(struct mem_cgroup *memcg, struct kmem_cache *s,
+       struct kmem_cache *root_cache)
 {
  return 0;
 }
diff --git a/include/linux/slab.h b/include/linux/slab.h
index b7eea06..f65c6c4 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 *);
diff --git a/mm/memcontrol.c b/mm/memcontrol.c
index da39f47..5eedacc 100644
--- a/mm/memcontrol.c
+++ b/mm/memcontrol.c
@@ -2944,7 +2944,8 @@ int memcg_update_cache_size(struct kmem_cache *s, int
num_groups)
  return 0;
 }

-int memcg_register_cache(struct mem_cgroup *memcg, struct kmem_cache *s)
+int memcg_register_cache(struct mem_cgroup *memcg, struct kmem_cache *s,
+    struct kmem_cache *root_cache)
 {
  size_t size = sizeof(struct memcg_cache_params);

@@ -2958,8 +2959,10 @@ int memcg_register_cache(struct mem_cgroup *memcg, struct
```

```
kmem_cache *s)
  if (!s->memcg_params)
   return -ENOMEM;

- if (memcg)
+ if (memcg) {
   s->memcg_params->memcg = memcg;
+  s->memcg_params->root_cache = root_cache;
+ }
  return 0;
 }

@@ -3082,7 +3085,7 @@ 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);

  if (new)
   new->allocflags |= __GFP_KMEMCG;
@@ -3130,7 +3133,6 @@ static struct kmem_cache *memcg_create_kmem_cache(struct
mem_cgroup *memcg,
  cachep->memcg_params->memcg_caches[idx] = new_cachep;
  wmb(); /* the readers won't lock, make sure everybody sees it */
  new_cachep->memcg_params->memcg = memcg;
- new_cachep->memcg_params->root_cache = cachep;
  new_cachep->memcg_params->cachep = new_cachep;
  atomic_set(&new_cachep->memcg_params->nr_pages , 0);
 out:
diff --git a/mm/slab.c b/mm/slab.c
index 39127f6..42278d3 100644
--- a/mm/slab.c
+++ b/mm/slab.c
@@ -4089,7 +4089,7 @@ 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;
@@ -4132,12 +4132,48 @@ 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;
+ struct kmem_cache *c = NULL;
+ int i = 0;
+
+ ret = __do_tune_cpucache(cachep, limit, batchcount, shared, gfp);
+
+ if (slab_state < FULL)
+  return ret;
+
+ if ((ret < 0) || !is_root_cache(cachep))
+  return ret;
+
+ for_each_memcg_cache_index(i) {
+  c = cache_from_memcg(cachep, i);
+  if (c)
+   /* return value determined by the parent cache only */
+   __do_tune_cpucache(c, limit, batchcount, shared, gfp);
+ }
+
+ return ret;
+}
+
 /* Called with slab_mutex held always */
 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;
+
+ if (!is_root_cache(cachep)) {
+  struct kmem_cache *root = memcg_root_cache(cachep);
+  limit = root->limit;
+  shared = root->shared;
+  batchcount = root->batchcount;
+ }

+ 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
@@ -4179,7 +4215,9 @@ 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.h b/mm/slab.h
index c60f649..bf599fa 100644
--- a/mm/slab.h
+++ b/mm/slab.h
@@ -148,6 +148,13 @@ static inline struct kmem_cache *cache_from_memcg(struct
kmem_cache *s, int idx)
 {
  return s->memcg_params->memcg_caches[idx];
 }
+
+static inline struct kmem_cache *memcg_root_cache(struct kmem_cache *s)
+{
+ if (is_root_cache(s))
+  return s;
+ return s->memcg_params->root_cache;
+}
 #else
 static inline bool is_root_cache(struct kmem_cache *s)
 {
@@ -183,5 +190,10 @@ static inline struct kmem_cache *cache_from_memcg(struct
kmem_cache *s, int idx)
 {
  return NULL;
 }
+
+static inline struct kmem_cache *memcg_root_cache(struct kmem_cache *s)
+{
+ return s;
+}
 #endif
 #endif
diff --git a/mm/slab_common.c b/mm/slab_common.c
index 7e6d503..0dc0c41 100644
--- a/mm/slab_common.c
+++ b/mm/slab_common.c
@@ -127,7 +127,8 @@ out:

 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;
@@ -156,7 +157,7 @@ kmem_cache_create_memcg(struct mem_cgroup *memcg, const char
*name, size_t size,
   s->align = align;
   s->ctor = ctor;

-  if (memcg_register_cache(memcg, s)) {
+  if (memcg_register_cache(memcg, s, parent_cache)) {
    kmem_cache_free(kmem_cache, s);
    err = -ENOMEM;
    goto out_locked;
@@ -208,7 +209,7 @@ 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 v4 18/19] slub: slub-specific propagation changes.
Posted by Glauber Costa on Fri, 12 Oct 2012 13:41:12 GMT
View Forum Message <> Reply to Message

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

This can be done by tapping into the store attribute function provided
by the allocator. We of course don't need to mess with read-only
fields. Since the attributes can have multiple types and are stored
internally by sysfs, the best strategy is to issue a ->show() in the
root cache, and then ->store() in the memcg cache.

The drawback of that, is that sysfs can allocate up to a page in
buffering for show(), that we are likely not to need, but also can't
guarantee. To avoid always allocating a page for that, we can update the
caches at store time with the maximum attribute size ever stored to the
root cache. We will then get a buffer big enough to hold it. The

corolary to this, is that if no stores happened, nothing will be
propagated.

It can also happen that a root cache has its tunables updated during
normal system operation. In this case, we will propagate the change to
all caches that are already active.

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>
CC: Tejun Heo <tj@kernel.org>
---
 include/linux/slub_def.h | 1 +
 mm/slub.c | 71 ++
 2 files changed, 72 insertions(+)

diff --git a/include/linux/slub_def.h b/include/linux/slub_def.h
index ed330df..f41acb9 100644
--- a/include/linux/slub_def.h
+++ b/include/linux/slub_def.h
@@ -105,6 +105,7 @@ struct kmem_cache {
 #endif
 #ifdef CONFIG_MEMCG_KMEM
  struct memcg_cache_params *memcg_params;
+ int max_attr_size; /* for propagation, maximum size of a stored attr */
 #endif

 #ifdef CONFIG_NUMA
diff --git a/mm/slub.c b/mm/slub.c
index e98fdf0..617d7ad 100644
--- a/mm/slub.c
+++ b/mm/slub.c
@@ -3965,6 +3965,8 @@ __kmem_cache_alias(struct mem_cgroup *memcg, const char *name,
size_t size,
  return s;
 }

+static void memcg_propagate_slab_attrs(struct kmem_cache *s);
+
 int __kmem_cache_create(struct kmem_cache *s, unsigned long flags)
 {
  int err;
@@ -3973,6 +3975,7 @@ int __kmem_cache_create(struct kmem_cache *s, unsigned long
flags)

```
  if (err)
   return err;

+ memcg_propagate_slab_attrs(s);
  mutex_unlock(&slab_mutex);
  err = sysfs_slab_add(s);
  mutex_lock(&slab_mutex);
@@ -5198,6 +5201,7 @@ static ssize_t slab_attr_store(struct kobject *kobj,
  struct slab_attribute *attribute;
  struct kmem_cache *s;
  int err;
+ int i __maybe_unused;

  attribute = to_slab_attr(attr);
  s = to_slab(kobj);
@@ -5206,10 +5210,77 @@ 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) || !is_root_cache(s))
+  return err;
+
+ mutex_lock(&slab_mutex);
+ if (s->max_attr_size < len)
+  s->max_attr_size = len;
+
+ for_each_memcg_cache_index(i) {
+  struct kmem_cache *c = cache_from_memcg(s, i);
+  if (c)
+   /* return value determined by the parent cache only */
+   attribute->store(c, buf, len);
+ }
+ mutex_unlock(&slab_mutex);
+#endif
  return err;
 }

+static void memcg_propagate_slab_attrs(struct kmem_cache *s)
+{
+#ifdef CONFIG_MEMCG_KMEM
+ int i;
+ char *buffer = NULL;
+
+ if (!is_root_cache(s))
```

```
+  return;
+
+ if (!s->max_attr_size)
+  return;
+
+ for (i = 0; i < ARRAY_SIZE(slab_attrs); i++) {
+  char mbuf[64];
+  char *buf;
+  struct slab_attribute *attr = to_slab_attr(slab_attrs[i]);
+
+  if (!attr || !attr->store || !attr->show)
+   continue;
+
+  /*
+   * It is really bad that we have to allocate here, so we will
+   * do it only as a fallback. If we actually allocate, though,
+   * we can just use the allocated buffer until the end.
+   *
+   * Most of the slub attributes will tend to be very small in
+   * size, but sysfs allows buffers up to a page, so they can
+   * theoretically happen.
+   */
+  if (buffer)
+   buf = buffer;
+  else if (s->max_attr_size < ARRAY_SIZE(mbuf))
+   buf = mbuf;
+  else {
+   buffer = (char *) get_zeroed_page(GFP_KERNEL);
+   if (WARN_ON(!buffer))
+    continue;
+   buf = buffer;
+  }
+
+  attr->show(s->memcg_params->root_cache, buf);
+  attr->store(s, buf, strlen(buf));
+ }
+
+ if (buffer)
+  free_page((unsigned long)buffer);
+#endif
+}
+
 static const struct sysfs_ops slab_sysfs_ops = {
  .show = slab_attr_show,
  .store = slab_attr_store,
--
1.7.11.4
```

Subject: [PATCH v4 19/19] Add slab-specific documentation about the kmem controller
Posted by Glauber Costa on Fri, 12 Oct 2012 13:41:13 GMT
View Forum Message <> Reply to Message

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>
CC: Tejun Heo <tj@kernel.org>

 Documentation/cgroups/memory.txt | 7 +++++++
 1 file changed, 7 insertions(+)

diff --git a/Documentation/cgroups/memory.txt b/Documentation/cgroups/memory.txt
index dd15be8..95e4809 100644
--- a/Documentation/cgroups/memory.txt
+++ b/Documentation/cgroups/memory.txt
@@ -300,6 +300,13 @@ to trigger slab reclaim when those limits are reached.
 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.
--
1.7.11.4

Subject: Re: [PATCH v4 00/19] slab accounting for memcg
Posted by Glauber Costa on Thu, 18 Oct 2012 11:38:44 GMT
View Forum Message <> Reply to Message

On 10/12/2012 05:40 PM, 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
>

To all reviewers of my previous series (and others as well), I'd like to draw attention to this one.

This is the follow up to the kmemcg-stack series, to be applied right ontop. I believe the first series (kmemcg-stack) got more attention (which is good), but this one got a bunch of fixes and reviews in the process as well.

Thanks
