Subject: [PATCH v5 00/14] kmem controller for memcg.
Posted by Glauber Costa on Tue, 16 Oct 2012 10:16:37 GMT
View Forum Message <> Reply to Message

Hi,

This is the first part of the kernel memory controller for memcg. It has been discussed many times, and I consider this stable enough to be on tree. A follow up to this series are the patches to also track slab memory. They are not included here because I believe we could benefit from merging them separately for better testing coverage. If there are any issues preventing this to be merged, let me know. I'll be happy to address them.

*v5: - changed charged order, kmem charged first.
- minor nits and comments merged.

*v4: - kmem_accounted can no longer become unlimited
- kmem_accounted can no longer become limited, if group has children.
- documentation moved to this patchset
- more style changes
- css_get in charge path to ensure task won't move during charge
*v3:
- Changed function names to match memcg's
- avoid doing get/put in charge/uncharge path
- revert back to keeping the account enabled after it is first activated

Numbers can be found at https://lkml.org/lkml/2012/9/13/239

A (throwaway) git tree with them is placed at:

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

A general explanation of what this is all about follows:

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

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

One example of such problematic pressure that can be prevented by this work is a fork bomb conducted in a shell. We prevent it by noting that processes use a limited amount of stack pages. Seen this way, a fork bomb is just a special

case of resource abuse. If the offender is unable to grab more pages for the stack, no new processes can be created.

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

In fact, a simple:

while true; do mkdir x; cd x; done

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

An important use case for all that, is concerned with people offering hosting services through containers. In a physical box we can put a limit to some resources, like total number of processes or threads. But in an environment where each independent user gets its own piece of the machine, we don't want a potentially malicious user to destroy good users' services.

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

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

Glauber Costa (12):
  memcg: change defines to an enum
  kmem accounting basic infrastructure
  Add a __GFP_KMEMCG flag
  memcg: kmem controller infrastructure
  mm: Allocate kernel pages to the right memcg
  res_counter: return amount of charges after res_counter_uncharge
  memcg: kmem accounting lifecycle management
  memcg: use static branches when code not in use
  memcg: allow a memcg with kmem charges to be destructed.
  execute the whole memcg freeing in free_worker
  protect architectures where THREAD_SIZE >= PAGE_SIZE against fork

bombs
Add documentation about the kmem controller

Suleiman Souhlal (2):
memcg: Make it possible to use the stock for more than one page.
memcg: Reclaim when more than one page needed.

```
Documentation/cgroups/memory.txt          |  58 ++-
Documentation/cgroups/resource_counter.txt |   7 +-
include/linux/gfp.h                       |   6 +-
include/linux/memcontrol.h                | 100 ++++++
include/linux/res_counter.h               |  12 +-
include/linux/thread_info.h               |   2 +
include/trace/events/gfpflags.h           |   1 +
kernel/fork.c                             |   4 +-
kernel/res_counter.c                      |  20 +-
mm/memcontrol.c                           | 553 ++++++++++++++++++++++++++----
mm/page_alloc.c                           |  35 ++
11 files changed, 717 insertions(+), 81 deletions(-)
```

--
1.7.11.7

Subject: [PATCH v5 01/14] memcg: Make it possible to use the stock for more than one page.
Posted by Glauber Costa on Tue, 16 Oct 2012 10:16:38 GMT
View Forum Message <> Reply to Message

From: Suleiman Souhlal <ssouhlal@FreeBSD.org>

We currently have a percpu stock cache scheme that charges one page at a time from memcg->res, the user counter. When the kernel memory controller comes into play, we'll need to charge more than that.

This is because kernel memory allocations will also draw from the user counter, and can be bigger than a single page, as it is the case with the stack (usually 2 pages) or some higher order slabs.

[glommer@parallels.com: added a changelog]

Signed-off-by: Suleiman Souhlal <suleiman@google.com>
Signed-off-by: Glauber Costa <glommer@parallels.com>
Acked-by: David Rientjes <rientjes@google.com>
Acked-by: Kamezawa Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com>
Acked-by: Michal Hocko <mhocko@suse.cz>
Acked-by: Johannes Weiner <hannes@cmpxchg.org>
CC: Tejun Heo <tj@kernel.org>

```
---
 mm/memcontrol.c | 28 ++++++++++++++++++++----------
 1 file changed, 18 insertions(+), 10 deletions(-)

diff --git a/mm/memcontrol.c b/mm/memcontrol.c
index 7acf43b..47cb019 100644
--- a/mm/memcontrol.c
+++ b/mm/memcontrol.c
@@ -2028,20 +2028,28 @@ struct memcg_stock_pcp {
 static DEFINE_PER_CPU(struct memcg_stock_pcp, memcg_stock);
 static DEFINE_MUTEX(percpu_charge_mutex);

-/*
- * Try to consume stocked charge on this cpu. If success, one page is consumed
- * from local stock and true is returned. If the stock is 0 or charges from a
- * cgroup which is not current target, returns false. This stock will be
- * refilled.
+/**
+ * consume_stock: Try to consume stocked charge on this cpu.
+ * @memcg: memcg to consume from.
+ * @nr_pages: how many pages to charge.
+ *
+ * The charges will only happen if @memcg matches the current cpu's memcg
+ * stock, and at least @nr_pages are available in that stock.  Failure to
+ * service an allocation will refill the stock.
+ *
+ * returns true if succesfull, false otherwise.
  */
-static bool consume_stock(struct mem_cgroup *memcg)
+static bool consume_stock(struct mem_cgroup *memcg, int nr_pages)
 {
  struct memcg_stock_pcp *stock;
  bool ret = true;

+ if (nr_pages > CHARGE_BATCH)
+  return false;
+
  stock = &get_cpu_var(memcg_stock);
- if (memcg == stock->cached && stock->nr_pages)
-  stock->nr_pages--;
+ if (memcg == stock->cached && stock->nr_pages >= nr_pages)
+  stock->nr_pages -= nr_pages;
  else /* need to call res_counter_charge */
   ret = false;
  put_cpu_var(memcg_stock);
@@ -2340,7 +2348,7 @@ again:
   VM_BUG_ON(css_is_removed(&memcg->css));
   if (mem_cgroup_is_root(memcg))
```

```
    goto done;
-  if (nr_pages == 1 && consume_stock(memcg))
+  if (consume_stock(memcg, nr_pages))
    goto done;
   css_get(&memcg->css);
  } else {
@@ -2365,7 +2373,7 @@ again:
    rcu_read_unlock();
    goto done;
   }
-  if (nr_pages == 1 && consume_stock(memcg)) {
+  if (consume_stock(memcg, nr_pages)) {
    /*
     * It seems dagerous to access memcg without css_get().
     * But considering how consume_stok works, it's not
--
1.7.11.7
```

Subject: [PATCH v5 03/14] memcg: change defines to an enum
Posted by Glauber Costa on Tue, 16 Oct 2012 10:16:40 GMT
View Forum Message <> Reply to Message

This is just a cleanup patch for clarity of expression. In earlier
submissions, people asked it to be in a separate patch, so here it is.

[v2: use named enum as type throughout the file as well]

Signed-off-by: Glauber Costa <glommer@parallels.com>
Acked-by: Kamezawa Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com>
Acked-by: Michal Hocko <mhocko@suse.cz>
Acked-by: Johannes Weiner <hannes@cmpxchg.org>
CC: Tejun Heo <tj@kernel.org>
---
 mm/memcontrol.c | 26 ++++++++++++++++++----------
 1 file changed, 16 insertions(+), 10 deletions(-)

```
diff --git a/mm/memcontrol.c b/mm/memcontrol.c
index 7a9652a..71d259e 100644
--- a/mm/memcontrol.c
+++ b/mm/memcontrol.c
@@ -386,9 +386,12 @@ enum charge_type {
 };

 /* for encoding cft->private value on file */
-#define _MEM   (0)
-#define _MEMSWAP  (1)
-#define _OOM_TYPE  (2)
```

```
+enum res_type {
+ _MEM,
+ _MEMSWAP,
+ _OOM_TYPE,
+};
+
 #define MEMFILE_PRIVATE(x, val) ((x) << 16 | (val))
 #define MEMFILE_TYPE(val) ((val) >> 16 & 0xffff)
 #define MEMFILE_ATTR(val) ((val) & 0xffff)
@@ -3915,7 +3918,8 @@ static ssize_t mem_cgroup_read(struct cgroup *cont, struct cftype *cft,
  struct mem_cgroup *memcg = mem_cgroup_from_cont(cont);
  char str[64];
  u64 val;
- int type, name, len;
+ int name, len;
+ enum res_type type;

  type = MEMFILE_TYPE(cft->private);
  name = MEMFILE_ATTR(cft->private);
@@ -3951,7 +3955,8 @@ static int mem_cgroup_write(struct cgroup *cont, struct cftype *cft,
       const char *buffer)
 {
  struct mem_cgroup *memcg = mem_cgroup_from_cont(cont);
- int type, name;
+ enum res_type type;
+ int name;
  unsigned long long val;
  int ret;

@@ -4027,7 +4032,8 @@ out:
 static int mem_cgroup_reset(struct cgroup *cont, unsigned int event)
 {
  struct mem_cgroup *memcg = mem_cgroup_from_cont(cont);
- int type, name;
+ int name;
+ enum res_type type;

  type = MEMFILE_TYPE(event);
  name = MEMFILE_ATTR(event);
@@ -4363,7 +4369,7 @@ static int mem_cgroup_usage_register_event(struct cgroup *cgrp,
  struct mem_cgroup *memcg = mem_cgroup_from_cont(cgrp);
  struct mem_cgroup_thresholds *thresholds;
  struct mem_cgroup_threshold_ary *new;
- int type = MEMFILE_TYPE(cft->private);
+ enum res_type type = MEMFILE_TYPE(cft->private);
  u64 threshold, usage;
  int i, size, ret;
```

```
@@ -4446,7 +4452,7 @@ static void mem_cgroup_usage_unregister_event(struct cgroup *cgrp,
  struct mem_cgroup *memcg = mem_cgroup_from_cont(cgrp);
  struct mem_cgroup_thresholds *thresholds;
  struct mem_cgroup_threshold_ary *new;
- int type = MEMFILE_TYPE(cft->private);
+ enum res_type type = MEMFILE_TYPE(cft->private);
  u64 usage;
  int i, j, size;

@@ -4524,7 +4530,7 @@ static int mem_cgroup_oom_register_event(struct cgroup *cgrp,
 {
  struct mem_cgroup *memcg = mem_cgroup_from_cont(cgrp);
  struct mem_cgroup_eventfd_list *event;
- int type = MEMFILE_TYPE(cft->private);
+ enum res_type type = MEMFILE_TYPE(cft->private);

  BUG_ON(type != _OOM_TYPE);
  event = kmalloc(sizeof(*event), GFP_KERNEL);
@@ -4549,7 +4555,7 @@ static void mem_cgroup_oom_unregister_event(struct cgroup *cgrp,
 {
  struct mem_cgroup *memcg = mem_cgroup_from_cont(cgrp);
  struct mem_cgroup_eventfd_list *ev, *tmp;
- int type = MEMFILE_TYPE(cft->private);
+ enum res_type type = MEMFILE_TYPE(cft->private);

  BUG_ON(type != _OOM_TYPE);

--
1.7.11.7
```

Subject: [PATCH v5 04/14] kmem accounting basic infrastructure
Posted by Glauber Costa on Tue, 16 Oct 2012 10:16:41 GMT
View Forum Message <> Reply to Message

This patch adds the basic infrastructure for the accounting of kernel
memory. To control that, the following files are created:

 * memory.kmem.usage_in_bytes
 * memory.kmem.limit_in_bytes
 * memory.kmem.failcnt
 * memory.kmem.max_usage_in_bytes

They have the same meaning of their user memory counterparts. They
reflect the state of the "kmem" res_counter.

Per cgroup kmem memory accounting is not enabled until a limit is set
for the group. Once the limit is set the accounting cannot be disabled

for that group. This means that after the patch is applied, no
behavioral changes exists for whoever is still using memcg to control
their memory usage, until memory.kmem.limit_in_bytes is set for the
first time.

We always account to both user and kernel resource_counters. This
effectively means that an independent kernel limit is in place when the
limit is set to a lower value than the user memory. A equal or higher
value means that the user limit will always hit first, meaning that kmem
is effectively unlimited.

People who want to track kernel memory but not limit it, can set this
limit to a very high number (like RESOURCE_MAX - 1page - that no one
will ever hit, or equal to the user memory)

[v4: make kmem files part of the main array;
      do not allow limit to be set for non-empty cgroups]
[v5: cosmetic changes]

Signed-off-by: Glauber Costa <glommer@parallels.com>
Acked-by: Kamezawa Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com>
CC: Michal Hocko <mhocko@suse.cz>
CC: Johannes Weiner <hannes@cmpxchg.org>
CC: Tejun Heo <tj@kernel.org>
---
 mm/memcontrol.c | 116 ++-
 1 file changed, 115 insertions(+), 1 deletion(-)

diff --git a/mm/memcontrol.c b/mm/memcontrol.c
index 71d259e..30eafeb 100644
--- a/mm/memcontrol.c
+++ b/mm/memcontrol.c
@@ -266,6 +266,10 @@ struct mem_cgroup {
 };

 /*
+ * the counter to account for kernel memory usage.
+ */
+ struct res_counter kmem;
+ /*
  * Per cgroup active and inactive list, similar to the
  * per zone LRU lists.
  */
@@ -280,6 +284,7 @@ struct mem_cgroup {
  * Should the accounting and control be hierarchical, per subtree?
  */
 bool use_hierarchy;
+ unsigned long kmem_accounted; /* See KMEM_ACCOUNTED_*, below */

```
  bool  oom_lock;
  atomic_t under_oom;
@@ -332,6 +337,20 @@ struct mem_cgroup {
 #endif
 };

+/* internal only representation about the status of kmem accounting. */
+enum {
+ KMEM_ACCOUNTED_ACTIVE = 0, /* accounted by this cgroup itself */
+};
+
+#define KMEM_ACCOUNTED_MASK (1 << KMEM_ACCOUNTED_ACTIVE)
+
+#ifdef CONFIG_MEMCG_KMEM
+static void memcg_kmem_set_active(struct mem_cgroup *memcg)
+{
+ set_bit(KMEM_ACCOUNTED_ACTIVE, &memcg->kmem_accounted);
+}
+#endif
+
 /* Stuffs for move charges at task migration. */
 /*
  * Types of charges to be moved. "move_charge_at_immitgrate" is treated as a
@@ -390,6 +409,7 @@ enum res_type {
  _MEM,
  _MEMSWAP,
  _OOM_TYPE,
+ _KMEM,
 };

 #define MEMFILE_PRIVATE(x, val) ((x) << 16 | (val))
@@ -1433,6 +1453,10 @@ done:
   res_counter_read_u64(&memcg->memsw, RES_USAGE) >> 10,
   res_counter_read_u64(&memcg->memsw, RES_LIMIT) >> 10,
   res_counter_read_u64(&memcg->memsw, RES_FAILCNT));
+ printk(KERN_INFO "kmem: usage %llukB, limit %llukB, failcnt %llu\n",
+  res_counter_read_u64(&memcg->kmem, RES_USAGE) >> 10,
+  res_counter_read_u64(&memcg->kmem, RES_LIMIT) >> 10,
+  res_counter_read_u64(&memcg->kmem, RES_FAILCNT));
 }

 /*
@@ -3940,6 +3964,9 @@ static ssize_t mem_cgroup_read(struct cgroup *cont, struct cftype *cft,
   else
    val = res_counter_read_u64(&memcg->memsw, name);
   break;
+ case _KMEM:
```

```
+  val = res_counter_read_u64(&memcg->kmem, name);
+  break;
   default:
    BUG();
   }
@@ -3947,6 +3974,57 @@ static ssize_t mem_cgroup_read(struct cgroup *cont, struct cftype
*cft,
   len = scnprintf(str, sizeof(str), "%llu\n", (unsigned long long)val);
   return simple_read_from_buffer(buf, nbytes, ppos, str, len);
  }
+
+static int memcg_update_kmem_limit(struct cgroup *cont, u64 val)
+{
+ int ret = -EINVAL;
+#ifdef CONFIG_MEMCG_KMEM
+ struct mem_cgroup *memcg = mem_cgroup_from_cont(cont);
+ /*
+  * For simplicity, we won't allow this to be disabled.  It also can't
+  * be changed if the cgroup has children already, or if tasks had
+  * already joined.
+  *
+  * If tasks join before we set the limit, a person looking at
+  * kmem.usage_in_bytes will have no way to determine when it took
+  * place, which makes the value quite meaningless.
+  *
+  * After it first became limited, changes in the value of the limit are
+  * of course permitted.
+  *
+  * Taking the cgroup_lock is really offensive, but it is so far the only
+  * way to guarantee that no children will appear. There are plenty of
+  * other offenders, and they should all go away. Fine grained locking
+  * is probably the way to go here. When we are fully hierarchical, we
+  * can also get rid of the use_hierarchy check.
+  */
+ cgroup_lock();
+ mutex_lock(&set_limit_mutex);
+ if (!memcg->kmem_accounted && val != RESOURCE_MAX) {
+  if (cgroup_task_count(cont) || (memcg->use_hierarchy &&
+      !list_empty(&cont->children))) {
+   ret = -EBUSY;
+   goto out;
+  }
+  ret = res_counter_set_limit(&memcg->kmem, val);
+  VM_BUG_ON(ret);
+
+  memcg_kmem_set_active(memcg);
+ } else
+  ret = res_counter_set_limit(&memcg->kmem, val);
```

```
+out:
+ mutex_unlock(&set_limit_mutex);
+ cgroup_unlock();
+#endif
+ return ret;
+}
+
+static void memcg_propagate_kmem(struct mem_cgroup *memcg,
+     struct mem_cgroup *parent)
+{
+ memcg->kmem_accounted = parent->kmem_accounted;
+}
+
 /*
  * The user of this function is...
  * RES_LIMIT.
@@ -3978,8 +4056,12 @@ static int mem_cgroup_write(struct cgroup *cont, struct cftype *cft,
    break;
   if (type == _MEM)
    ret = mem_cgroup_resize_limit(memcg, val);
-  else
+  else if (type == _MEMSWAP)
    ret = mem_cgroup_resize_memsw_limit(memcg, val);
+  else if (type == _KMEM)
+   ret = memcg_update_kmem_limit(cont, val);
+  else
+   return -EINVAL;
   break;
  case RES_SOFT_LIMIT:
   ret = res_counter_memparse_write_strategy(buffer, &val);
@@ -4045,12 +4127,16 @@ static int mem_cgroup_reset(struct cgroup *cont, unsigned int
event)
  case RES_MAX_USAGE:
   if (type == _MEM)
    res_counter_reset_max(&memcg->res);
+  else if (type == _KMEM)
+   res_counter_reset_max(&memcg->kmem);
   else
    res_counter_reset_max(&memcg->memsw);
   break;
  case RES_FAILCNT:
   if (type == _MEM)
    res_counter_reset_failcnt(&memcg->res);
+  else if (type == _KMEM)
+   res_counter_reset_failcnt(&memcg->kmem);
   else
    res_counter_reset_failcnt(&memcg->memsw);
   break;
```

```
@@ -4728,6 +4814,31 @@ static struct cftype mem_cgroup_files[] = {
   .read = mem_cgroup_read,
  },
 #endif
+#ifdef CONFIG_MEMCG_KMEM
+ {
+  .name = "kmem.limit_in_bytes",
+  .private = MEMFILE_PRIVATE(_KMEM, RES_LIMIT),
+  .write_string = mem_cgroup_write,
+  .read = mem_cgroup_read,
+ },
+ {
+  .name = "kmem.usage_in_bytes",
+  .private = MEMFILE_PRIVATE(_KMEM, RES_USAGE),
+  .read = mem_cgroup_read,
+ },
+ {
+  .name = "kmem.failcnt",
+  .private = MEMFILE_PRIVATE(_KMEM, RES_FAILCNT),
+  .trigger = mem_cgroup_reset,
+  .read = mem_cgroup_read,
+ },
+ {
+  .name = "kmem.max_usage_in_bytes",
+  .private = MEMFILE_PRIVATE(_KMEM, RES_MAX_USAGE),
+  .trigger = mem_cgroup_reset,
+  .read = mem_cgroup_read,
+ },
+#endif
  { }, /* terminate */
 };

@@ -4973,6 +5084,7 @@ mem_cgroup_create(struct cgroup *cont)
  if (parent && parent->use_hierarchy) {
   res_counter_init(&memcg->res, &parent->res);
   res_counter_init(&memcg->memsw, &parent->memsw);
+  res_counter_init(&memcg->kmem, &parent->kmem);
   /*
    * We increment refcnt of the parent to ensure that we can
    * safely access it on res_counter_charge/uncharge.
@@ -4980,9 +5092,11 @@ mem_cgroup_create(struct cgroup *cont)
    * mem_cgroup(see mem_cgroup_put).
    */
   mem_cgroup_get(parent);
+  memcg_propagate_kmem(memcg, parent);
  } else {
   res_counter_init(&memcg->res, NULL);
   res_counter_init(&memcg->memsw, NULL);
```

```
+  res_counter_init(&memcg->kmem, NULL);
   /*
    * Deeper hierachy with use_hierarchy == false doesn't make
    * much sense so let cgroup subsystem know about this
--
1.7.11.7
```

Subject: [PATCH v5 05/14] Add a __GFP_KMEMCG flag
Posted by Glauber Costa on Tue, 16 Oct 2012 10:16:42 GMT
View Forum Message <> Reply to Message

This flag is used to indicate to the callees that this allocation is a
kernel allocation in process context, and should be accounted to
current's memcg. It takes numerical place of the of the recently removed
__GFP_NO_KSWAPD.

[v4: make flag unconditional, also declare it in trace code]

Signed-off-by: Glauber Costa <glommer@parallels.com>
Acked-by: Johannes Weiner <hannes@cmpxchg.org>
Acked-by: Rik van Riel <riel@redhat.com>
Acked-by: Mel Gorman <mel@csn.ul.ie>
Acked-by: Kamezawa Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com>
CC: Christoph Lameter <cl@linux.com>
CC: Pekka Enberg <penberg@cs.helsinki.fi>
CC: Michal Hocko <mhocko@suse.cz>
CC: Suleiman Souhlal <suleiman@google.com>
CC: Tejun Heo <tj@kernel.org>
---
 include/linux/gfp.h | 3 ++-
 include/trace/events/gfpflags.h | 1 +
 2 files changed, 3 insertions(+), 1 deletion(-)

diff --git a/include/linux/gfp.h b/include/linux/gfp.h
index 02c1c97..9289d46 100644
--- a/include/linux/gfp.h
+++ b/include/linux/gfp.h
@@ -31,6 +31,7 @@ struct vm_area_struct;
 #define ___GFP_THISNODE 0x40000u
 #define ___GFP_RECLAIMABLE 0x80000u
 #define ___GFP_NOTRACK 0x200000u
+#define ___GFP_KMEMCG 0x400000u
 #define ___GFP_OTHER_NODE 0x800000u
 #define ___GFP_WRITE 0x1000000u

@@ -87,7 +88,7 @@ struct vm_area_struct;

```
 #define __GFP_OTHER_NODE ((__force gfp_t)___GFP_OTHER_NODE) /* On behalf of other
node */
 #define __GFP_WRITE ((__force gfp_t)___GFP_WRITE) /* Allocator intends to dirty page */
-
+#define __GFP_KMEMCG ((__force gfp_t)___GFP_KMEMCG) /* Allocation comes from a
memcg-accounted resource */
 /*
  * This may seem redundant, but it's a way of annotating false positives vs.
  * allocations that simply cannot be supported (e.g. page tables).
diff --git a/include/trace/events/gfpflags.h b/include/trace/events/gfpflags.h
index 9391706..730df12 100644
--- a/include/trace/events/gfpflags.h
+++ b/include/trace/events/gfpflags.h
@@ -36,6 +36,7 @@
  {(unsigned long)__GFP_RECLAIMABLE, "GFP_RECLAIMABLE"}, \
  {(unsigned long)__GFP_MOVABLE,  "GFP_MOVABLE"},  \
  {(unsigned long)__GFP_NOTRACK,  "GFP_NOTRACK"},  \
+ {(unsigned long)__GFP_KMEMCG,  "GFP_KMEMCG"},  \
  {(unsigned long)__GFP_OTHER_NODE, "GFP_OTHER_NODE"} \
  ) : "GFP_NOWAIT"

--
1.7.11.7
```

Subject: [PATCH v5 06/14] memcg: kmem controller infrastructure
Posted by Glauber Costa on Tue, 16 Oct 2012 10:16:43 GMT
View Forum Message <> Reply to Message

This patch introduces infrastructure for tracking kernel memory pages to
a given memcg. This will happen whenever the caller includes the flag
__GFP_KMEMCG flag, and the task belong to a memcg other than the root.

In memcontrol.h those functions are wrapped in inline acessors. The
idea is to later on, patch those with static branches, so we don't incur
any overhead when no mem cgroups with limited kmem are being used.

Users of this functionality shall interact with the memcg core code
through the following functions:

memcg_kmem_newpage_charge: will return true if the group can handle the
                           allocation. At this point, struct page is not
                           yet allocated.

memcg_kmem_commit_charge: will either revert the charge, if struct page
                          allocation failed, or embed memcg information
                          into page_cgroup.

memcg_kmem_uncharge_page: called at free time, will revert the charge.

[v2: improved comments and standardized function names]
[v3: handle no longer opaque, functions not exported,
  even more comments]
[v4: reworked Used bit handling and surroundings for more clarity]
[v5: simplified code for kmemcg compiled out and core functions in
  memcontrol.c, moved kmem code to the middle to avoid forward decls]

Signed-off-by: Glauber Costa <glommer@parallels.com>
Acked-by: Michal Hocko <mhocko@suse.cz>
CC: Christoph Lameter <cl@linux.com>
CC: Pekka Enberg <penberg@cs.helsinki.fi>
CC: Kamezawa Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com>
CC: Johannes Weiner <hannes@cmpxchg.org>
CC: Tejun Heo <tj@kernel.org>
---
 include/linux/memcontrol.h | 98 ++++++

```
+
+/**
+ * memcg_kmem_newpage_charge: verify if a new kmem allocation is allowed.
+ * @gfp: the gfp allocation flags.
+ * @memcg: a pointer to the memcg this was charged against.
+ * @order: allocation order.
+ *
+ * returns true if the memcg where the current task belongs can hold this
+ * allocation.
+ *
+ * We return true automatically if this allocation is not to be accounted to
+ * any memcg.
+ */
+static __always_inline bool
+memcg_kmem_newpage_charge(gfp_t gfp, struct mem_cgroup **memcg, int order)
+{
+ if (!memcg_kmem_enabled())
+  return true;
+
+ /*
+  * __GFP_NOFAIL allocations will move on even if charging is not
+  * possible. Therefore we don't even try, and have this allocation
+  * unaccounted. We could in theory charge it with
+  * res_counter_charge_nofail, but we hope those allocations are rare,
+  * and won't be worth the trouble.
+  */
+ if (!(gfp & __GFP_KMEMCG) || (gfp & __GFP_NOFAIL))
+  return true;
+ if (in_interrupt() || (!current->mm) || (current->flags & PF_KTHREAD))
+  return true;
+
+ /* If the test is dying, just let it go. */
+       if (unlikely(test_thread_flag(TIF_MEMDIE)
+                    || fatal_signal_pending(current)))
+  return true;
+
+ return __memcg_kmem_newpage_charge(gfp, memcg, order);
+}
+
+/**
+ * memcg_kmem_uncharge_page: uncharge pages from memcg
+ * @page: pointer to struct page being freed
+ * @order: allocation order.
+ *
+ * there is no need to specify memcg here, since it is embedded in page_cgroup
+ */
+static __always_inline void
+memcg_kmem_uncharge_page(struct page *page, int order)
```

```
+{
+ if (memcg_kmem_enabled())
+  __memcg_kmem_uncharge_page(page, order);
+}
+
+/**
+ * memcg_kmem_commit_charge: embeds correct memcg in a page
+ * @page: pointer to struct page recently allocated
+ * @memcg: the memcg structure we charged against
+ * @order: allocation order.
+ *
+ * Needs to be called after memcg_kmem_newpage_charge, regardless of success or
+ * failure of the allocation. if @page is NULL, this function will revert the
+ * charges. Otherwise, it will commit the memcg given by @memcg to the
+ * corresponding page_cgroup.
+ */
+static __always_inline void
+memcg_kmem_commit_charge(struct page *page, struct mem_cgroup *memcg, int order)
+{
+ if (memcg_kmem_enabled() && memcg)
+  __memcg_kmem_commit_charge(page, memcg, order);
+}
+
 #else
 static inline void sock_update_memcg(struct sock *sk)
 {
@@ -406,6 +489,21 @@ static inline void sock_update_memcg(struct sock *sk)
 static inline void sock_release_memcg(struct sock *sk)
 {
 }
+
+static inline bool
+memcg_kmem_newpage_charge(gfp_t gfp, struct mem_cgroup **memcg, int order)
+{
+ return true;
+}
+
+static inline void  memcg_kmem_uncharge_page(struct page *page, int order)
+{
+}
+
+static inline void
+memcg_kmem_commit_charge(struct page *page, struct mem_cgroup *memcg, int order)
+{
+}
 #endif /* CONFIG_MEMCG_KMEM */
 #endif /* _LINUX_MEMCONTROL_H */
```

```
diff --git a/mm/memcontrol.c b/mm/memcontrol.c
index 30eafeb..1182188 100644
--- a/mm/memcontrol.c
+++ b/mm/memcontrol.c
@@ -10,6 +10,10 @@
  * Copyright (C) 2009 Nokia Corporation
  * Author: Kirill A. Shutemov
  *
+ * Kernel Memory Controller
+ * Copyright (C) 2012 Parallels Inc. and Google Inc.
+ * Authors: Glauber Costa and Suleiman Souhlal
+ *
  * This program is free software; you can redistribute it and/or modify
  * it under the terms of the GNU General Public License as published by
  * the Free Software Foundation; either version 2 of the License, or
@@ -2630,6 +2634,171 @@ static void __mem_cgroup_commit_charge(struct mem_cgroup
*memcg,
  memcg_check_events(memcg, page);
 }

+#ifdef CONFIG_MEMCG_KMEM
+static inline bool memcg_can_account_kmem(struct mem_cgroup *memcg)
+{
+ return !mem_cgroup_disabled() && !mem_cgroup_is_root(memcg) &&
+  (memcg->kmem_accounted & KMEM_ACCOUNTED_MASK);
+}
+
+static int memcg_charge_kmem(struct mem_cgroup *memcg, gfp_t gfp, u64 size)
+{
+ struct res_counter *fail_res;
+ struct mem_cgroup *_memcg;
+ int ret = 0;
+ bool may_oom;
+
+ ret = res_counter_charge(&memcg->kmem, size, &fail_res);
+ if (ret)
+  return ret;
+
+ /*
+  * Conditions under which we can wait for the oom_killer.
+  * We have to be able to wait, but also, if we can't retry,
+  * we obviously shouldn't go mess with oom.
+  */
+ may_oom = (gfp & __GFP_WAIT) && !(gfp & __GFP_NORETRY);
+
+ _memcg = memcg;
+ ret = __mem_cgroup_try_charge(NULL, gfp, size >> PAGE_SHIFT,
+         &_memcg, may_oom);
```

```
+
+ if (ret == -EINTR)  {
+  /*
+   * __mem_cgroup_try_charge() chosed to bypass to root due to
+   * OOM kill or fatal signal.  Since our only options are to
+   * either fail the allocation or charge it to this cgroup, do
+   * it as a temporary condition. But we can't fail. From a
+   * kmem/slab perspective, the cache has already been selected,
+   * by mem_cgroup_get_kmem_cache(), so it is too late to change
+   * our minds. This condition will only trigger if the task
+   * entered memcg_charge_kmem in a sane state, but was
+   * OOM-killed.  during __mem_cgroup_try_charge. Tasks that are
+   * already dying when the allocation triggers should have been
+   * already directed to the root cgroup.
+   */
+  res_counter_charge_nofail(&memcg->res, size, &fail_res);
+  if (do_swap_account)
+   res_counter_charge_nofail(&memcg->memsw, size,
+         &fail_res);
+  ret = 0;
+ } else if (ret)
+  res_counter_uncharge(&memcg->kmem, size);
+
+ return ret;
+}
+
+static void memcg_uncharge_kmem(struct mem_cgroup *memcg, u64 size)
+{
+ res_counter_uncharge(&memcg->kmem, size);
+ res_counter_uncharge(&memcg->res, size);
+ if (do_swap_account)
+  res_counter_uncharge(&memcg->memsw, size);
+}
+
+/*
+ * 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
+ * need a further commit step to do the final arrangements.
+ *
+ * It is possible for the task to switch cgroups in this mean time, so at
+ * commit time, we can't rely on task conversion any longer.  We'll then use
+ * the handle argument to return to the caller which cgroup we should commit
+ * against. We could also return the memcg directly and avoid the pointer
+ * passing, but a boolean return value gives better semantics considering
+ * the compiled-out case as well.
+ *
+ * Returning true means the allocation is possible.
+ */
```

```
+bool
+__memcg_kmem_newpage_charge(gfp_t gfp, struct mem_cgroup **_memcg, int order)
+{
+ struct mem_cgroup *memcg;
+ int ret;
+
+ *_memcg = NULL;
+ memcg = try_get_mem_cgroup_from_mm(current->mm);
+
+ /*
+  * very rare case described in mem_cgroup_from_task. Unfortunately there
+  * isn't much we can do without complicating this too much, and it would
+  * be gfp-dependent anyway. Just let it go
+  */
+ if (unlikely(!memcg))
+  return true;
+
+ if (!memcg_can_account_kmem(memcg)) {
+  css_put(&memcg->css);
+  return true;
+ }
+
+ mem_cgroup_get(memcg);
+
+ ret = memcg_charge_kmem(memcg, gfp, PAGE_SIZE << order);
+ if (!ret)
+  *_memcg = memcg;
+ else
+  mem_cgroup_put(memcg);
+
+ css_put(&memcg->css);
+ return (ret == 0);
+}
+
+void __memcg_kmem_commit_charge(struct page *page, struct mem_cgroup *memcg,
+         int order)
+{
+ struct page_cgroup *pc;
+
+ VM_BUG_ON(mem_cgroup_is_root(memcg));
+
+ /* The page allocation failed. Revert */
+ if (!page) {
+  memcg_uncharge_kmem(memcg, PAGE_SIZE << order);
+  mem_cgroup_put(memcg);
+  return;
+ }
+
```

```
+ pc = lookup_page_cgroup(page);
+ lock_page_cgroup(pc);
+ pc->mem_cgroup = memcg;
+ SetPageCgroupUsed(pc);
+ unlock_page_cgroup(pc);
+}
+
+void __memcg_kmem_uncharge_page(struct page *page, int order)
+{
+ struct mem_cgroup *memcg = NULL;
+ struct page_cgroup *pc;
+
+
+ pc = lookup_page_cgroup(page);
+ /*
+  * Fast unlocked return. Theoretically might have changed, have to
+  * check again after locking.
+  */
+ if (!PageCgroupUsed(pc))
+  return;
+
+ lock_page_cgroup(pc);
+ if (PageCgroupUsed(pc)) {
+  memcg = pc->mem_cgroup;
+  ClearPageCgroupUsed(pc);
+ }
+ unlock_page_cgroup(pc);
+
+ /*
+  * We trust that only if there is a memcg associated with the page, it
+  * is a valid allocation
+  */
+ if (!memcg)
+  return;
+
+ VM_BUG_ON(mem_cgroup_is_root(memcg));
+ memcg_uncharge_kmem(memcg, PAGE_SIZE << order);
+ mem_cgroup_put(memcg);
+}
+#endif /* CONFIG_MEMCG_KMEM */
+
 #ifdef CONFIG_TRANSPARENT_HUGEPAGE

 #define PCGF_NOCOPY_AT_SPLIT (1 << PCG_LOCK | 1 << PCG_MIGRATION)
--
1.7.11.7
```

Subject: [PATCH v5 07/14] mm: Allocate kernel pages to the right memcg
Posted by Glauber Costa on Tue, 16 Oct 2012 10:16:44 GMT
View Forum Message <> Reply to Message

When a process tries to allocate a page with the __GFP_KMEMCG flag, the
page allocator will call the corresponding memcg functions to validate
the allocation. Tasks in the root memcg can always proceed.

To avoid adding markers to the page - and a kmem flag that would
necessarily follow, as much as doing page_cgroup lookups for no reason,
whoever is marking its allocations with __GFP_KMEMCG flag is responsible
for telling the page allocator that this is such an allocation at
free_pages() time. This is done by the invocation of
__free_accounted_pages() and free_accounted_pages().

[v2: inverted test order to avoid a memcg_get leak,
  free_accounted_pages simplification]
[v4: test for TIF_MEMDIE at newpage_charge]

Signed-off-by: Glauber Costa <glommer@parallels.com>
Acked-by: Michal Hocko <mhocko@suse.cz>
Acked-by: Mel Gorman <mgorman@suse.de>
Acked-by: Kamezawa Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com>
CC: Christoph Lameter <cl@linux.com>
CC: Pekka Enberg <penberg@cs.helsinki.fi>
CC: Johannes Weiner <hannes@cmpxchg.org>
CC: Suleiman Souhlal <suleiman@google.com>
CC: Tejun Heo <tj@kernel.org>
---
 include/linux/gfp.h | 3 +++
 mm/page_alloc.c | 35 +++++++++++++++++++++++++++++++++++
 2 files changed, 38 insertions(+)

diff --git a/include/linux/gfp.h b/include/linux/gfp.h
index 9289d46..8f6fe34 100644
--- a/include/linux/gfp.h
+++ b/include/linux/gfp.h
@@ -362,6 +362,9 @@ extern void free_pages(unsigned long addr, unsigned int order);
 extern void free_hot_cold_page(struct page *page, int cold);
 extern void free_hot_cold_page_list(struct list_head *list, int cold);

+extern void __free_accounted_pages(struct page *page, unsigned int order);
+extern void free_accounted_pages(unsigned long addr, unsigned int order);
+
 #define __free_page(page) __free_pages((page), 0)
 #define free_page(addr) free_pages((addr), 0)

diff --git a/mm/page_alloc.c b/mm/page_alloc.c
index feddc7f..dcf33ad 100644

```
--- a/mm/page_alloc.c
+++ b/mm/page_alloc.c
@@ -2595,6 +2595,7 @@ __alloc_pages_nodemask(gfp_t gfp_mask, unsigned int order,
  int migratetype = allocflags_to_migratetype(gfp_mask);
  unsigned int cpuset_mems_cookie;
  int alloc_flags = ALLOC_WMARK_LOW|ALLOC_CPUSET;
+ struct mem_cgroup *memcg = NULL;

  gfp_mask &= gfp_allowed_mask;

@@ -2613,6 +2614,13 @@ __alloc_pages_nodemask(gfp_t gfp_mask, unsigned int order,
  if (unlikely(!zonelist->_zonerefs->zone))
   return NULL;

+ /*
+  * Will only have any effect when __GFP_KMEMCG is set.  This is
+  * verified in the (always inline) callee
+  */
+ if (!memcg_kmem_newpage_charge(gfp_mask, &memcg, order))
+  return NULL;
+
 retry_cpuset:
  cpuset_mems_cookie = get_mems_allowed();

@@ -2648,6 +2656,8 @@ out:
  if (unlikely(!put_mems_allowed(cpuset_mems_cookie) && !page))
   goto retry_cpuset;

+ memcg_kmem_commit_charge(page, memcg, order);
+
  return page;
 }
 EXPORT_SYMBOL(__alloc_pages_nodemask);
@@ -2700,6 +2710,31 @@ void free_pages(unsigned long addr, unsigned int order)

 EXPORT_SYMBOL(free_pages);

+/*
+ * __free_accounted_pages and free_accounted_pages will free pages allocated
+ * with __GFP_KMEMCG.
+ *
+ * Those pages are accounted to a particular memcg, embedded in the
+ * corresponding page_cgroup. To avoid adding a hit in the allocator to search
+ * for that information only to find out that it is NULL for users who have no
+ * interest in that whatsoever, we provide these functions.
+ *
+ * The caller knows better which flags it relies on.
+ */
```

```
+void __free_accounted_pages(struct page *page, unsigned int order)
+{
+ memcg_kmem_uncharge_page(page, order);
+ __free_pages(page, order);
+}
+
+void free_accounted_pages(unsigned long addr, unsigned int order)
+{
+ if (addr != 0) {
+  VM_BUG_ON(!virt_addr_valid((void *)addr));
+  __free_accounted_pages(virt_to_page((void *)addr), order);
+ }
+}
+
 static void *make_alloc_exact(unsigned long addr, unsigned order, size_t size)
 {
  if (addr) {
--
1.7.11.7
```

Subject: [PATCH v5 09/14] memcg: kmem accounting lifecycle management
Posted by Glauber Costa on Tue, 16 Oct 2012 10:16:46 GMT
View Forum Message <> Reply to Message

Because kmem charges can outlive the cgroup, we need to make sure that
we won't free the memcg structure while charges are still in flight.
For reviewing simplicity, the charge functions will issue
mem_cgroup_get() at every charge, and mem_cgroup_put() at every
uncharge.

This can get expensive, however, and we can do better. mem_cgroup_get()
only really needs to be issued once: when the first limit is set. In the
same spirit, we only need to issue mem_cgroup_put() when the last charge
is gone.

We'll need an extra bit in kmem_accounted for that: KMEM_ACCOUNTED_DEAD.
it will be set when the cgroup dies, if there are charges in the group.
If there aren't, we can proceed right away.

Our uncharge function will have to test that bit every time the charges
drop to 0. Because that is not the likely output of
res_counter_uncharge, this should not impose a big hit on us: it is
certainly much better than a reference count decrease at every
operation.

[v3: merged all lifecycle related patches in one]
[v5: changed memcg_kmem_dead's name]

```
Signed-off-by: Glauber Costa <glommer@parallels.com>
Acked-by: Michal Hocko <mhocko@suse.cz>
Acked-by: Kamezawa Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com>
CC: Christoph Lameter <cl@linux.com>
CC: Pekka Enberg <penberg@cs.helsinki.fi>
CC: Johannes Weiner <hannes@cmpxchg.org>
CC: Suleiman Souhlal <suleiman@google.com>
CC: Tejun Heo <tj@kernel.org>
---
 mm/memcontrol.c | 56 ++++++++++++++++++++++++++++++++++++++++++++++++-------
 1 file changed, 49 insertions(+), 7 deletions(-)

diff --git a/mm/memcontrol.c b/mm/memcontrol.c
index 1182188..e24b388 100644
--- a/mm/memcontrol.c
+++ b/mm/memcontrol.c
@@ -344,6 +344,7 @@ struct mem_cgroup {
 /* internal only representation about the status of kmem accounting. */
 enum {
  KMEM_ACCOUNTED_ACTIVE = 0, /* accounted by this cgroup itself */
+ KMEM_ACCOUNTED_DEAD, /* dead memcg, pending kmem charges */
 };

 #define KMEM_ACCOUNTED_MASK (1 << KMEM_ACCOUNTED_ACTIVE)
@@ -353,6 +354,22 @@ static void memcg_kmem_set_active(struct mem_cgroup *memcg)
 {
  set_bit(KMEM_ACCOUNTED_ACTIVE, &memcg->kmem_accounted);
 }
+
+static bool memcg_kmem_is_active(struct mem_cgroup *memcg)
+{
+ return test_bit(KMEM_ACCOUNTED_ACTIVE, &memcg->kmem_accounted);
+}
+
+static void memcg_kmem_mark_dead(struct mem_cgroup *memcg)
+{
+ if (test_bit(KMEM_ACCOUNTED_ACTIVE, &memcg->kmem_accounted))
+  set_bit(KMEM_ACCOUNTED_DEAD, &memcg->kmem_accounted);
+}
+
+static bool memcg_kmem_test_and_clear_dead(struct mem_cgroup *memcg)
+{
+ return test_and_clear_bit(KMEM_ACCOUNTED_DEAD, &memcg->kmem_accounted);
+}
 #endif

 /* Stuffs for move charges at task migration. */
```

```
@@ -2690,10 +2707,16 @@ static int memcg_charge_kmem(struct mem_cgroup *memcg, gfp_t
gfp, u64 size)

 static void memcg_uncharge_kmem(struct mem_cgroup *memcg, u64 size)
 {
- res_counter_uncharge(&memcg->kmem, size);
  res_counter_uncharge(&memcg->res, size);
  if (do_swap_account)
   res_counter_uncharge(&memcg->memsw, size);
+
+ /* Not down to 0 */
+ if (res_counter_uncharge(&memcg->kmem, size))
+  return;
+
+ if (memcg_kmem_test_and_clear_dead(memcg))
+  mem_cgroup_put(memcg);
 }

 /*
@@ -2732,13 +2755,9 @@ __memcg_kmem_newpage_charge(gfp_t gfp, struct mem_cgroup
**_memcg, int order)
   return true;
  }

- mem_cgroup_get(memcg);
-
  ret = memcg_charge_kmem(memcg, gfp, PAGE_SIZE << order);
  if (!ret)
   *_memcg = memcg;
- else
-  mem_cgroup_put(memcg);

  css_put(&memcg->css);
  return (ret == 0);
@@ -2754,7 +2773,6 @@ void __memcg_kmem_commit_charge(struct page *page, struct
mem_cgroup *memcg,
  /* The page allocation failed. Revert */
  if (!page) {
   memcg_uncharge_kmem(memcg, PAGE_SIZE << order);
-  mem_cgroup_put(memcg);
   return;
  }

@@ -2795,7 +2813,6 @@ 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);
- mem_cgroup_put(memcg);
```

```
 }
 #endif /* CONFIG_MEMCG_KMEM */

@@ -4179,6 +4196,13 @@ static int memcg_update_kmem_limit(struct cgroup *cont, u64 val)
   VM_BUG_ON(ret);

   memcg_kmem_set_active(memcg);
+  /*
+   * kmem charges can outlive the cgroup. In the case of slab
+   * pages, for instance, a page contain objects from various
+   * processes, so it is unfeasible to migrate them away. We
+   * need to reference count the memcg because of that.
+   */
+  mem_cgroup_get(memcg);
  } else
   ret = res_counter_set_limit(&memcg->kmem, val);
 out:
@@ -4192,6 +4216,10 @@ static void memcg_propagate_kmem(struct mem_cgroup *memcg,
      struct mem_cgroup *parent)
 {
  memcg->kmem_accounted = parent->kmem_accounted;
+#ifdef CONFIG_MEMCG_KMEM
+ if (memcg_kmem_is_active(memcg))
+  mem_cgroup_get(memcg);
+#endif
 }

 /*
@@ -4875,6 +4903,20 @@ static int memcg_init_kmem(struct mem_cgroup *memcg, struct
cgroup_subsys *ss)
 static void kmem_cgroup_destroy(struct mem_cgroup *memcg)
 {
  mem_cgroup_sockets_destroy(memcg);
+
+ memcg_kmem_mark_dead(memcg);
+
+ if (res_counter_read_u64(&memcg->kmem, RES_USAGE) != 0)
+  return;
+
+ /*
+  * Charges already down to 0, undo mem_cgroup_get() done in the charge
+  * path here, being careful not to race with memcg_uncharge_kmem: it is
+  * possible that the charges went down to 0 between mark_dead and the
+  * res_counter read, so in that case, we don't need the put
+  */
+ if (memcg_kmem_test_and_clear_dead(memcg))
+  mem_cgroup_put(memcg);
 }
```

```
 #else
 static int memcg_init_kmem(struct mem_cgroup *memcg, struct cgroup_subsys *ss)
--
1.7.11.7
```

Subject: [PATCH v5 10/14] memcg: use static branches when code not in use
Posted by Glauber Costa on Tue, 16 Oct 2012 10:16:47 GMT
View Forum Message <> Reply to Message

We can use static branches to patch the code in or out when not used.

Because the _ACTIVE bit on kmem_accounted is only set after the increment is done, we guarantee that the root memcg will always be selected for kmem charges until all call sites are patched (see memcg_kmem_enabled). This guarantees that no mischarges are applied.

static branch decrement happens when the last reference count from the kmem accounting in memcg dies. This will only happen when the charges drop down to 0.

When that happen, we need to disable the static branch only on those memcgs that enabled it. To achieve this, we would be forced to complicate the code by keeping track of which memcgs were the ones that actually enabled limits, and which ones got it from its parents.

It is a lot simpler just to do static_key_slow_inc() on every child that is accounted.

[v4: adapted this patch to the changes in kmem_accounted]

Signed-off-by: Glauber Costa <glommer@parallels.com>
Acked-by: Michal Hocko <mhocko@suse.cz>
Acked-by: Kamezawa Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com>
CC: Christoph Lameter <cl@linux.com>
CC: Pekka Enberg <penberg@cs.helsinki.fi>
CC: Johannes Weiner <hannes@cmpxchg.org>
CC: Suleiman Souhlal <suleiman@google.com>
CC: Tejun Heo <tj@kernel.org>
---
 include/linux/memcontrol.h | 4 ++-
 mm/memcontrol.c | 79 ++---
 2 files changed, 78 insertions(+), 5 deletions(-)

diff --git a/include/linux/memcontrol.h b/include/linux/memcontrol.h
index 303a456..34e96cf 100644
--- a/include/linux/memcontrol.h
+++ b/include/linux/memcontrol.h

```
@@ -22,6 +22,7 @@
 #include <linux/cgroup.h>
 #include <linux/vm_event_item.h>
 #include <linux/hardirq.h>
+#include <linux/jump_label.h>

 struct mem_cgroup;
 struct page_cgroup;
@@ -401,9 +402,10 @@ struct sock;
 void sock_update_memcg(struct sock *sk);
 void sock_release_memcg(struct sock *sk);

+extern struct static_key memcg_kmem_enabled_key;
 static inline bool memcg_kmem_enabled(void)
 {
- return true;
+ return static_key_false(&memcg_kmem_enabled_key);
 }

 bool __memcg_kmem_newpage_charge(gfp_t gfp, struct mem_cgroup **memcg,
diff --git a/mm/memcontrol.c b/mm/memcontrol.c
index e24b388..1dd31a1 100644
--- a/mm/memcontrol.c
+++ b/mm/memcontrol.c
@@ -344,10 +344,13 @@ struct mem_cgroup {
 /* internal only representation about the status of kmem accounting. */
 enum {
  KMEM_ACCOUNTED_ACTIVE = 0, /* accounted by this cgroup itself */
+ KMEM_ACCOUNTED_ACTIVATED, /* static key enabled. */
  KMEM_ACCOUNTED_DEAD, /* dead memcg, pending kmem charges */
 };

-#define KMEM_ACCOUNTED_MASK (1 << KMEM_ACCOUNTED_ACTIVE)
+/* We account when limit is on, but only after call sites are patched */
+#define KMEM_ACCOUNTED_MASK \
+  ((1 << KMEM_ACCOUNTED_ACTIVE) | (1 << KMEM_ACCOUNTED_ACTIVATED))

 #ifdef CONFIG_MEMCG_KMEM
 static void memcg_kmem_set_active(struct mem_cgroup *memcg)
@@ -360,6 +363,11 @@ 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)
+{
+ set_bit(KMEM_ACCOUNTED_ACTIVATED, &memcg->kmem_accounted);
+}
+
```

```
 static void memcg_kmem_mark_dead(struct mem_cgroup *memcg)
 {
  if (test_bit(KMEM_ACCOUNTED_ACTIVE, &memcg->kmem_accounted))
@@ -529,6 +537,26 @@ static void disarm_sock_keys(struct mem_cgroup *memcg)
 }
 #endif

+#ifdef CONFIG_MEMCG_KMEM
+struct static_key memcg_kmem_enabled_key;
+
+static void disarm_kmem_keys(struct mem_cgroup *memcg)
+{
+ if (memcg_kmem_is_active(memcg))
+  static_key_slow_dec(&memcg_kmem_enabled_key);
+}
+#else
+static void disarm_kmem_keys(struct mem_cgroup *memcg)
+{
+}
+#endif /* CONFIG_MEMCG_KMEM */
+
+static void disarm_static_keys(struct mem_cgroup *memcg)
+{
+ disarm_sock_keys(memcg);
+ disarm_kmem_keys(memcg);
+}
+
 static void drain_all_stock_async(struct mem_cgroup *memcg);

 static struct mem_cgroup_per_zone *
@@ -4165,6 +4193,8 @@ static int memcg_update_kmem_limit(struct cgroup *cont, u64 val)
 {
  int ret = -EINVAL;
 #ifdef CONFIG_MEMCG_KMEM
+ bool must_inc_static_branch = false;
+
  struct mem_cgroup *memcg = mem_cgroup_from_cont(cont);
  /*
   * For simplicity, we won't allow this to be disabled.  It also can't
@@ -4195,7 +4225,15 @@ static int memcg_update_kmem_limit(struct cgroup *cont, u64 val)
   ret = res_counter_set_limit(&memcg->kmem, val);
   VM_BUG_ON(ret);

-  memcg_kmem_set_active(memcg);
+  /*
+   * 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);
+  must_inc_static_branch = true;
   /*
    * kmem charges can outlive the cgroup. In the case of slab
    * pages, for instance, a page contain objects from various
@@ -4208,6 +4246,27 @@ static int memcg_update_kmem_limit(struct cgroup *cont, u64 val)
 out:
  mutex_unlock(&set_limit_mutex);
  cgroup_unlock();
+
+ /*
+  * We are by now familiar with the fact that we can't inc the static
+  * branch inside cgroup_lock. See disarm functions for details. A
+  * worker here is overkill, but also wrong: After the limit is set, we
+  * must start accounting right away. Since this operation can't fail,
+  * we can safely defer it to here - no rollback will be needed.
+  *
+  * The boolean used to control this is also safe, because
+  * KMEM_ACCOUNTED_ACTIVATED guarantees that only one process will be
+  * able to set it to true;
+  */
+ if (must_inc_static_branch) {
+  static_key_slow_inc(&memcg_kmem_enabled_key);
+  /*
+   * setting the active bit after the inc will guarantee no one
+   * starts accounting before all call sites are patched
+   */
+  memcg_kmem_set_active(memcg);
+ }
+
 #endif
  return ret;
 }
@@ -4217,8 +4276,20 @@ static void memcg_propagate_kmem(struct mem_cgroup *memcg,
 {
  memcg->kmem_accounted = parent->kmem_accounted;
 #ifdef CONFIG_MEMCG_KMEM
- if (memcg_kmem_is_active(memcg))
+ /*
+  * When that happen, we need to disable the static branch only on those
+  * memcgs that enabled it. To achieve this, we would be forced to
+  * complicate the code by keeping track of which memcgs were the ones
+  * that actually enabled limits, and which ones got it from its
+  * parents.
+  *
```

```
+  * 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);
+ }
 #endif
 }

@@ -5138,7 +5209,7 @@ static void free_work(struct work_struct *work)
   * to move this code around, and make sure it is outside
   * the cgroup_lock.
   */
- disarm_sock_keys(memcg);
+ disarm_static_keys(memcg);
  if (size < PAGE_SIZE)
   kfree(memcg);
  else
--
1.7.11.7
```

Subject: [PATCH v5 11/14] memcg: allow a memcg with kmem charges to be destructed.
Posted by Glauber Costa on Tue, 16 Oct 2012 10:16:48 GMT
View Forum Message <> Reply to Message

Because the ultimate goal of the kmem tracking in memcg is to track slab pages as well, we can't guarantee that we'll always be able to point a page to a particular process, and migrate the charges along with it - since in the common case, a page will contain data belonging to multiple processes.

Because of that, when we destroy a memcg, we only make sure the destruction will succeed by discounting the kmem charges from the user charges when we try to empty the cgroup.

Signed-off-by: Glauber Costa <glommer@parallels.com>
Acked-by: Kamezawa Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com>
Reviewed-by: Michal Hocko <mhocko@suse.cz>
CC: Christoph Lameter <cl@linux.com>
CC: Pekka Enberg <penberg@cs.helsinki.fi>
CC: Johannes Weiner <hannes@cmpxchg.org>
CC: Suleiman Souhlal <suleiman@google.com>
CC: Tejun Heo <tj@kernel.org>
---
 mm/memcontrol.c | 17 ++++++++++++++++-

```
 1 file changed, 16 insertions(+), 1 deletion(-)

diff --git a/mm/memcontrol.c b/mm/memcontrol.c
index 1dd31a1..9086cf2 100644
--- a/mm/memcontrol.c
+++ b/mm/memcontrol.c
@@ -544,6 +544,11 @@ static void disarm_kmem_keys(struct mem_cgroup *memcg)
 {
  if (memcg_kmem_is_active(memcg))
   static_key_slow_dec(&memcg_kmem_enabled_key);
+ /*
+  * This check can't live in kmem destruction function,
+  * since the charges will outlive the cgroup
+  */
+ WARN_ON(res_counter_read_u64(&memcg->kmem, RES_USAGE) != 0);
 }
 #else
 static void disarm_kmem_keys(struct mem_cgroup *memcg)
@@ -3993,6 +3998,7 @@ static int mem_cgroup_force_empty(struct mem_cgroup *memcg, bool
free_all)
  int node, zid, shrink;
  int nr_retries = MEM_CGROUP_RECLAIM_RETRIES;
  struct cgroup *cgrp = memcg->css.cgroup;
+ u64 usage;

  css_get(&memcg->css);

@@ -4026,8 +4032,17 @@ move_account:
   mem_cgroup_end_move(memcg);
   memcg_oom_recover(memcg);
   cond_resched();
+  /*
+   * Kernel memory may not necessarily be trackable to a specific
+   * process. So they are not migrated, and therefore we can't
+   * expect their value to drop to 0 here.
+   *
+   * having res filled up with kmem only is enough
+   */
+  usage = res_counter_read_u64(&memcg->res, RES_USAGE) -
+   res_counter_read_u64(&memcg->kmem, RES_USAGE);
  /* "ret" should also be checked to ensure all lists are empty. */
- } while (res_counter_read_u64(&memcg->res, RES_USAGE) > 0 || ret);
+ } while (usage > 0 || ret);
 out:
  css_put(&memcg->css);
  return ret;
--
1.7.11.7
```

Subject: [PATCH v5 13/14] protect architectures where THREAD_SIZE >= PAGE_SIZE against fork bombs
Posted by Glauber Costa on Tue, 16 Oct 2012 10:16:50 GMT
View Forum Message <> Reply to Message

Because those architectures will draw their stacks directly from the
page allocator, rather than the slab cache, we can directly pass
__GFP_KMEMCG flag, and issue the corresponding free_pages.

This code path is taken when the architecture doesn't define
CONFIG_ARCH_THREAD_INFO_ALLOCATOR (only ia64 seems to), and has
THREAD_SIZE >= PAGE_SIZE. Luckily, most - if not all - of the remaining
architectures fall in this category.

This will guarantee that every stack page is accounted to the memcg the
process currently lives on, and will have the allocations to fail if
they go over limit.

For the time being, I am defining a new variant of THREADINFO_GFP, not
to mess with the other path. Once the slab is also tracked by memcg, we
can get rid of that flag.

Tested to successfully protect against :(){ :|:& };:

Signed-off-by: Glauber Costa <glommer@parallels.com>
Acked-by: Frederic Weisbecker <fweisbec@redhat.com>
Acked-by: Kamezawa Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com>
Reviewed-by: Michal Hocko <mhocko@suse.cz>
CC: Christoph Lameter <cl@linux.com>
CC: Pekka Enberg <penberg@cs.helsinki.fi>
CC: Johannes Weiner <hannes@cmpxchg.org>
CC: Suleiman Souhlal <suleiman@google.com>
CC: Tejun Heo <tj@kernel.org>
---
 include/linux/thread_info.h | 2 ++
 kernel/fork.c | 4 ++--
 2 files changed, 4 insertions(+), 2 deletions(-)

diff --git a/include/linux/thread_info.h b/include/linux/thread_info.h
index ccc1899..e7e0473 100644
--- a/include/linux/thread_info.h
+++ b/include/linux/thread_info.h
@@ -61,6 +61,8 @@ extern long do_no_restart_syscall(struct restart_block *parm);
 # define THREADINFO_GFP (GFP_KERNEL | __GFP_NOTRACK)
 #endif

+#define THREADINFO_GFP_ACCOUNTED (THREADINFO_GFP | __GFP_KMEMCG)
+
 /*

```
  * flag set/clear/test wrappers
  * - pass TIF_xxxx constants to these functions
diff --git a/kernel/fork.c b/kernel/fork.c
index 03b86f1..b3f6298 100644
--- a/kernel/fork.c
+++ b/kernel/fork.c
@@ -146,7 +146,7 @@ void __weak arch_release_thread_info(struct thread_info *ti)
 static struct thread_info *alloc_thread_info_node(struct task_struct *tsk,
         int node)
 {
- struct page *page = alloc_pages_node(node, THREADINFO_GFP,
+ struct page *page = alloc_pages_node(node, THREADINFO_GFP_ACCOUNTED,
           THREAD_SIZE_ORDER);

  return page ? page_address(page) : NULL;
@@ -154,7 +154,7 @@ static struct thread_info *alloc_thread_info_node(struct task_struct *tsk,

 static inline void free_thread_info(struct thread_info *ti)
 {
- free_pages((unsigned long)ti, THREAD_SIZE_ORDER);
+ free_accounted_pages((unsigned long)ti, THREAD_SIZE_ORDER);
 }
 # else
 static struct kmem_cache *thread_info_cache;
--
1.7.11.7
```

Subject: [PATCH v5 14/14] Add documentation about the kmem controller
Posted by Glauber Costa on Tue, 16 Oct 2012 10:16:51 GMT
View Forum Message <> Reply to Message

```
Signed-off-by: Glauber Costa <glommer@parallels.com>
CC: Frederic Weisbecker <fweisbec@redhat.com>
CC: Kamezawa Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com>
CC: Michal Hocko <mhocko@suse.cz>
CC: Christoph Lameter <cl@linux.com>
CC: Pekka Enberg <penberg@cs.helsinki.fi>
CC: Johannes Weiner <hannes@cmpxchg.org>
CC: Suleiman Souhlal <suleiman@google.com>
CC: Tejun Heo <tj@kernel.org>
---
 Documentation/cgroups/memory.txt | 58 ++++++++++++++++++++++++++++++++++++++-
 1 file changed, 57 insertions(+), 1 deletion(-)

diff --git a/Documentation/cgroups/memory.txt b/Documentation/cgroups/memory.txt
index c07f7b4..dd15be8 100644
--- a/Documentation/cgroups/memory.txt
```

```
+++ b/Documentation/cgroups/memory.txt
@@ -71,6 +71,11 @@ Brief summary of control files.
  memory.oom_control   # set/show oom controls.
  memory.numa_stat   # show the number of memory usage per numa node

+ memory.kmem.limit_in_bytes      # set/show hard limit for kernel memory
+ memory.kmem.usage_in_bytes      # show current kernel memory allocation
+ memory.kmem.failcnt             # show the number of kernel memory usage hits limits
+ memory.kmem.max_usage_in_bytes  # show max kernel memory usage recorded
+
  memory.kmem.tcp.limit_in_bytes  # set/show hard limit for tcp buf memory
  memory.kmem.tcp.usage_in_bytes  # show current tcp buf memory allocation
  memory.kmem.tcp.failcnt            # show the number of tcp buf memory usage hits limits
@@ -268,20 +273,65 @@ the amount of kernel memory used by the system. Kernel memory is
fundamentally
 different than user memory, since it can't be swapped out, which makes it
 possible to DoS the system by consuming too much of this precious resource.

+Kernel memory won't be accounted at all until limit on a group is set. This
+allows for existing setups to continue working without disruption.  The limit
+cannot be set if the cgroup have children, or if there are already tasks in the
+cgroup. When use_hierarchy == 1 and a group is accounted, its children will
+automatically be accounted regardless of their limit value.
+
+After a controller is first limited, it will be kept being accounted until it
+is removed. The memory limitation itself, can of course be removed by writing
+-1 to memory.kmem.limit_in_bytes. In this case, kmem will be accounted, but not
+limited.
+
 Kernel memory limits are not imposed for the root cgroup. Usage for the root
-cgroup may or may not be accounted.
+cgroup may or may not be accounted. The memory used is accumulated into
+memory.kmem.usage_in_bytes, or in a separate counter when it makes sense.
+(currently only for tcp).
+The main "kmem" counter is fed into the main counter, so kmem charges will
+also be visible from the user counter.

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

 2.7.1 Current Kernel Memory resources accounted

+* stack pages: every process consumes some stack pages. By accounting into
+kernel memory, we prevent new processes from being created when the kernel
+memory usage is too high.
+
 * sockets memory pressure: some sockets protocols have memory pressure
 thresholds. The Memory Controller allows them to be controlled individually
```

per cgroup, instead of globally.

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

+2.7.3 Common use cases
+
+Because the "kmem" counter is fed to the main user counter, kernel memory can
+never be limited completely independently of user memory. Say "U" is the user
+limit, and "K" the kernel limit. There are three possible ways limits can be
+set:
+
+   U != 0, K = unlimited:
+   This is the standard memcg limitation mechanism already present before kmem
+   accounting. Kernel memory is completely ignored.
+
+   U != 0, K < U:
+   Kernel memory is a subset of the user memory. This setup is useful in
+   deployments where the total amount of memory per-cgroup is overcommited.
+   Overcommiting kernel memory limits is definitely not recommended, since the
+   box can still run out of non-reclaimable memory.
+   In this case, the admin could set up K so that the sum of all groups is
+   never greater than the total memory, and freely set U at the cost of his
+   QoS.
+
+   U != 0, K >= U:
+   Since kmem charges will also be fed to the user counter and reclaim will be
+   triggered for the cgroup for both kinds of memory. This setup gives the
+   admin a unified view of memory, and it is also useful for people who just
+   want to track kernel memory usage.
+
 3. User Interface

 0. Configuration
@@ -290,6 +340,7 @@ a. Enable CONFIG_CGROUPS
 b. Enable CONFIG_RESOURCE_COUNTERS
 c. Enable CONFIG_MEMCG
 d. Enable CONFIG_MEMCG_SWAP (to use swap extension)
+d. Enable CONFIG_MEMCG_KMEM (to use kmem extension)

 1. Prepare the cgroups (see cgroups.txt, Why are cgroups needed?)
 # mount -t tmpfs none /sys/fs/cgroup
@@ -406,6 +457,11 @@ About use_hierarchy, see Section 6.
   Because rmdir() moves all pages to parent, some out-of-use page caches can be
   moved to the parent. If you want to avoid that, force_empty will be useful.

+ Also, note that when memory.kmem.limit_in_bytes is set the charges due to
+ kernel pages will still be seen. This is not considered a failure and the
+ write will still return success. In this case, it is expected that

+ memory.kmem.usage_in_bytes == memory.usage_in_bytes.
+
   About use_hierarchy, see Section 6.

 5.2 stat file
--
1.7.11.7

Subject: Re: [PATCH v5 04/14] kmem accounting basic infrastructure
Posted by Michal Hocko on Tue, 16 Oct 2012 12:14:58 GMT
View Forum Message <> Reply to Message

On Tue 16-10-12 14:16:41, Glauber Costa wrote:
> This patch adds the basic infrastructure for the accounting of kernel
> memory. To control that, the following files are created:
>
> * memory.kmem.usage_in_bytes
> * memory.kmem.limit_in_bytes
> * memory.kmem.failcnt
> * memory.kmem.max_usage_in_bytes
>
> They have the same meaning of their user memory counterparts. They
> reflect the state of the "kmem" res_counter.
>
> Per cgroup kmem memory accounting is not enabled until a limit is set
> for the group. Once the limit is set the accounting cannot be disabled
> for that group. This means that after the patch is applied, no
> behavioral changes exists for whoever is still using memcg to control
> their memory usage, until memory.kmem.limit_in_bytes is set for the
> first time.
>
> We always account to both user and kernel resource_counters. This
> effectively means that an independent kernel limit is in place when the
> limit is set to a lower value than the user memory. A equal or higher
> value means that the user limit will always hit first, meaning that kmem
> is effectively unlimited.
>
> People who want to track kernel memory but not limit it, can set this
> limit to a very high number (like RESOURCE_MAX - 1page - that no one
> will ever hit, or equal to the user memory)
>
> [v4: make kmem files part of the main array;
>   do not allow limit to be set for non-empty cgroups]
> [v5: cosmetic changes]
>
> Signed-off-by: Glauber Costa <glommer@parallels.com>
> Acked-by: Kamezawa Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com>

> CC: Michal Hocko <mhocko@suse.cz>
> CC: Johannes Weiner <hannes@cmpxchg.org>
> CC: Tejun Heo <tj@kernel.org>

Acked-by: Michal Hocko <mhocko@suse.cz>

> ---
> mm/memcontrol.c | 116
++-
> 1 file changed, 115 insertions(+), 1 deletion(-)
>
> diff --git a/mm/memcontrol.c b/mm/memcontrol.c
> index 71d259e..30eafeb 100644
> --- a/mm/memcontrol.c
> +++ b/mm/memcontrol.c
> @@ -266,6 +266,10 @@ struct mem_cgroup {
> };
>
> /*
> + * the counter to account for kernel memory usage.
> + */
> + struct res_counter kmem;
> + /*
>   * Per cgroup active and inactive list, similar to the
>   * per zone LRU lists.
>   */
> @@ -280,6 +284,7 @@ struct mem_cgroup {
>   * Should the accounting and control be hierarchical, per subtree?
>   */
> bool use_hierarchy;
> + unsigned long kmem_accounted; /* See KMEM_ACCOUNTED_*, below */
>
> bool oom_lock;
> atomic_t under_oom;
> @@ -332,6 +337,20 @@ struct mem_cgroup {
> #endif
> };
>
> +/* internal only representation about the status of kmem accounting. */
> +enum {
> + KMEM_ACCOUNTED_ACTIVE = 0, /* accounted by this cgroup itself */
> +};
> +
> +#define KMEM_ACCOUNTED_MASK (1 << KMEM_ACCOUNTED_ACTIVE)
> +
> +#ifdef CONFIG_MEMCG_KMEM
> +static void memcg_kmem_set_active(struct mem_cgroup *memcg)
> +{

> + set_bit(KMEM_ACCOUNTED_ACTIVE, &memcg->kmem_accounted);
> +}
> +#endif
> +
> /* Stuffs for move charges at task migration. */
> /*
>   * Types of charges to be moved. "move_charge_at_immitgrate" is treated as a
> @@ -390,6 +409,7 @@ enum res_type {
>   _MEM,
>   _MEMSWAP,
>   _OOM_TYPE,
> + _KMEM,
> };
>
> #define MEMFILE_PRIVATE(x, val) ((x) << 16 | (val))
> @@ -1433,6 +1453,10 @@ done:
>   res_counter_read_u64(&memcg->memsw, RES_USAGE) >> 10,
>   res_counter_read_u64(&memcg->memsw, RES_LIMIT) >> 10,
>   res_counter_read_u64(&memcg->memsw, RES_FAILCNT));
> + printk(KERN_INFO "kmem: usage %llukB, limit %llukB, failcnt %llu\n",
> + res_counter_read_u64(&memcg->kmem, RES_USAGE) >> 10,
> + res_counter_read_u64(&memcg->kmem, RES_LIMIT) >> 10,
> + res_counter_read_u64(&memcg->kmem, RES_FAILCNT));
> }
>
> /*
> @@ -3940,6 +3964,9 @@ static ssize_t mem_cgroup_read(struct cgroup *cont, struct cftype
*cft,
>   else
>   val = res_counter_read_u64(&memcg->memsw, name);
>   break;
> + case _KMEM:
> + val = res_counter_read_u64(&memcg->kmem, name);
> + break;
>   default:
>   BUG();
>   }
> @@ -3947,6 +3974,57 @@ static ssize_t mem_cgroup_read(struct cgroup *cont, struct cftype
*cft,
>   len = scnprintf(str, sizeof(str), "%llu\n", (unsigned long long)val);
>   return simple_read_from_buffer(buf, nbytes, ppos, str, len);
> }
> +
> +static int memcg_update_kmem_limit(struct cgroup *cont, u64 val)
> +{
> + int ret = -EINVAL;
> +#ifdef CONFIG_MEMCG_KMEM
> + struct mem_cgroup *memcg = mem_cgroup_from_cont(cont);

```
> + /*
> +  * For simplicity, we won't allow this to be disabled.  It also can't
> +  * be changed if the cgroup has children already, or if tasks had
> +  * already joined.
> +  *
> +  * If tasks join before we set the limit, a person looking at
> +  * kmem.usage_in_bytes will have no way to determine when it took
> +  * place, which makes the value quite meaningless.
> +  *
> +  * After it first became limited, changes in the value of the limit are
> +  * of course permitted.
> +  *
> +  * Taking the cgroup_lock is really offensive, but it is so far the only
> +  * way to guarantee that no children will appear. There are plenty of
> +  * other offenders, and they should all go away. Fine grained locking
> +  * is probably the way to go here. When we are fully hierarchical, we
> +  * can also get rid of the use_hierarchy check.
> +  */
> + cgroup_lock();
> + mutex_lock(&set_limit_mutex);
> + if (!memcg->kmem_accounted && val != RESOURCE_MAX) {
> +  if (cgroup_task_count(cont) || (memcg->use_hierarchy &&
> +      !list_empty(&cont->children))) {
> +   ret = -EBUSY;
> +   goto out;
> +  }
> +  ret = res_counter_set_limit(&memcg->kmem, val);
> +  VM_BUG_ON(ret);
> +
> +  memcg_kmem_set_active(memcg);
> + } else
> +  ret = res_counter_set_limit(&memcg->kmem, val);
> +out:
> + mutex_unlock(&set_limit_mutex);
> + cgroup_unlock();
> +#endif
> + return ret;
> +}
> +
> +static void memcg_propagate_kmem(struct mem_cgroup *memcg,
> +     struct mem_cgroup *parent)
> +{
> + memcg->kmem_accounted = parent->kmem_accounted;
> +}
> +
>  /*
>   * The user of this function is...
>   * RES_LIMIT.
```

> @@ -3978,8 +4056,12 @@ static int mem_cgroup_write(struct cgroup *cont, struct cftype *cft,
>   break;
>   if (type == _MEM)
>   ret = mem_cgroup_resize_limit(memcg, val);
> - else
> + else if (type == _MEMSWAP)
>   ret = mem_cgroup_resize_memsw_limit(memcg, val);
> + else if (type == _KMEM)
> +   ret = memcg_update_kmem_limit(cont, val);
> + else
> +   return -EINVAL;
>   break;
>   case RES_SOFT_LIMIT:
>   ret = res_counter_memparse_write_strategy(buffer, &val);
> @@ -4045,12 +4127,16 @@ static int mem_cgroup_reset(struct cgroup *cont, unsigned int
event)
>   case RES_MAX_USAGE:
>   if (type == _MEM)
>   res_counter_reset_max(&memcg->res);
> + else if (type == _KMEM)
> +   res_counter_reset_max(&memcg->kmem);
>   else
>   res_counter_reset_max(&memcg->memsw);
>   break;
>   case RES_FAILCNT:
>   if (type == _MEM)
>   res_counter_reset_failcnt(&memcg->res);
> + else if (type == _KMEM)
> +   res_counter_reset_failcnt(&memcg->kmem);
>   else
>   res_counter_reset_failcnt(&memcg->memsw);
>   break;
> @@ -4728,6 +4814,31 @@ static struct cftype mem_cgroup_files[] = {
>   .read = mem_cgroup_read,
>   },
> #endif
> +#ifdef CONFIG_MEMCG_KMEM
> + {
> + .name = "kmem.limit_in_bytes",
> + .private = MEMFILE_PRIVATE(_KMEM, RES_LIMIT),
> + .write_string = mem_cgroup_write,
> + .read = mem_cgroup_read,
> + },
> + {
> + .name = "kmem.usage_in_bytes",
> + .private = MEMFILE_PRIVATE(_KMEM, RES_USAGE),
> + .read = mem_cgroup_read,
> + },

```
> + {
> +  .name = "kmem.failcnt",
> +  .private = MEMFILE_PRIVATE(_KMEM, RES_FAILCNT),
> +  .trigger = mem_cgroup_reset,
> +  .read = mem_cgroup_read,
> + },
> + {
> +  .name = "kmem.max_usage_in_bytes",
> +  .private = MEMFILE_PRIVATE(_KMEM, RES_MAX_USAGE),
> +  .trigger = mem_cgroup_reset,
> +  .read = mem_cgroup_read,
> + },
> +#endif
>   { }, /* terminate */
>  };
>
> @@ -4973,6 +5084,7 @@ mem_cgroup_create(struct cgroup *cont)
>   if (parent && parent->use_hierarchy) {
>    res_counter_init(&memcg->res, &parent->res);
>    res_counter_init(&memcg->memsw, &parent->memsw);
> +  res_counter_init(&memcg->kmem, &parent->kmem);
>    /*
>     * We increment refcnt of the parent to ensure that we can
>     * safely access it on res_counter_charge/uncharge.
> @@ -4980,9 +5092,11 @@ mem_cgroup_create(struct cgroup *cont)
>     * mem_cgroup(see mem_cgroup_put).
>     */
>    mem_cgroup_get(parent);
> +  memcg_propagate_kmem(memcg, parent);
>   } else {
>    res_counter_init(&memcg->res, NULL);
>    res_counter_init(&memcg->memsw, NULL);
> +  res_counter_init(&memcg->kmem, NULL);
>    /*
>     * Deeper hierachy with use_hierarchy == false doesn't make
>     * much sense so let cgroup subsystem know about this
> --
> 1.7.11.7
>
> --
> To unsubscribe, send a message with 'unsubscribe linux-mm' in
> the body to majordomo@kvack.org.  For more info on Linux MM,
> see: http://www.linux-mm.org/ .
> Don't email: <a href=mailto:"dont@kvack.org"> email@kvack.org </a>

--
Michal Hocko
SUSE Labs
```

Subject: Re: [PATCH v5 05/14] Add a __GFP_KMEMCG flag
Posted by Michal Hocko on Tue, 16 Oct 2012 12:15:58 GMT
View Forum Message <> Reply to Message

On Tue 16-10-12 14:16:42, Glauber Costa wrote:
> This flag is used to indicate to the callees that this allocation is a
> kernel allocation in process context, and should be accounted to
> current's memcg. It takes numerical place of the of the recently removed
> __GFP_NO_KSWAPD.
>
> [v4: make flag unconditional, also declare it in trace code]
>
> Signed-off-by: Glauber Costa <glommer@parallels.com>
> Acked-by: Johannes Weiner <hannes@cmpxchg.org>
> Acked-by: Rik van Riel <riel@redhat.com>
> Acked-by: Mel Gorman <mel@csn.ul.ie>
> Acked-by: Kamezawa Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com>
> CC: Christoph Lameter <cl@linux.com>
> CC: Pekka Enberg <penberg@cs.helsinki.fi>
> CC: Michal Hocko <mhocko@suse.cz>
> CC: Suleiman Souhlal <suleiman@google.com>
> CC: Tejun Heo <tj@kernel.org>

I thought I have acked the patch already

Acked-by: Michal Hocko <mhocko@suse.cz>

> ---
> include/linux/gfp.h | 3 ++-
> include/trace/events/gfpflags.h | 1 +
> 2 files changed, 3 insertions(+), 1 deletion(-)
>
> diff --git a/include/linux/gfp.h b/include/linux/gfp.h
> index 02c1c97..9289d46 100644
> --- a/include/linux/gfp.h
> +++ b/include/linux/gfp.h
> @@ -31,6 +31,7 @@ struct vm_area_struct;
> #define ___GFP_THISNODE 0x40000u
> #define ___GFP_RECLAIMABLE 0x80000u
> #define ___GFP_NOTRACK 0x200000u
> +#define ___GFP_KMEMCG 0x400000u
> #define ___GFP_OTHER_NODE 0x800000u
> #define ___GFP_WRITE 0x1000000u
>
> @@ -87,7 +88,7 @@ struct vm_area_struct;
>
> #define __GFP_OTHER_NODE ((__force gfp_t)___GFP_OTHER_NODE) /* On behalf of other node */
> #define __GFP_WRITE ((__force gfp_t)___GFP_WRITE) /* Allocator intends to dirty page */

> -
> +#define __GFP_KMEMCG ((__force gfp_t)___GFP_KMEMCG) /* Allocation comes from a memcg-accounted resource */
> /*
>   * This may seem redundant, but it's a way of annotating false positives vs.
>   * allocations that simply cannot be supported (e.g. page tables).
> diff --git a/include/trace/events/gfpflags.h b/include/trace/events/gfpflags.h
> index 9391706..730df12 100644
> --- a/include/trace/events/gfpflags.h
> +++ b/include/trace/events/gfpflags.h
> @@ -36,6 +36,7 @@
> {(unsigned long)__GFP_RECLAIMABLE, "GFP_RECLAIMABLE"}, \
> {(unsigned long)__GFP_MOVABLE, "GFP_MOVABLE"}, \
> {(unsigned long)__GFP_NOTRACK, "GFP_NOTRACK"}, \
> + {(unsigned long)__GFP_KMEMCG, "GFP_KMEMCG"}, \
> {(unsigned long)__GFP_OTHER_NODE, "GFP_OTHER_NODE"} \
>) : "GFP_NOWAIT"
>
> --
> 1.7.11.7
>
> --
> To unsubscribe, send a message with 'unsubscribe linux-mm' in
> the body to majordomo@kvack.org. For more info on Linux MM,
> see: http://www.linux-mm.org/ .
> Don't email: email@kvack.org

--
Michal Hocko
SUSE Labs

Subject: Re: [PATCH v5 14/14] Add documentation about the kmem controller
Posted by Michal Hocko on Tue, 16 Oct 2012 12:23:24 GMT
View Forum Message <> Reply to Message

On Tue 16-10-12 14:16:51, Glauber Costa wrote:
> Signed-off-by: Glauber Costa <glommer@parallels.com>
> CC: Frederic Weisbecker <fweisbec@redhat.com>
> CC: Kamezawa Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com>
> CC: Michal Hocko <mhocko@suse.cz>
> CC: Christoph Lameter <cl@linux.com>
> CC: Pekka Enberg <penberg@cs.helsinki.fi>
> CC: Johannes Weiner <hannes@cmpxchg.org>
> CC: Suleiman Souhlal <suleiman@google.com>
> CC: Tejun Heo <tj@kernel.org>

Acked-by: Michal Hocko <mhocko@suse.cz

Just a nit..
> ---
> Documentation/cgroups/memory.txt | 58 ++++++++++++++++++++++++++++++++++++++-
> 1 file changed, 57 insertions(+), 1 deletion(-)
>
> diff --git a/Documentation/cgroups/memory.txt b/Documentation/cgroups/memory.txt
> index c07f7b4..dd15be8 100644
> --- a/Documentation/cgroups/memory.txt
> +++ b/Documentation/cgroups/memory.txt
[...]
> @@ -268,20 +273,65 @@ the amount of kernel memory used by the system. Kernel memory is fundamentally
> different than user memory, since it can't be swapped out, which makes it
> possible to DoS the system by consuming too much of this precious resource.
>
> +Kernel memory won't be accounted at all until limit on a group is set. This
> +allows for existing setups to continue working without disruption. The limit
> +cannot be set if the cgroup have children, or if there are already tasks in the
> +cgroup. When use_hierarchy == 1 and a group is accounted, its children will
> +automatically be accounted regardless of their limit value.
> +
> +After a controller is first limited, it will be kept being accounted until it

s/controller/group/

> +is removed. The memory limitation itself, can of course be removed by writing
> +-1 to memory.kmem.limit_in_bytes. In this case, kmem will be accounted, but not
> +limited.
> +

--
Michal Hocko
SUSE Labs

Subject: Re: [PATCH v5 07/14] mm: Allocate kernel pages to the right memcg
Posted by Christoph Lameter on Tue, 16 Oct 2012 15:31:13 GMT
View Forum Message <> Reply to Message

On Tue, 16 Oct 2012, Glauber Costa wrote:

> To avoid adding markers to the page - and a kmem flag that would
> necessarily follow, as much as doing page_cgroup lookups for no reason,
> whoever is marking its allocations with __GFP_KMEMCG flag is responsible
> for telling the page allocator that this is such an allocation at
> free_pages() time. This is done by the invocation of
> __free_accounted_pages() and free_accounted_pages().

Hmmm... The code paths to free pages are often shared between multiple
subsystems. Are you sure that this is actually working and accurately
tracks the MEMCG pages?

> +/*
> + * __free_accounted_pages and free_accounted_pages will free pages allocated
> + * with __GFP_KMEMCG.
> + *
> + * Those pages are accounted to a particular memcg, embedded in the
> + * corresponding page_cgroup. To avoid adding a hit in the allocator to search
> + * for that information only to find out that it is NULL for users who have no
> + * interest in that whatsoever, we provide these functions.
> + *
> + * The caller knows better which flags it relies on.
> + */
> +void __free_accounted_pages(struct page *page, unsigned int order)
> +{
> + memcg_kmem_uncharge_page(page, order);
> + __free_pages(page, order);
> +}

If we already are introducing such an API: Could it not be made more
general so that it can also be used in the future to communicate other
characteristics of a page on free?

Subject: Re: [PATCH v5 14/14] Add documentation about the kmem controller
Posted by Christoph Lameter on Tue, 16 Oct 2012 18:25:06 GMT
View Forum Message <> Reply to Message

On Tue, 16 Oct 2012, Glauber Costa wrote:

>
> + memory.kmem.limit_in_bytes   # set/show hard limit for kernel memory
> + memory.kmem.usage_in_bytes   # show current kernel memory allocation
> + memory.kmem.failcnt   # show the number of kernel memory usage hits limits
> + memory.kmem.max_usage_in_bytes # show max kernel memory usage recorded

Does it actually make sense to limit kernel memory? The user generally has
no idea how much kernel memory a process is using and kernel changes can
change the memory footprint. Given the fuzzy accounting in the kernel a
large cache refill (if someone configures the slab batch count to be
really big f.e.) can account a lot of memory to the wrong cgroup. The
allocation could fail.

Limiting the total memory use of a process (U+K) would make more sense I
guess. Only U is probably sufficient? In what way would a limitation on

kernel memory in use be good?

Subject: Re: [PATCH v5 07/14] mm: Allocate kernel pages to the right memcg
Posted by Glauber Costa on Tue, 16 Oct 2012 18:55:30 GMT
View Forum Message <> Reply to Message

On 10/16/2012 07:31 PM, Christoph Lameter wrote:
> On Tue, 16 Oct 2012, Glauber Costa wrote:
>
>> To avoid adding markers to the page - and a kmem flag that would
>> necessarily follow, as much as doing page_cgroup lookups for no reason,
>> whoever is marking its allocations with __GFP_KMEMCG flag is responsible
>> for telling the page allocator that this is such an allocation at
>> free_pages() time. This is done by the invocation of
>> __free_accounted_pages() and free_accounted_pages().
>
> Hmmm... The code paths to free pages are often shared between multiple
> subsystems. Are you sure that this is actually working and accurately
> tracks the MEMCG pages?
>

As described above, only call sites that are switched to
free_accounted_pages are affected. There are very few of them. The stack
case is particularly easy to test: every time a process appears, usage
is increased in 8k. Every time a process dies, usage decreases by 8k.

In my other patchseries, I include the object allocators into this. So
again: there are very few call sites actually being patched.

>> +/*
>> + * __free_accounted_pages and free_accounted_pages will free pages allocated
>> + * with __GFP_KMEMCG.
>> + *
>> + * Those pages are accounted to a particular memcg, embedded in the
>> + * corresponding page_cgroup. To avoid adding a hit in the allocator to search
>> + * for that information only to find out that it is NULL for users who have no
>> + * interest in that whatsoever, we provide these functions.
>> + *
>> + * The caller knows better which flags it relies on.
>> + */
>> +void __free_accounted_pages(struct page *page, unsigned int order)
>> +{
>> + memcg_kmem_uncharge_page(page, order);
>> + __free_pages(page, order);
>> +}
>

> If we already are introducing such an API: Could it not be made more
> general so that it can also be used in the future to communicate other
> characteristics of a page on free?
>

I guess so. Which other use case do you have in mind?
In any case, I don't see this as a blocker to this patchset. There is no
reason why it can't be done should the need arise.

Subject: Re: [PATCH v5 14/14] Add documentation about the kmem controller
Posted by Glauber Costa on Tue, 16 Oct 2012 19:02:46 GMT
View Forum Message <> Reply to Message

On 10/16/2012 10:25 PM, Christoph Lameter wrote:
> On Tue, 16 Oct 2012, Glauber Costa wrote:
>
>>
>> + memory.kmem.limit_in_bytes   # set/show hard limit for kernel memory
>> + memory.kmem.usage_in_bytes   # show current kernel memory allocation
>> + memory.kmem.failcnt   # show the number of kernel memory usage hits limits
>> + memory.kmem.max_usage_in_bytes # show max kernel memory usage recorded
>
> Does it actually make sense to limit kernel memory?

Yes.

> The user generally has
> no idea how much kernel memory a process is using and kernel changes can
> change the memory footprint. Given the fuzzy accounting in the kernel a
> large cache refill (if someone configures the slab batch count to be
> really big f.e.) can account a lot of memory to the wrong cgroup. The
> allocation could fail.
>

It heavily depends on the type of the user. The user may not know how
much kernel memory precisely will be used, but he/she usually knows
quite well that it shouldn't be all cgroups together shouldn't use more
than available in the system.

IOW: It is usually safe to overcommit user memory, but not kernel
memory. This is absolutely crucial in any high-density container host,
and we've been doing this in OpenVZ for ages (in an uglier form than this)

> Limiting the total memory use of a process (U+K) would make more sense I
> guess. Only U is probably sufficient? In what way would a limitation on
> kernel memory in use be good?
>

The kmem counter is also fed into the u counter. If the limit value of
"u" is equal or greater than "k", this is actually what you are doing.

For a lot of application yes, only U is sufficient. This is the default,
btw, since "k" is only even accounted if you set the limit.

All those use cases are detailed a bit below in this file.

A limitation of kernel memory use would be good, for example, to prevent
abuse from non-trusted containers in a high density, shared, container
environment.

Subject: Re: [PATCH v5 14/14] Add documentation about the kmem controller
Posted by Christoph Lameter on Tue, 16 Oct 2012 19:30:23 GMT
View Forum Message <> Reply to Message

On Tue, 16 Oct 2012, Glauber Costa wrote:

> A limitation of kernel memory use would be good, for example, to prevent
> abuse from non-trusted containers in a high density, shared, container
> environment.

But that would be against intentional abuse by someone who has code that
causes the kernel to use a lot of memory on its behalf. We already need
protection from that without memcg.

Subject: Re: [PATCH v5 06/14] memcg: kmem controller infrastructure
Posted by KAMEZAWA Hiroyuki on Wed, 17 Oct 2012 06:40:46 GMT
View Forum Message <> Reply to Message

(2012/10/16 19:16), Glauber Costa wrote:
> This patch introduces infrastructure for tracking kernel memory pages to
> a given memcg. This will happen whenever the caller includes the flag
> __GFP_KMEMCG flag, and the task belong to a memcg other than the root.
>
> In memcontrol.h those functions are wrapped in inline acessors. The
> idea is to later on, patch those with static branches, so we don't incur
> any overhead when no mem cgroups with limited kmem are being used.
>
> Users of this functionality shall interact with the memcg core code
> through the following functions:
>
> memcg_kmem_newpage_charge: will return true if the group can handle the
>   allocation. At this point, struct page is not

> yet allocated.
>
> memcg_kmem_commit_charge: will either revert the charge, if struct page
> allocation failed, or embed memcg information
> into page_cgroup.
>
> memcg_kmem_uncharge_page: called at free time, will revert the charge.
>
> [v2: improved comments and standardized function names]
> [v3: handle no longer opaque, functions not exported,
> even more comments]
> [v4: reworked Used bit handling and surroundings for more clarity]
> [v5: simplified code for kmemcg compiled out and core functions in
> memcontrol.c, moved kmem code to the middle to avoid forward decls]
>
> Signed-off-by: Glauber Costa <glommer@parallels.com>
> Acked-by: Michal Hocko <mhocko@suse.cz>
> CC: Christoph Lameter <cl@linux.com>
> CC: Pekka Enberg <penberg@cs.helsinki.fi>
> CC: Kamezawa Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com>
> CC: Johannes Weiner <hannes@cmpxchg.org>
> CC: Tejun Heo <tj@kernel.org>

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

Subject: Re: [PATCH v5 03/14] memcg: change defines to an enum
Posted by David Rientjes on Wed, 17 Oct 2012 21:50:17 GMT
View Forum Message <> Reply to Message

On Tue, 16 Oct 2012, Glauber Costa wrote:

> This is just a cleanup patch for clarity of expression. In earlier
> submissions, people asked it to be in a separate patch, so here it is.
>
> [v2: use named enum as type throughout the file as well]
>
> Signed-off-by: Glauber Costa <glommer@parallels.com>
> Acked-by: Kamezawa Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com>
> Acked-by: Michal Hocko <mhocko@suse.cz>
> Acked-by: Johannes Weiner <hannes@cmpxchg.org>
> CC: Tejun Heo <tj@kernel.org>

Acked-by: David Rientjes <rientjes@google.com>

Subject: Re: [PATCH v5 04/14] kmem accounting basic infrastructure

Posted by David Rientjes on Wed, 17 Oct 2012 22:08:04 GMT
View Forum Message <> Reply to Message

On Tue, 16 Oct 2012, Glauber Costa wrote:

> This patch adds the basic infrastructure for the accounting of kernel
> memory. To control that, the following files are created:
>
> * memory.kmem.usage_in_bytes
> * memory.kmem.limit_in_bytes
> * memory.kmem.failcnt
> * memory.kmem.max_usage_in_bytes
>
> They have the same meaning of their user memory counterparts. They
> reflect the state of the "kmem" res_counter.
>
> Per cgroup kmem memory accounting is not enabled until a limit is set
> for the group. Once the limit is set the accounting cannot be disabled
> for that group. This means that after the patch is applied, no
> behavioral changes exists for whoever is still using memcg to control
> their memory usage, until memory.kmem.limit_in_bytes is set for the
> first time.
>
> We always account to both user and kernel resource_counters. This
> effectively means that an independent kernel limit is in place when the
> limit is set to a lower value than the user memory. A equal or higher
> value means that the user limit will always hit first, meaning that kmem
> is effectively unlimited.
>
> People who want to track kernel memory but not limit it, can set this
> limit to a very high number (like RESOURCE_MAX - 1page - that no one
> will ever hit, or equal to the user memory)
>
> [v4: make kmem files part of the main array;
> do not allow limit to be set for non-empty cgroups]
> [v5: cosmetic changes]
>
> Signed-off-by: Glauber Costa <glommer@parallels.com>
> Acked-by: Kamezawa Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com>
> CC: Michal Hocko <mhocko@suse.cz>
> CC: Johannes Weiner <hannes@cmpxchg.org>
> CC: Tejun Heo <tj@kernel.org>
> ---
> mm/memcontrol.c | 116
++-
> 1 file changed, 115 insertions(+), 1 deletion(-)
>
> diff --git a/mm/memcontrol.c b/mm/memcontrol.c
> index 71d259e..30eafeb 100644

> --- a/mm/memcontrol.c
> +++ b/mm/memcontrol.c
> @@ -266,6 +266,10 @@ struct mem_cgroup {
> };
>
> /*
> + * the counter to account for kernel memory usage.
> + */
> + struct res_counter kmem;
> + /*
>   * Per cgroup active and inactive list, similar to the
>   * per zone LRU lists.
>   */
> @@ -280,6 +284,7 @@ struct mem_cgroup {
>   * Should the accounting and control be hierarchical, per subtree?
>   */
> bool use_hierarchy;
> + unsigned long kmem_accounted; /* See KMEM_ACCOUNTED_*, below */

I think this should be named kmem_account_flags or kmem_flags, otherwise
it appears that this is the actual account.

>
> bool oom_lock;
> atomic_t under_oom;
> @@ -332,6 +337,20 @@ struct mem_cgroup {
> #endif
> };
>
> +/* internal only representation about the status of kmem accounting. */
> +enum {
> + KMEM_ACCOUNTED_ACTIVE = 0, /* accounted by this cgroup itself */
> +};
> +
> +#define KMEM_ACCOUNTED_MASK (1 << KMEM_ACCOUNTED_ACTIVE)
> +
> +#ifdef CONFIG_MEMCG_KMEM

memcg->kmem_accounted isn't only defined for this configuration, so would
it be simpler to define this unconditionally?

> +static void memcg_kmem_set_active(struct mem_cgroup *memcg)

inline?

> +{
> + set_bit(KMEM_ACCOUNTED_ACTIVE, &memcg->kmem_accounted);
> +}

> +#endif
> +
> /* Stuffs for move charges at task migration. */
> /*
>   * Types of charges to be moved. "move_charge_at_immitgrate" is treated as a
> @@ -390,6 +409,7 @@ enum res_type {
>   _MEM,
>   _MEMSWAP,
>   _OOM_TYPE,
> + _KMEM,
> };
>
> #define MEMFILE_PRIVATE(x, val) ((x) << 16 | (val))
> @@ -1433,6 +1453,10 @@ done:
>   res_counter_read_u64(&memcg->memsw, RES_USAGE) >> 10,
>   res_counter_read_u64(&memcg->memsw, RES_LIMIT) >> 10,
>   res_counter_read_u64(&memcg->memsw, RES_FAILCNT));
> + printk(KERN_INFO "kmem: usage %llukB, limit %llukB, failcnt %llu\n",
> + res_counter_read_u64(&memcg->kmem, RES_USAGE) >> 10,
> + res_counter_read_u64(&memcg->kmem, RES_LIMIT) >> 10,
> + res_counter_read_u64(&memcg->kmem, RES_FAILCNT));
> }
>
> /*
> @@ -3940,6 +3964,9 @@ static ssize_t mem_cgroup_read(struct cgroup *cont, struct cftype
*cft,
>   else
>   val = res_counter_read_u64(&memcg->memsw, name);
>   break;
> + case _KMEM:
> + val = res_counter_read_u64(&memcg->kmem, name);
> + break;
>   default:
>   BUG();
>   }
> @@ -3947,6 +3974,57 @@ static ssize_t mem_cgroup_read(struct cgroup *cont, struct cftype
*cft,
>   len = scnprintf(str, sizeof(str), "%llu\n", (unsigned long long)val);
>   return simple_read_from_buffer(buf, nbytes, ppos, str, len);
> }
> +
> +static int memcg_update_kmem_limit(struct cgroup *cont, u64 val)
> +{
> + int ret = -EINVAL;
> +#ifdef CONFIG_MEMCG_KMEM
> + struct mem_cgroup *memcg = mem_cgroup_from_cont(cont);
> + /*
> + * For simplicity, we won't allow this to be disabled. It also can't

> + * be changed if the cgroup has children already, or if tasks had
> + * already joined.
> + *
> + * If tasks join before we set the limit, a person looking at
> + * kmem.usage_in_bytes will have no way to determine when it took
> + * place, which makes the value quite meaningless.
> + *
> + * After it first became limited, changes in the value of the limit are
> + * of course permitted.
> + *
> + * Taking the cgroup_lock is really offensive, but it is so far the only
> + * way to guarantee that no children will appear. There are plenty of
> + * other offenders, and they should all go away. Fine grained locking
> + * is probably the way to go here. When we are fully hierarchical, we
> + * can also get rid of the use_hierarchy check.

Not sure it's so offensive, it's a pretty standard way of ensuring that
cont->children doesn't get manipulated in a race.

> + */
> + cgroup_lock();
> + mutex_lock(&set_limit_mutex);
> + if (!memcg->kmem_accounted && val != RESOURCE_MAX) {
> + if (cgroup_task_count(cont) || (memcg->use_hierarchy &&
> +   !list_empty(&cont->children))) {
> +   ret = -EBUSY;
> +   goto out;
> + }
> + ret = res_counter_set_limit(&memcg->kmem, val);
> + VM_BUG_ON(ret);
> +
> + memcg_kmem_set_active(memcg);
> + } else
> + ret = res_counter_set_limit(&memcg->kmem, val);
> +out:
> + mutex_unlock(&set_limit_mutex);
> + cgroup_unlock();
> +#endif
> + return ret;
> +}
> +
> +static void memcg_propagate_kmem(struct mem_cgroup *memcg,
> +   struct mem_cgroup *parent)
> +{
> + memcg->kmem_accounted = parent->kmem_accounted;
> +}
> +
> /*

>   * The user of this function is...
>   * RES_LIMIT.
> @@ -3978,8 +4056,12 @@ static int mem_cgroup_write(struct cgroup *cont, struct cftype *cft,
>   break;
>   if (type == _MEM)
>   ret = mem_cgroup_resize_limit(memcg, val);
> - else
> + else if (type == _MEMSWAP)
>   ret = mem_cgroup_resize_memsw_limit(memcg, val);
> + else if (type == _KMEM)
> +   ret = memcg_update_kmem_limit(cont, val);
> + else
> +   return -EINVAL;

I like how this is done in a maintainable way to ensure no other types can
inadvertently update the memsw limit as it was previously written. All
other returns of -EINVAL just cause the switch statement to break, though,
rather than return directly.

>   break;
>   case RES_SOFT_LIMIT:
>   ret = res_counter_memparse_write_strategy(buffer, &val);
> @@ -4045,12 +4127,16 @@ static int mem_cgroup_reset(struct cgroup *cont, unsigned int
event)
>   case RES_MAX_USAGE:
>   if (type == _MEM)
>   res_counter_reset_max(&memcg->res);
> + else if (type == _KMEM)
> +   res_counter_reset_max(&memcg->kmem);

Could this be written in the same way above, i.e. check _MEMSWAP to pass
memcg->memsw, _KMEM for memcg->kmem, etc?

>   else
>   res_counter_reset_max(&memcg->memsw);
>   break;
>   case RES_FAILCNT:
>   if (type == _MEM)
>   res_counter_reset_failcnt(&memcg->res);
> + else if (type == _KMEM)
> +   res_counter_reset_failcnt(&memcg->kmem);

Same.

Subject: Re: [PATCH v5 05/14] Add a __GFP_KMEMCG flag
Posted by David Rientjes on Wed, 17 Oct 2012 22:09:04 GMT

On Tue, 16 Oct 2012, Glauber Costa wrote:

> This flag is used to indicate to the callees that this allocation is a
> kernel allocation in process context, and should be accounted to
> current's memcg. It takes numerical place of the of the recently removed
> __GFP_NO_KSWAPD.
>
> [v4: make flag unconditional, also declare it in trace code]
>
> Signed-off-by: Glauber Costa <glommer@parallels.com>
> Acked-by: Johannes Weiner <hannes@cmpxchg.org>
> Acked-by: Rik van Riel <riel@redhat.com>
> Acked-by: Mel Gorman <mel@csn.ul.ie>
> Acked-by: Kamezawa Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com>
> CC: Christoph Lameter <cl@linux.com>
> CC: Pekka Enberg <penberg@cs.helsinki.fi>
> CC: Michal Hocko <mhocko@suse.cz>
> CC: Suleiman Souhlal <suleiman@google.com>
> CC: Tejun Heo <tj@kernel.org>

Acked-by: David Rientjes <rientjes@google.com>

Subject: Re: [PATCH v5 00/14] kmem controller for memcg.
Posted by akpm on Wed, 17 Oct 2012 22:11:42 GMT

On Tue, 16 Oct 2012 14:16:37 +0400
Glauber Costa <glommer@parallels.com> wrote:

> ...
>
> A general explanation of what this is all about follows:
>
> The kernel memory limitation mechanism for memcg concerns itself with
> disallowing potentially non-reclaimable allocations to happen in exaggerate
> quantities by a particular set of processes (cgroup). Those allocations could
> create pressure that affects the behavior of a different and unrelated set of
> processes.
>
> Its basic working mechanism is to annotate some allocations with the
> _GFP_KMEMCG flag. When this flag is set, the current process allocating will
> have its memcg identified and charged against. When reaching a specific limit,
> further allocations will be denied.

The need to set _GFP_KMEMCG is rather unpleasing, and makes one wonder

"why didn't it just track all allocations".

Does this mean that over time we can expect more sites to get the
_GFP_KMEMCG tagging? If so, are there any special implications, or do
we just go in, do the one-line patch and expect everything to work? If
so, why don't we go in and do that tagging right now?

And how *accurate* is the proposed code? What percentage of kernel
memory allocations are unaccounted, typical case and worst case?

All sorts of questions come to mind over this decision, but it was
unexplained. It should be, please. A lot!

>
> ...
>
> Limits lower than
> the user limit effectively means there is a separate kernel memory limit that
> may be reached independently than the user limit. Values equal or greater than
> the user limit implies only that kernel memory is tracked. This provides a
> unified vision of "maximum memory", be it kernel or user memory.
>

I'm struggling to understand that text much at all. Reading the
Documentation/cgroups/memory.txt patch helped.

Subject: Re: [PATCH v5 01/14] memcg: Make it possible to use the stock for more than one page.
Posted by akpm on Wed, 17 Oct 2012 22:11:55 GMT
View Forum Message <> Reply to Message

On Tue, 16 Oct 2012 14:16:38 +0400
Glauber Costa <glommer@parallels.com> wrote:

> From: Suleiman Souhlal <ssouhlal@FreeBSD.org>
>
> We currently have a percpu stock cache scheme that charges one page at a
> time from memcg->res, the user counter. When the kernel memory
> controller comes into play, we'll need to charge more than that.
>
> This is because kernel memory allocations will also draw from the user
> counter, and can be bigger than a single page, as it is the case with
> the stack (usually 2 pages) or some higher order slabs.
>
> ...
>
> -/*

> - * Try to consume stocked charge on this cpu. If success, one page is consumed
> - * from local stock and true is returned. If the stock is 0 or charges from a
> - * cgroup which is not current target, returns false. This stock will be
> - * refilled.
> +/**
> + * consume_stock: Try to consume stocked charge on this cpu.
> + * @memcg: memcg to consume from.
> + * @nr_pages: how many pages to charge.
> + *
> + * The charges will only happen if @memcg matches the current cpu's memcg
> + * stock, and at least @nr_pages are available in that stock. Failure to
> + * service an allocation will refill the stock.
> + *
> + * returns true if succesfull, false otherwise.

spello.

>   */
> -static bool consume_stock(struct mem_cgroup *memcg)
> +static bool consume_stock(struct mem_cgroup *memcg, int nr_pages)

I don't believe there is a case for nr_pages < 0 here? If not then I
suggest that it would be clearer to use an unsigned type, like
memcg_stock_pcp.stock.

>
> ...
>

Subject: Re: [PATCH v5 04/14] kmem accounting basic infrastructure
Posted by akpm on Wed, 17 Oct 2012 22:12:07 GMT
View Forum Message <> Reply to Message

On Tue, 16 Oct 2012 14:16:41 +0400
Glauber Costa <glommer@parallels.com> wrote:

> This patch adds the basic infrastructure for the accounting of kernel
> memory. To control that, the following files are created:
>
> * memory.kmem.usage_in_bytes
> * memory.kmem.limit_in_bytes
> * memory.kmem.failcnt

gargh. "failcnt" is not a word. Who was it who first thought that
omitting voewls from words improves anything?

Sigh. That pooch is already screwed and there's nothing we can do

about it now.

> * memory.kmem.max_usage_in_bytes
>
> They have the same meaning of their user memory counterparts. They
> reflect the state of the "kmem" res_counter.
>
> Per cgroup kmem memory accounting is not enabled until a limit is set
> for the group. Once the limit is set the accounting cannot be disabled
> for that group. This means that after the patch is applied, no
> behavioral changes exists for whoever is still using memcg to control
> their memory usage, until memory.kmem.limit_in_bytes is set for the
> first time.
>
> We always account to both user and kernel resource_counters. This
> effectively means that an independent kernel limit is in place when the
> limit is set to a lower value than the user memory. A equal or higher
> value means that the user limit will always hit first, meaning that kmem
> is effectively unlimited.
>
> People who want to track kernel memory but not limit it, can set this
> limit to a very high number (like RESOURCE_MAX - 1page - that no one
> will ever hit, or equal to the user memory)
>
>
> ...
>
> +/* internal only representation about the status of kmem accounting. */
> +enum {
> + KMEM_ACCOUNTED_ACTIVE = 0, /* accounted by this cgroup itself */
> +};
> +
> +#define KMEM_ACCOUNTED_MASK (1 << KMEM_ACCOUNTED_ACTIVE)
> +
> +#ifdef CONFIG_MEMCG_KMEM
> +static void memcg_kmem_set_active(struct mem_cgroup *memcg)
> +{
> + set_bit(KMEM_ACCOUNTED_ACTIVE, &memcg->kmem_accounted);
> +}
> +#endif

I don't think memcg_kmem_set_active() really needs to exist. It has a
single caller and is unlikely to get any additional callers, so just
open-code it there?

Subject: Re: [PATCH v5 06/14] memcg: kmem controller infrastructure

On Tue, 16 Oct 2012 14:16:43 +0400
Glauber Costa <glommer@parallels.com> wrote:

> This patch introduces infrastructure for tracking kernel memory pages to
> a given memcg. This will happen whenever the caller includes the flag
> __GFP_KMEMCG flag, and the task belong to a memcg other than the root.
>
> In memcontrol.h those functions are wrapped in inline acessors. The
> idea is to later on, patch those with static branches, so we don't incur
> any overhead when no mem cgroups with limited kmem are being used.
>
> Users of this functionality shall interact with the memcg core code
> through the following functions:
>
> memcg_kmem_newpage_charge: will return true if the group can handle the
>   allocation. At this point, struct page is not
>   yet allocated.
>
> memcg_kmem_commit_charge: will either revert the charge, if struct page
>   allocation failed, or embed memcg information
>   into page_cgroup.
>
> memcg_kmem_uncharge_page: called at free time, will revert the charge.
>
> ...
>
> +static __always_inline bool
> +memcg_kmem_newpage_charge(gfp_t gfp, struct mem_cgroup **memcg, int order)
> +{
> + if (!memcg_kmem_enabled())
> + return true;
> +
> + /*
> + * __GFP_NOFAIL allocations will move on even if charging is not
> + * possible. Therefore we don't even try, and have this allocation
> + * unaccounted. We could in theory charge it with
> + * res_counter_charge_nofail, but we hope those allocations are rare,
> + * and won't be worth the trouble.
> + */
> + if (!(gfp & __GFP_KMEMCG) || (gfp & __GFP_NOFAIL))
> + return true;
> + if (in_interrupt() || (!current->mm) || (current->flags & PF_KTHREAD))
> + return true;
> +
> + /* If the test is dying, just let it go. */
> +   if (unlikely(test_thread_flag(TIF_MEMDIE)

> + || fatal_signal_pending(current)))
> + return true;
> +
> + return __memcg_kmem_newpage_charge(gfp, memcg, order);
> +}

That's a big function! Why was it __always_inline? I'd have thought
it would be better to move the code after memcg_kmem_enabled() out of
line.

Do we actually need to test PF_KTHREAD when current->mm == NULL?
Perhaps because of aio threads whcih temporarily adopt a userspace mm?

> +/**
> + * memcg_kmem_uncharge_page: uncharge pages from memcg
> + * @page: pointer to struct page being freed
> + * @order: allocation order.
> + *
> + * there is no need to specify memcg here, since it is embedded in page_cgroup
> + */
> +static __always_inline void
> +memcg_kmem_uncharge_page(struct page *page, int order)
> +{
> + if (memcg_kmem_enabled())
> + __memcg_kmem_uncharge_page(page, order);
> +}
> +
> +/**
> + * memcg_kmem_commit_charge: embeds correct memcg in a page
> + * @page: pointer to struct page recently allocated
> + * @memcg: the memcg structure we charged against
> + * @order: allocation order.
> + *
> + * Needs to be called after memcg_kmem_newpage_charge, regardless of success or
> + * failure of the allocation. if @page is NULL, this function will revert the
> + * charges. Otherwise, it will commit the memcg given by @memcg to the
> + * corresponding page_cgroup.
> + */
> +static __always_inline void
> +memcg_kmem_commit_charge(struct page *page, struct mem_cgroup *memcg, int order)
> +{
> + if (memcg_kmem_enabled() && memcg)
> + __memcg_kmem_commit_charge(page, memcg, order);
> +}

I suspect the __always_inline's here are to do with static branch
trickery. A code comment is warranted if so?

Subject: Re: [PATCH v5 07/14] mm: Allocate kernel pages to the right memcg
Posted by akpm on Wed, 17 Oct 2012 22:12:21 GMT
View Forum Message <> Reply to Message

On Tue, 16 Oct 2012 14:16:44 +0400
Glauber Costa <glommer@parallels.com> wrote:

> When a process tries to allocate a page with the __GFP_KMEMCG flag, the
> page allocator will call the corresponding memcg functions to validate
> the allocation. Tasks in the root memcg can always proceed.
>
> To avoid adding markers to the page - and a kmem flag that would
> necessarily follow, as much as doing page_cgroup lookups for no reason,
> whoever is marking its allocations with __GFP_KMEMCG flag is responsible
> for telling the page allocator that this is such an allocation at
> free_pages() time.

Well, why? Was that the correct decision?

> This is done by the invocation of
> __free_accounted_pages() and free_accounted_pages().

These are very general-sounding names. I'd expect the identifiers to
contain "memcg" and/or "kmem", to identify what's going on.

Subject: Re: [PATCH v5 11/14] memcg: allow a memcg with kmem charges to be destructed.
Posted by akpm on Wed, 17 Oct 2012 22:12:35 GMT
View Forum Message <> Reply to Message

On Tue, 16 Oct 2012 14:16:48 +0400
Glauber Costa <glommer@parallels.com> wrote:

> Because the ultimate goal of the kmem tracking in memcg is to track slab
> pages as well,

It is? For a major patchset such as this, it's pretty important to
discuss such long-term plans in the top-level discussion. Covering
things such as expected complexity, expected performance hit, how these
plans affected the current implementation, etc.

The main reason for this is that if the future plans appear to be of
doubtful feasibility and the current implementation isn't sufficiently
useful without the future stuff, we shouldn't merge the current
implementation. It's a big issue!

> we can't guarantee that we'll always be able to point a

> page to a particular process, and migrate the charges along with it -
> since in the common case, a page will contain data belonging to multiple
> processes.
>
> Because of that, when we destroy a memcg, we only make sure the
> destruction will succeed by discounting the kmem charges from the user
> charges when we try to empty the cgroup.
>
> ...
>

Subject: Re: [PATCH v5 13/14] protect architectures where THREAD_SIZE >= PAGE_SIZE against fork bombs
Posted by akpm on Wed, 17 Oct 2012 22:12:45 GMT
View Forum Message <> Reply to Message

On Tue, 16 Oct 2012 14:16:50 +0400
Glauber Costa <glommer@parallels.com> wrote:

> @@ -146,7 +146,7 @@ void __weak arch_release_thread_info(struct thread_info *ti)
> static struct thread_info *alloc_thread_info_node(struct task_struct *tsk,
> int node)
> {
> - struct page *page = alloc_pages_node(node, THREADINFO_GFP,
> + struct page *page = alloc_pages_node(node, THREADINFO_GFP_ACCOUNTED,
> THREAD_SIZE_ORDER);

yay, we actually used all this code for something ;)

I don't think we really saw a comprehensive list of what else the kmem
controller will be used for, but I believe that all other envisaged
applications will require slab accounting, yes?

So it appears that all we have at present is a
yet-another-fork-bomb-preventer, but one which requires that the
culprit be in a container? That's reasonable, given your
hosted-environment scenario. It's unclear (to me) that we should merge
all this code for only this feature. Again, it would be good to have a
clear listing of and plan for other applications of this code.

Subject: Re: [PATCH v5 14/14] Add documentation about the kmem controller
Posted by akpm on Wed, 17 Oct 2012 22:12:54 GMT
View Forum Message <> Reply to Message

On Tue, 16 Oct 2012 14:16:51 +0400
Glauber Costa <glommer@parallels.com> wrote:

> +Kernel memory won't be accounted at all until limit on a group is set. This
> +allows for existing setups to continue working without disruption. The limit
> +cannot be set if the cgroup have children, or if there are already tasks in the
> +cgroup.

What behaviour will usersapce see if "The limit cannot be set"?
write() returns -EINVAL, something like that?

Subject: Re: [PATCH v5 06/14] memcg: kmem controller infrastructure
Posted by David Rientjes on Wed, 17 Oct 2012 22:37:32 GMT
View Forum Message <> Reply to Message

On Tue, 16 Oct 2012, Glauber Costa wrote:

> diff --git a/include/linux/memcontrol.h b/include/linux/memcontrol.h
> index 8d9489f..303a456 100644
> --- a/include/linux/memcontrol.h
> +++ b/include/linux/memcontrol.h
> @@ -21,6 +21,7 @@
> #define _LINUX_MEMCONTROL_H
> #include <linux/cgroup.h>
> #include <linux/vm_event_item.h>
> +#include <linux/hardirq.h>
>
> struct mem_cgroup;
> struct page_cgroup;
> @@ -399,6 +400,88 @@ struct sock;
> #ifdef CONFIG_MEMCG_KMEM
> void sock_update_memcg(struct sock *sk);
> void sock_release_memcg(struct sock *sk);
> +
> +static inline bool memcg_kmem_enabled(void)
> +{
> + return true;
> +}
> +
> +bool __memcg_kmem_newpage_charge(gfp_t gfp, struct mem_cgroup **memcg,
> +   int order);
> +void __memcg_kmem_commit_charge(struct page *page,
> +   struct mem_cgroup *memcg, int order);
> +void __memcg_kmem_uncharge_page(struct page *page, int order);
> +
> +/**
> + * memcg_kmem_newpage_charge: verify if a new kmem allocation is allowed.

> + * @gfp: the gfp allocation flags.
> + * @memcg: a pointer to the memcg this was charged against.
> + * @order: allocation order.
> + *
> + * returns true if the memcg where the current task belongs can hold this
> + * allocation.
> + *
> + * We return true automatically if this allocation is not to be accounted to
> + * any memcg.
> + */
> +static __always_inline bool
> +memcg_kmem_newpage_charge(gfp_t gfp, struct mem_cgroup **memcg, int order)
> +{
> + if (!memcg_kmem_enabled())
> + return true;
> +
> + /*
> + * __GFP_NOFAIL allocations will move on even if charging is not
> + * possible. Therefore we don't even try, and have this allocation
> + * unaccounted. We could in theory charge it with
> + * res_counter_charge_nofail, but we hope those allocations are rare,
> + * and won't be worth the trouble.
> + */
> + if (!(gfp & __GFP_KMEMCG) || (gfp & __GFP_NOFAIL))
> + return true;
> + if (in_interrupt() || (!current->mm) || (current->flags & PF_KTHREAD))
> + return true;
> +
> + /* If the test is dying, just let it go. */
> + if (unlikely(test_thread_flag(TIF_MEMDIE)
> + || fatal_signal_pending(current)))
> + return true;

This can be simplified to just check fatal_signal_pending(), all threads
with TIF_MEMDIE also have a pending SIGKILL.

It also has whitespace damage.

> +
> + return __memcg_kmem_newpage_charge(gfp, memcg, order);
> +}
> +
> +/**
> + * memcg_kmem_uncharge_page: uncharge pages from memcg

Should be memcg_kmem_uncharge_pages() since it takes an order argument?

> + * @page: pointer to struct page being freed

> + * @order: allocation order.
> + *
> + * there is no need to specify memcg here, since it is embedded in page_cgroup
> + */
> +static __always_inline void
> +memcg_kmem_uncharge_page(struct page *page, int order)
> +{
> + if (memcg_kmem_enabled())
> + __memcg_kmem_uncharge_page(page, order);
> +}
> +
> +/**
> + * memcg_kmem_commit_charge: embeds correct memcg in a page
> + * @page: pointer to struct page recently allocated
> + * @memcg: the memcg structure we charged against
> + * @order: allocation order.
> + *
> + * Needs to be called after memcg_kmem_newpage_charge, regardless of success or
> + * failure of the allocation. if @page is NULL, this function will revert the
> + * charges. Otherwise, it will commit the memcg given by @memcg to the
> + * corresponding page_cgroup.
> + */
> +static __always_inline void
> +memcg_kmem_commit_charge(struct page *page, struct mem_cgroup *memcg, int order)
> +{
> + if (memcg_kmem_enabled() && memcg)
> + __memcg_kmem_commit_charge(page, memcg, order);
> +}
> +
> #else
> static inline void sock_update_memcg(struct sock *sk)
> {
> @@ -406,6 +489,21 @@ static inline void sock_update_memcg(struct sock *sk)
> static inline void sock_release_memcg(struct sock *sk)
> {
> }
> +
> +static inline bool
> +memcg_kmem_newpage_charge(gfp_t gfp, struct mem_cgroup **memcg, int order)
> +{
> + return true;
> +}
> +
> +static inline void memcg_kmem_uncharge_page(struct page *page, int order)

Two spaces.

> +{

> +}
> +
> +static inline void
> +memcg_kmem_commit_charge(struct page *page, struct mem_cgroup *memcg, int order)
> +{
> +}
> #endif /* CONFIG_MEMCG_KMEM */
> #endif /* _LINUX_MEMCONTROL_H */
>
> diff --git a/mm/memcontrol.c b/mm/memcontrol.c
> index 30eafeb..1182188 100644
> --- a/mm/memcontrol.c
> +++ b/mm/memcontrol.c
> @@ -10,6 +10,10 @@
>   * Copyright (C) 2009 Nokia Corporation
>   * Author: Kirill A. Shutemov
>   *
> + * Kernel Memory Controller
> + * Copyright (C) 2012 Parallels Inc. and Google Inc.
> + * Authors: Glauber Costa and Suleiman Souhlal
> + *
>   * This program is free software; you can redistribute it and/or modify
>   * it under the terms of the GNU General Public License as published by
>   * the Free Software Foundation; either version 2 of the License, or
> @@ -2630,6 +2634,171 @@ static void __mem_cgroup_commit_charge(struct mem_cgroup
*memcg,
>   memcg_check_events(memcg, page);
> }
>
> +#ifdef CONFIG_MEMCG_KMEM
> +static inline bool memcg_can_account_kmem(struct mem_cgroup *memcg)
> +{
> + return !mem_cgroup_disabled() && !mem_cgroup_is_root(memcg) &&
> + (memcg->kmem_accounted & KMEM_ACCOUNTED_MASK);
> +}
> +
> +static int memcg_charge_kmem(struct mem_cgroup *memcg, gfp_t gfp, u64 size)
> +{
> + struct res_counter *fail_res;
> + struct mem_cgroup *_memcg;
> + int ret = 0;
> + bool may_oom;
> +
> + ret = res_counter_charge(&memcg->kmem, size, &fail_res);
> + if (ret)
> + return ret;
> +
> + /*

```
> +  * Conditions under which we can wait for the oom_killer.
> +  * We have to be able to wait, but also, if we can't retry,
> +  * we obviously shouldn't go mess with oom.
> +  */
> + may_oom = (gfp & __GFP_WAIT) && !(gfp & __GFP_NORETRY);
```

What about gfp & __GFP_FS?

```
> +
> + _memcg = memcg;
> + ret = __mem_cgroup_try_charge(NULL, gfp, size >> PAGE_SHIFT,
> +          &_memcg, may_oom);
> +
> + if (ret == -EINTR)  {
> +  /*
> +   * __mem_cgroup_try_charge() chosed to bypass to root due to
> +   * OOM kill or fatal signal.  Since our only options are to
> +   * either fail the allocation or charge it to this cgroup, do
> +   * it as a temporary condition. But we can't fail. From a
> +   * kmem/slab perspective, the cache has already been selected,
> +   * by mem_cgroup_get_kmem_cache(), so it is too late to change
> +   * our minds. This condition will only trigger if the task
> +   * entered memcg_charge_kmem in a sane state, but was
> +   * OOM-killed.  during __mem_cgroup_try_charge. Tasks that are
```

Looks like some copy-and-paste damage.

```
> +   * already dying when the allocation triggers should have been
> +   * already directed to the root cgroup.
> +   */
> +  res_counter_charge_nofail(&memcg->res, size, &fail_res);
> +  if (do_swap_account)
> +   res_counter_charge_nofail(&memcg->memsw, size,
> +         &fail_res);
> +  ret = 0;
> + } else if (ret)
> +  res_counter_uncharge(&memcg->kmem, size);
> +
> + return ret;
> +}
> +
> +static void memcg_uncharge_kmem(struct mem_cgroup *memcg, u64 size)
> +{
> + res_counter_uncharge(&memcg->kmem, size);
> + res_counter_uncharge(&memcg->res, size);
> + if (do_swap_account)
> +  res_counter_uncharge(&memcg->memsw, size);
> +}
```

```
> +
> +/*
> + * 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
> + * need a further commit step to do the final arrangements.
> + *
> + * It is possible for the task to switch cgroups in this mean time, so at
> + * commit time, we can't rely on task conversion any longer.  We'll then use
> + * the handle argument to return to the caller which cgroup we should commit
> + * against. We could also return the memcg directly and avoid the pointer
> + * passing, but a boolean return value gives better semantics considering
> + * the compiled-out case as well.
> + *
> + * Returning true means the allocation is possible.
> + */
> +bool
> +__memcg_kmem_newpage_charge(gfp_t gfp, struct mem_cgroup **_memcg, int order)
> +{
> + struct mem_cgroup *memcg;
> + int ret;
> +
> + *_memcg = NULL;
> + memcg = try_get_mem_cgroup_from_mm(current->mm);
> +
> + /*
> +  * very rare case described in mem_cgroup_from_task. Unfortunately there
> +  * isn't much we can do without complicating this too much, and it would
> +  * be gfp-dependent anyway. Just let it go
> +  */
> + if (unlikely(!memcg))
> +  return true;
> +
> + if (!memcg_can_account_kmem(memcg)) {
> +  css_put(&memcg->css);
> +  return true;
> + }
> +
> + mem_cgroup_get(memcg);
> +
> + ret = memcg_charge_kmem(memcg, gfp, PAGE_SIZE << order);
> + if (!ret)
> +  *_memcg = memcg;
> + else
> +  mem_cgroup_put(memcg);
> +
> + css_put(&memcg->css);
> + return (ret == 0);
> +}
```

```
> +
> +void __memcg_kmem_commit_charge(struct page *page, struct mem_cgroup *memcg,
> +         int order)
> +{
> + struct page_cgroup *pc;
> +
> + VM_BUG_ON(mem_cgroup_is_root(memcg));
> +
> + /* The page allocation failed. Revert */
> + if (!page) {
> +  memcg_uncharge_kmem(memcg, PAGE_SIZE << order);
> +  mem_cgroup_put(memcg);
> +  return;
> + }
> +
> + pc = lookup_page_cgroup(page);
> + lock_page_cgroup(pc);
> + pc->mem_cgroup = memcg;
> + SetPageCgroupUsed(pc);
> + unlock_page_cgroup(pc);
> +}
> +
> +void __memcg_kmem_uncharge_page(struct page *page, int order)
> +{
> + struct mem_cgroup *memcg = NULL;
> + struct page_cgroup *pc;
> +
> +
> + pc = lookup_page_cgroup(page);
> + /*
> +  * Fast unlocked return. Theoretically might have changed, have to
> +  * check again after locking.
> +  */
> + if (!PageCgroupUsed(pc))
> +  return;
> +
> + lock_page_cgroup(pc);
> + if (PageCgroupUsed(pc)) {
> +  memcg = pc->mem_cgroup;
> +  ClearPageCgroupUsed(pc);
> + }
> + unlock_page_cgroup(pc);
> +
> + /*
> +  * We trust that only if there is a memcg associated with the page, it
> +  * is a valid allocation
> +  */
> + if (!memcg)
```

> + return;
> +
> + VM_BUG_ON(mem_cgroup_is_root(memcg));
> + memcg_uncharge_kmem(memcg, PAGE_SIZE << order);
> + mem_cgroup_put(memcg);

Should this mem_cgroup_put() be done conditionally on
memcg->kmem_accounted & KMEM_ACCOUNTED_MASK?

The next patch in the series does memcg_kmem_newpage_charge() in the page
allocator which will return true for memcg_can_account_kmem() without
doing mem_cgroup_get().

> +}
> +#endif /* CONFIG_MEMCG_KMEM */
> +
> #ifdef CONFIG_TRANSPARENT_HUGEPAGE
>
> #define PCGF_NOCOPY_AT_SPLIT (1 << PCG_LOCK | 1 << PCG_MIGRATION)

Subject: Re: [PATCH v5 07/14] mm: Allocate kernel pages to the right memcg
Posted by David Rientjes on Wed, 17 Oct 2012 22:43:23 GMT
View Forum Message <> Reply to Message

On Tue, 16 Oct 2012, Glauber Costa wrote:

> When a process tries to allocate a page with the __GFP_KMEMCG flag, the
> page allocator will call the corresponding memcg functions to validate
> the allocation. Tasks in the root memcg can always proceed.
>
> To avoid adding markers to the page - and a kmem flag that would
> necessarily follow, as much as doing page_cgroup lookups for no reason,
> whoever is marking its allocations with __GFP_KMEMCG flag is responsible
> for telling the page allocator that this is such an allocation at
> free_pages() time. This is done by the invocation of
> __free_accounted_pages() and free_accounted_pages().
>
> [v2: inverted test order to avoid a memcg_get leak,
>   free_accounted_pages simplification]
> [v4: test for TIF_MEMDIE at newpage_charge]
>
> Signed-off-by: Glauber Costa <glommer@parallels.com>
> Acked-by: Michal Hocko <mhocko@suse.cz>
> Acked-by: Mel Gorman <mgorman@suse.de>
> Acked-by: Kamezawa Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com>
> CC: Christoph Lameter <cl@linux.com>
> CC: Pekka Enberg <penberg@cs.helsinki.fi>

> CC: Johannes Weiner <hannes@cmpxchg.org>
> CC: Suleiman Souhlal <suleiman@google.com>
> CC: Tejun Heo <tj@kernel.org>

Acked-by: David Rientjes <rientjes@google.com>

Subject: Re: [PATCH v5 09/14] memcg: kmem accounting lifecycle management
Posted by David Rientjes on Wed, 17 Oct 2012 23:28:38 GMT
View Forum Message <> Reply to Message

On Tue, 16 Oct 2012, Glauber Costa wrote:

> diff --git a/mm/memcontrol.c b/mm/memcontrol.c
> index 1182188..e24b388 100644
> --- a/mm/memcontrol.c
> +++ b/mm/memcontrol.c
> @@ -344,6 +344,7 @@ struct mem_cgroup {
> /* internal only representation about the status of kmem accounting. */
> enum {
> KMEM_ACCOUNTED_ACTIVE = 0, /* accounted by this cgroup itself */
> + KMEM_ACCOUNTED_DEAD, /* dead memcg, pending kmem charges */

"dead memcg with pending kmem charges" seems better.

> };
>
> #define KMEM_ACCOUNTED_MASK (1 << KMEM_ACCOUNTED_ACTIVE)
> @@ -353,6 +354,22 @@ static void memcg_kmem_set_active(struct mem_cgroup *memcg)
> {
> set_bit(KMEM_ACCOUNTED_ACTIVE, &memcg->kmem_accounted);
> }
> +
> +static bool memcg_kmem_is_active(struct mem_cgroup *memcg)
> +{
> + return test_bit(KMEM_ACCOUNTED_ACTIVE, &memcg->kmem_accounted);
> +}

I think all of these should be inline.

> +
> +static void memcg_kmem_mark_dead(struct mem_cgroup *memcg)
> +{
> + if (test_bit(KMEM_ACCOUNTED_ACTIVE, &memcg->kmem_accounted))
> + set_bit(KMEM_ACCOUNTED_DEAD, &memcg->kmem_accounted);
> +}

The set_bit() doesn't happen atomically with the test_bit(), what

synchronization is required for this?

Subject: Re: [PATCH v5 09/14] memcg: kmem accounting lifecycle management
Posted by Michal Hocko on Thu, 18 Oct 2012 06:14:51 GMT
View Forum Message <> Reply to Message

On Wed 17-10-12 16:28:38, David Rientjes wrote:
> On Tue, 16 Oct 2012, Glauber Costa wrote:
[...]
> > +
> > +static void memcg_kmem_mark_dead(struct mem_cgroup *memcg)
> > +{
> > + if (test_bit(KMEM_ACCOUNTED_ACTIVE, &memcg->kmem_accounted))
> > + set_bit(KMEM_ACCOUNTED_DEAD, &memcg->kmem_accounted);
> > +}
>
> The set_bit() doesn't happen atomically with the test_bit(), what
> synchronization is required for this?

The group has to be active in order to become dead so the ordering is
natural and you do not need to test&set atomicaly. Also once a group
becomes active it is always marked that way until it goes away.
--
Michal Hocko
SUSE Labs

Subject: Re: [PATCH v5 00/14] kmem controller for memcg.
Posted by Glauber Costa on Thu, 18 Oct 2012 08:52:14 GMT
View Forum Message <> Reply to Message

On 10/18/2012 02:11 AM, Andrew Morton wrote:
> On Tue, 16 Oct 2012 14:16:37 +0400
> Glauber Costa <glommer@parallels.com> wrote:
>
>> ...
>>
>> A general explanation of what this is all about follows:
>>
>> The kernel memory limitation mechanism for memcg concerns itself with
>> disallowing potentially non-reclaimable allocations to happen in exaggerate
>> quantities by a particular set of processes (cgroup). Those allocations could
>> create pressure that affects the behavior of a different and unrelated set of
>> processes.
>>
>> Its basic working mechanism is to annotate some allocations with the

>> _GFP_KMEMCG flag. When this flag is set, the current process allocating will
>> have its memcg identified and charged against. When reaching a specific limit,
>> further allocations will be denied.
>
> The need to set _GFP_KMEMCG is rather unpleasing, and makes one wonder
> "why didn't it just track all allocations".
>
This was raised as well by Peter Zijlstra during the memcg summit. The
answer I gave to him still stands: There is a cost associated with it.
We believe it comes down to a trade off situation. How much tracking a
particular kind of allocation help vs how much does it cost.

The free path is specially more expensive, since it will always incur in
a page_cgroup lookup.

> Does this mean that over time we can expect more sites to get the
> _GFP_KMEMCG tagging?

We have being doing kernel memory limitation for OpenVZ for a lot of
times, using a quite different mechanism. What we do in this work (with
slab included), allows us to achieve feature parity with that. It means
it is good enough for production environments.

Whether or not more people will want other allocations to be tracked, I
can't predict. What I do can say is that stack + slab is a very
significant part of the memory one potentially cares about, and if
anyone else ever have the need for more, it will come down to a
trade-off calculation.

> If so, are there any special implications, or do
> we just go in, do the one-line patch and expect everything to work?

With the infrastructure in place, it shouldn't be hard. But it's not
necessarily a one-liner either. It depends on what are the pratical
considerations for having that specific kind of allocation tied to a
memcg. The slab, for instance, that follows this series, is far away
from a one-liner: it is in fact, a 19-patch patch series.

>
> And how *accurate* is the proposed code? What percentage of kernel
> memory allocations are unaccounted, typical case and worst case?

With both patchsets applied, all memory used for the stack and most of
the memory used for slab objects allocated in userspace process contexts
are accounted.

I honestly don't know which percentage of the total kernel memory this represents.

The accuracy for stack pages is very high: In this series, we don't move stack pages around when moving a task to other cgroups (for stack, it could be done), but other than that, all processes that pops up in a cgroup and stay there will have its memory accurately accounted.

The slab is more complicated, and depends on the workload. It will be more accurate in workloads in which the level of object-sharing among cgroups is low. A container, for instance, is the perfect example of where this happen.

>
> All sorts of questions come to mind over this decision, but it was
> unexplained. It should be, please. A lot!
>
>>
>> ...
>>
>> Limits lower than
>> the user limit effectively means there is a separate kernel memory limit that
>> may be reached independently than the user limit. Values equal or greater than
>> the user limit implies only that kernel memory is tracked. This provides a
>> unified vision of "maximum memory", be it kernel or user memory.
>>
>
> I'm struggling to understand that text much at all. Reading the
> Documentation/cgroups/memory.txt patch helped.
>

Great. If you have any specific suggestions I can change that. Maybe I should just paste the documentation bit in here...

Subject: Re: [PATCH v5 01/14] memcg: Make it possible to use the stock for more than one page.
Posted by Glauber Costa on Thu, 18 Oct 2012 08:55:32 GMT
View Forum Message <> Reply to Message

On 10/18/2012 02:11 AM, Andrew Morton wrote:
> On Tue, 16 Oct 2012 14:16:38 +0400
> Glauber Costa <glommer@parallels.com> wrote:
>
>> From: Suleiman Souhlal <ssouhlal@FreeBSD.org>
>>
>> We currently have a percpu stock cache scheme that charges one page at a
>> time from memcg->res, the user counter. When the kernel memory

>> controller comes into play, we'll need to charge more than that.
>>
>> This is because kernel memory allocations will also draw from the user
>> counter, and can be bigger than a single page, as it is the case with
>> the stack (usually 2 pages) or some higher order slabs.
>>
>> ...
>>
>> -/*
>> - * Try to consume stocked charge on this cpu. If success, one page is consumed
>> - * from local stock and true is returned. If the stock is 0 or charges from a
>> - * cgroup which is not current target, returns false. This stock will be
>> - * refilled.
>> +/**
>> + * consume_stock: Try to consume stocked charge on this cpu.
>> + * @memcg: memcg to consume from.
>> + * @nr_pages: how many pages to charge.
>> + *
>> + * The charges will only happen if @memcg matches the current cpu's memcg
>> + * stock, and at least @nr_pages are available in that stock. Failure to
>> + * service an allocation will refill the stock.
>> + *
>> + * returns true if succesfull, false otherwise.
>
> spello.
>
Thanks. I can never successfuly write successfull =(

>>   */
>> -static bool consume_stock(struct mem_cgroup *memcg)
>> +static bool consume_stock(struct mem_cgroup *memcg, int nr_pages)
>
> I don't believe there is a case for nr_pages < 0 here? If not then I
> suggest that it would be clearer to use an unsigned type, like
> memcg_stock_pcp.stock.
>

Done, thanks.

Subject: Re: [PATCH v5 04/14] kmem accounting basic infrastructure
Posted by Glauber Costa on Thu, 18 Oct 2012 09:02:50 GMT
View Forum Message <> Reply to Message

On 10/18/2012 02:08 AM, David Rientjes wrote:
> On Tue, 16 Oct 2012, Glauber Costa wrote:
>
>> This patch adds the basic infrastructure for the accounting of kernel

>> memory. To control that, the following files are created:
>>
>> * memory.kmem.usage_in_bytes
>> * memory.kmem.limit_in_bytes
>> * memory.kmem.failcnt
>> * memory.kmem.max_usage_in_bytes
>>
>> They have the same meaning of their user memory counterparts. They
>> reflect the state of the "kmem" res_counter.
>>
>> Per cgroup kmem memory accounting is not enabled until a limit is set
>> for the group. Once the limit is set the accounting cannot be disabled
>> for that group. This means that after the patch is applied, no
>> behavioral changes exists for whoever is still using memcg to control
>> their memory usage, until memory.kmem.limit_in_bytes is set for the
>> first time.
>>
>> We always account to both user and kernel resource_counters. This
>> effectively means that an independent kernel limit is in place when the
>> limit is set to a lower value than the user memory. A equal or higher
>> value means that the user limit will always hit first, meaning that kmem
>> is effectively unlimited.
>>
>> People who want to track kernel memory but not limit it, can set this
>> limit to a very high number (like RESOURCE_MAX - 1page - that no one
>> will ever hit, or equal to the user memory)
>>
>> [v4: make kmem files part of the main array;
>>   do not allow limit to be set for non-empty cgroups]
>> [v5: cosmetic changes]
>>
>> Signed-off-by: Glauber Costa <glommer@parallels.com>
>> Acked-by: Kamezawa Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com>
>> CC: Michal Hocko <mhocko@suse.cz>
>> CC: Johannes Weiner <hannes@cmpxchg.org>
>> CC: Tejun Heo <tj@kernel.org>
>> ---
>> mm/memcontrol.c | 116
++-
>> 1 file changed, 115 insertions(+), 1 deletion(-)
>>
>> diff --git a/mm/memcontrol.c b/mm/memcontrol.c
>> index 71d259e..30eafeb 100644
>> --- a/mm/memcontrol.c
>> +++ b/mm/memcontrol.c
>> @@ -266,6 +266,10 @@ struct mem_cgroup {
>>   };
>>

>>   /*
>> + * the counter to account for kernel memory usage.
>> + */
>> + struct res_counter kmem;
>> + /*
>>   * Per cgroup active and inactive list, similar to the
>>   * per zone LRU lists.
>>   */
>> @@ -280,6 +284,7 @@ struct mem_cgroup {
>>   * Should the accounting and control be hierarchical, per subtree?
>>   */
>>   bool use_hierarchy;
>> + unsigned long kmem_accounted; /* See KMEM_ACCOUNTED_*, below */
>
> I think this should be named kmem_account_flags or kmem_flags, otherwise
> it appears that this is the actual account.
>

ok.

>>
>>   bool oom_lock;
>>   atomic_t under_oom;
>> @@ -332,6 +337,20 @@ struct mem_cgroup {
>> #endif
>> };
>>
>> +/* internal only representation about the status of kmem accounting. */
>> +enum {
>> + KMEM_ACCOUNTED_ACTIVE = 0, /* accounted by this cgroup itself */
>> +};
>> +
>> +#define KMEM_ACCOUNTED_MASK (1 << KMEM_ACCOUNTED_ACTIVE)
>> +
>> +#ifdef CONFIG_MEMCG_KMEM
>
> memcg->kmem_accounted isn't only defined for this configuration, so would
> it be simpler to define this unconditionally?
>
The functions that manipulate those bits are. Reason being they are only
used for #CONFIG_KMEMCG_KMEM, and gcc will complain about
defined-but-not-used static functions.

>> +static void memcg_kmem_set_active(struct mem_cgroup *memcg)
>
> inline?
>
ok.

```
>> +{
>> + set_bit(KMEM_ACCOUNTED_ACTIVE, &memcg->kmem_accounted);
>> +}
>> +#endif
>> +
>>  /* Stuffs for move charges at task migration. */
>>  /*
>>   * Types of charges to be moved. "move_charge_at_immitgrate" is treated as a
>> @@ -390,6 +409,7 @@ enum res_type {
>>   _MEM,
>>   _MEMSWAP,
>>   _OOM_TYPE,
>> + _KMEM,
>>  };
>>
>>  #define MEMFILE_PRIVATE(x, val) ((x) << 16 | (val))
>> @@ -1433,6 +1453,10 @@ done:
>>    res_counter_read_u64(&memcg->memsw, RES_USAGE) >> 10,
>>    res_counter_read_u64(&memcg->memsw, RES_LIMIT) >> 10,
>>    res_counter_read_u64(&memcg->memsw, RES_FAILCNT));
>> + printk(KERN_INFO "kmem: usage %llukB, limit %llukB, failcnt %llu\n",
>> +  res_counter_read_u64(&memcg->kmem, RES_USAGE) >> 10,
>> +  res_counter_read_u64(&memcg->kmem, RES_LIMIT) >> 10,
>> +  res_counter_read_u64(&memcg->kmem, RES_FAILCNT));
>>  }
>>
>>  /*
>> @@ -3940,6 +3964,9 @@ static ssize_t mem_cgroup_read(struct cgroup *cont, struct cftype
*cft,
>>    else
>>     val = res_counter_read_u64(&memcg->memsw, name);
>>    break;
>> + case _KMEM:
>> +  val = res_counter_read_u64(&memcg->kmem, name);
>> +  break;
>>   default:
>>    BUG();
>>   }
>> @@ -3947,6 +3974,57 @@ static ssize_t mem_cgroup_read(struct cgroup *cont, struct cftype
*cft,
>>   len = scnprintf(str, sizeof(str), "%llu\n", (unsigned long long)val);
>>   return simple_read_from_buffer(buf, nbytes, ppos, str, len);
>>  }
>> +
>> +static int memcg_update_kmem_limit(struct cgroup *cont, u64 val)
>> +{
>> + int ret = -EINVAL;
```

>> +#ifdef CONFIG_MEMCG_KMEM
>> + struct mem_cgroup *memcg = mem_cgroup_from_cont(cont);
>> + /*
>> + * For simplicity, we won't allow this to be disabled. It also can't
>> + * be changed if the cgroup has children already, or if tasks had
>> + * already joined.
>> + *
>> + * If tasks join before we set the limit, a person looking at
>> + * kmem.usage_in_bytes will have no way to determine when it took
>> + * place, which makes the value quite meaningless.
>> + *
>> + * After it first became limited, changes in the value of the limit are
>> + * of course permitted.
>> + *
>> + * Taking the cgroup_lock is really offensive, but it is so far the only
>> + * way to guarantee that no children will appear. There are plenty of
>> + * other offenders, and they should all go away. Fine grained locking
>> + * is probably the way to go here. When we are fully hierarchical, we
>> + * can also get rid of the use_hierarchy check.
>
> Not sure it's so offensive, it's a pretty standard way of ensuring that
> cont->children doesn't get manipulated in a race.
>

Yes, it is. But Tejun is currently in a cruzade (in which I pretty much
back him up) to get rid of all uses of the cgroup_lock outside cgroup.c.

That is the offensive part. But it is also how things are done in memcg
right now, and there is nothing fundamentally different in this one.
Whatever lands in the remaining offenders, can land in here.

>> + */
>> + cgroup_lock();
>> + mutex_lock(&set_limit_mutex);
>> + if (!memcg->kmem_accounted && val != RESOURCE_MAX) {
>> + if (cgroup_task_count(cont) || (memcg->use_hierarchy &&
>> +   !list_empty(&cont->children))) {
>> +   ret = -EBUSY;
>> +   goto out;
>> + }
>> + ret = res_counter_set_limit(&memcg->kmem, val);
>> + VM_BUG_ON(ret);
>> +
>> + memcg_kmem_set_active(memcg);
>> + } else
>> + ret = res_counter_set_limit(&memcg->kmem, val);
>> +out:

>> + mutex_unlock(&set_limit_mutex);
>> + cgroup_unlock();
>> +#endif
>> + return ret;
>> +}
>> +
>> +static void memcg_propagate_kmem(struct mem_cgroup *memcg,
>> +   struct mem_cgroup *parent)
>> +{
>> + memcg->kmem_accounted = parent->kmem_accounted;
>> +}
>> +
>> /*
>>   * The user of this function is...
>>   * RES_LIMIT.
>> @@ -3978,8 +4056,12 @@ static int mem_cgroup_write(struct cgroup *cont, struct cftype *cft,
>>   break;
>>   if (type == _MEM)
>>   ret = mem_cgroup_resize_limit(memcg, val);
>> - else
>> + else if (type == _MEMSWAP)
>>   ret = mem_cgroup_resize_memsw_limit(memcg, val);
>> + else if (type == _KMEM)
>> +   ret = memcg_update_kmem_limit(cont, val);
>> + else
>> +   return -EINVAL;
>
> I like how this is done in a maintainable way to ensure no other types can
> inadvertently update the memsw limit as it was previously written. All
> other returns of -EINVAL just cause the switch statement to break, though,
> rather than return directly.
>
>>   break;
>>   case RES_SOFT_LIMIT:
>>   ret = res_counter_memparse_write_strategy(buffer, &val);
>> @@ -4045,12 +4127,16 @@ static int mem_cgroup_reset(struct cgroup *cont, unsigned int
event)
>>   case RES_MAX_USAGE:
>>   if (type == _MEM)
>>   res_counter_reset_max(&memcg->res);
>> + else if (type == _KMEM)
>> +   res_counter_reset_max(&memcg->kmem);
>
> Could this be written in the same way above, i.e. check _MEMSWAP to pass
> memcg->memsw, _KMEM for memcg->kmem, etc?
>
>>   else
>>   res_counter_reset_max(&memcg->memsw);

>>   break;
>>   case RES_FAILCNT:
>>   if (type == _MEM)
>>   res_counter_reset_failcnt(&memcg->res);
>> + else if (type == _KMEM)
>> +   res_counter_reset_failcnt(&memcg->kmem);
>
> Same.
>
Done.

Subject: Re: [PATCH v5 04/14] kmem accounting basic infrastructure
Posted by Glauber Costa on Thu, 18 Oct 2012 09:04:25 GMT
View Forum Message <> Reply to Message

On 10/18/2012 02:12 AM, Andrew Morton wrote:
> On Tue, 16 Oct 2012 14:16:41 +0400
> Glauber Costa <glommer@parallels.com> wrote:
>
>> This patch adds the basic infrastructure for the accounting of kernel
>> memory. To control that, the following files are created:
>>
>> * memory.kmem.usage_in_bytes
>> * memory.kmem.limit_in_bytes
>> * memory.kmem.failcnt
>
> gargh. "failcnt" is not a word. Who was it who first thought that
> omitting voewls from words improves anything?
>
> Sigh. That pooch is already screwed and there's nothing we can do
> about it now.
>

Dunno =(

>> * memory.kmem.max_usage_in_bytes
>>
>> They have the same meaning of their user memory counterparts. They
>> reflect the state of the "kmem" res_counter.
>>
>> Per cgroup kmem memory accounting is not enabled until a limit is set
>> for the group. Once the limit is set the accounting cannot be disabled
>> for that group. This means that after the patch is applied, no
>> behavioral changes exists for whoever is still using memcg to control
>> their memory usage, until memory.kmem.limit_in_bytes is set for the
>> first time.

>>
>> We always account to both user and kernel resource_counters. This
>> effectively means that an independent kernel limit is in place when the
>> limit is set to a lower value than the user memory. A equal or higher
>> value means that the user limit will always hit first, meaning that kmem
>> is effectively unlimited.
>>
>> People who want to track kernel memory but not limit it, can set this
>> limit to a very high number (like RESOURCE_MAX - 1page - that no one
>> will ever hit, or equal to the user memory)
>>
>>
>> ...
>>
>> +/* internal only representation about the status of kmem accounting. */
>> +enum {
>> + KMEM_ACCOUNTED_ACTIVE = 0, /* accounted by this cgroup itself */
>> +};
>> +
>> +#define KMEM_ACCOUNTED_MASK (1 << KMEM_ACCOUNTED_ACTIVE)
>> +
>> +#ifdef CONFIG_MEMCG_KMEM
>> +static void memcg_kmem_set_active(struct mem_cgroup *memcg)
>> +{
>> + set_bit(KMEM_ACCOUNTED_ACTIVE, &memcg->kmem_accounted);
>> +}
>> +#endif
>
> I don't think memcg_kmem_set_active() really needs to exist. It has a
> single caller and is unlikely to get any additional callers, so just
> open-code it there?
>

Actually they exist as a way to make everything fit in closer to
80-columns without writing the function spanning 10 lines.

I can open code them if you guys prefer.

Subject: Re: [PATCH v5 06/14] memcg: kmem controller infrastructure
Posted by Glauber Costa on Thu, 18 Oct 2012 09:16:19 GMT
View Forum Message <> Reply to Message

On 10/18/2012 02:12 AM, Andrew Morton wrote:
> On Tue, 16 Oct 2012 14:16:43 +0400
> Glauber Costa <glommer@parallels.com> wrote:
>
>> This patch introduces infrastructure for tracking kernel memory pages to

>> a given memcg. This will happen whenever the caller includes the flag
>> __GFP_KMEMCG flag, and the task belong to a memcg other than the root.
>>
>> In memcontrol.h those functions are wrapped in inline acessors. The
>> idea is to later on, patch those with static branches, so we don't incur
>> any overhead when no mem cgroups with limited kmem are being used.
>>
>> Users of this functionality shall interact with the memcg core code
>> through the following functions:
>>
>> memcg_kmem_newpage_charge: will return true if the group can handle the
>>   allocation. At this point, struct page is not
>>   yet allocated.
>>
>> memcg_kmem_commit_charge: will either revert the charge, if struct page
>>   allocation failed, or embed memcg information
>>   into page_cgroup.
>>
>> memcg_kmem_uncharge_page: called at free time, will revert the charge.
>>
>> ...
>>
>> +static __always_inline bool
>> +memcg_kmem_newpage_charge(gfp_t gfp, struct mem_cgroup **memcg, int order)
>> +{
>> + if (!memcg_kmem_enabled())
>> + return true;
>> +
>> + /*
>> + * __GFP_NOFAIL allocations will move on even if charging is not
>> + * possible. Therefore we don't even try, and have this allocation
>> + * unaccounted. We could in theory charge it with
>> + * res_counter_charge_nofail, but we hope those allocations are rare,
>> + * and won't be worth the trouble.
>> + */
>> + if (!(gfp & __GFP_KMEMCG) || (gfp & __GFP_NOFAIL))
>> + return true;
>> + if (in_interrupt() || (!current->mm) || (current->flags & PF_KTHREAD))
>> + return true;
>> +
>> + /* If the test is dying, just let it go. */
>> +   if (unlikely(test_thread_flag(TIF_MEMDIE)
>> + || fatal_signal_pending(current)))
>> + return true;
>> +
>> + return __memcg_kmem_newpage_charge(gfp, memcg, order);
>> +}
>

> That's a big function! Why was it __always_inline? I'd have thought
> it would be better to move the code after memcg_kmem_enabled() out of
> line.
>

it is big, but it is mostly bit testing. So the goal here is to avoid a
function call at all costs, this being a fast path.

> Do we actually need to test PF_KTHREAD when current->mm == NULL?
> Perhaps because of aio threads whcih temporarily adopt a userspace mm?

I believe so. I remember I discussed this in the past with David
Rientjes and he advised me to test for both.

>
>> +/**
>> + * memcg_kmem_uncharge_page: uncharge pages from memcg
>> + * @page: pointer to struct page being freed
>> + * @order: allocation order.
>> + *
>> + * there is no need to specify memcg here, since it is embedded in page_cgroup
>> + */
>> +static __always_inline void
>> +memcg_kmem_uncharge_page(struct page *page, int order)
>> +{
>> + if (memcg_kmem_enabled())
>> + __memcg_kmem_uncharge_page(page, order);
>> +}
>> +
>> +/**
>> + * memcg_kmem_commit_charge: embeds correct memcg in a page
>> + * @page: pointer to struct page recently allocated
>> + * @memcg: the memcg structure we charged against
>> + * @order: allocation order.
>> + *
>> + * Needs to be called after memcg_kmem_newpage_charge, regardless of success or
>> + * failure of the allocation. if @page is NULL, this function will revert the
>> + * charges. Otherwise, it will commit the memcg given by @memcg to the
>> + * corresponding page_cgroup.
>> + */
>> +static __always_inline void
>> +memcg_kmem_commit_charge(struct page *page, struct mem_cgroup *memcg, int order)
>> +{
>> + if (memcg_kmem_enabled() && memcg)
>> + __memcg_kmem_commit_charge(page, memcg, order);
>> +}
>
> I suspect the __always_inline's here are to do with static branch

> trickery. A code comment is warranted if so?
>

Not necessarily. Same thing as above. We want to avoid function calls in
those sites.

Subject: Re: [PATCH v5 06/14] memcg: kmem controller infrastructure
Posted by Glauber Costa on Thu, 18 Oct 2012 09:23:28 GMT
View Forum Message <> Reply to Message

On 10/18/2012 02:37 AM, David Rientjes wrote:
> On Tue, 16 Oct 2012, Glauber Costa wrote:
>
>> + /* If the test is dying, just let it go. */
>> +   if (unlikely(test_thread_flag(TIF_MEMDIE)
>> + || fatal_signal_pending(current)))
>> + return true;
>
> This can be simplified to just check fatal_signal_pending(), all threads
> with TIF_MEMDIE also have a pending SIGKILL.

Yes, I believe it is better. I will change.

>> +
>> + return __memcg_kmem_newpage_charge(gfp, memcg, order);
>> +}
>> +
>> +/**
>> + * memcg_kmem_uncharge_page: uncharge pages from memcg
>
> Should be memcg_kmem_uncharge_pages() since it takes an order argument?
>

I tried to use naming as close as possible to user-memcg. But to be
fair, their are always calling it page-by-page, so pages() won't be a
problem here.

>> + * @page: pointer to struct page being freed
>> + * @order: allocation order.
>> + *
>> + * there is no need to specify memcg here, since it is embedded in page_cgroup
>> + */
>> +static __always_inline void
>> +memcg_kmem_uncharge_page(struct page *page, int order)
>> +{
>> + if (memcg_kmem_enabled())
>> + __memcg_kmem_uncharge_page(page, order);

>> +}
>> +
>> +/**
>> + * memcg_kmem_commit_charge: embeds correct memcg in a page
>> + * @page: pointer to struct page recently allocated
>> + * @memcg: the memcg structure we charged against
>> + * @order: allocation order.
>> + *
>> + * Needs to be called after memcg_kmem_newpage_charge, regardless of success or
>> + * failure of the allocation. if @page is NULL, this function will revert the
>> + * charges. Otherwise, it will commit the memcg given by @memcg to the
>> + * corresponding page_cgroup.
>> + */
>> +static __always_inline void
>> +memcg_kmem_commit_charge(struct page *page, struct mem_cgroup *memcg, int order)
>> +{
>> + if (memcg_kmem_enabled() && memcg)
>> + __memcg_kmem_commit_charge(page, memcg, order);
>> +}
>> +
>> #else
>> static inline void sock_update_memcg(struct sock *sk)
>> {
>> @@ -406,6 +489,21 @@ static inline void sock_update_memcg(struct sock *sk)
>> static inline void sock_release_memcg(struct sock *sk)
>> {
>> }
>> +
>> +static inline bool
>> +memcg_kmem_newpage_charge(gfp_t gfp, struct mem_cgroup **memcg, int order)
>> +{
>> + return true;
>> +}
>> +
>> +static inline void memcg_kmem_uncharge_page(struct page *page, int order)
>
> Two spaces.
>

Thanks.

>> +{
>> +}
>> +
>> +static inline void
>> +memcg_kmem_commit_charge(struct page *page, struct mem_cgroup *memcg, int order)
>> +{
>> +}

>> #endif /* CONFIG_MEMCG_KMEM */
>> #endif /* _LINUX_MEMCONTROL_H */
>>
>> diff --git a/mm/memcontrol.c b/mm/memcontrol.c
>> index 30eafeb..1182188 100644
>> --- a/mm/memcontrol.c
>> +++ b/mm/memcontrol.c
>> @@ -10,6 +10,10 @@
>>   * Copyright (C) 2009 Nokia Corporation
>>   * Author: Kirill A. Shutemov
>>   *
>> + * Kernel Memory Controller
>> + * Copyright (C) 2012 Parallels Inc. and Google Inc.
>> + * Authors: Glauber Costa and Suleiman Souhlal
>> + *
>>   * This program is free software; you can redistribute it and/or modify
>>   * it under the terms of the GNU General Public License as published by
>>   * the Free Software Foundation; either version 2 of the License, or
>> @@ -2630,6 +2634,171 @@ static void __mem_cgroup_commit_charge(struct mem_cgroup
*memcg,
>>   memcg_check_events(memcg, page);
>> }
>>
>> +#ifdef CONFIG_MEMCG_KMEM
>> +static inline bool memcg_can_account_kmem(struct mem_cgroup *memcg)
>> +{
>> + return !mem_cgroup_disabled() && !mem_cgroup_is_root(memcg) &&
>> + (memcg->kmem_accounted & KMEM_ACCOUNTED_MASK);
>> +}
>> +
>> +static int memcg_charge_kmem(struct mem_cgroup *memcg, gfp_t gfp, u64 size)
>> +{
>> + struct res_counter *fail_res;
>> + struct mem_cgroup *_memcg;
>> + int ret = 0;
>> + bool may_oom;
>> +
>> + ret = res_counter_charge(&memcg->kmem, size, &fail_res);
>> + if (ret)
>> + return ret;
>> +
>> + /*
>> + * Conditions under which we can wait for the oom_killer.
>> + * We have to be able to wait, but also, if we can't retry,
>> + * we obviously shouldn't go mess with oom.
>> + */
>> + may_oom = (gfp & __GFP_WAIT) && !(gfp & __GFP_NORETRY);
>

> What about gfp & __GFP_FS?
>

Do you intend to prevent or allow OOM under that flag? I personally
think that anything that accepts to be OOM-killed should have GFP_WAIT
set, so that ought to be enough.

>> +
>> + _memcg = memcg;
>> + ret = __mem_cgroup_try_charge(NULL, gfp, size >> PAGE_SHIFT,
>> +   &_memcg, may_oom);
>> +
>> + if (ret == -EINTR) {
>> + /*
>> +   * __mem_cgroup_try_charge() chosed to bypass to root due to
>> +   * OOM kill or fatal signal. Since our only options are to
>> +   * either fail the allocation or charge it to this cgroup, do
>> +   * it as a temporary condition. But we can't fail. From a
>> +   * kmem/slab perspective, the cache has already been selected,
>> +   * by mem_cgroup_get_kmem_cache(), so it is too late to change
>> +   * our minds. This condition will only trigger if the task
>> +   * entered memcg_charge_kmem in a sane state, but was
>> +   * OOM-killed. during __mem_cgroup_try_charge. Tasks that are
>
> Looks like some copy-and-paste damage.
>

thanks.

>> +void __memcg_kmem_uncharge_page(struct page *page, int order)
>> +{
>> + struct mem_cgroup *memcg = NULL;
>> + struct page_cgroup *pc;
>> +
>> +
>> + pc = lookup_page_cgroup(page);
>> + /*
>> + * Fast unlocked return. Theoretically might have changed, have to
>> + * check again after locking.
>> + */
>> + if (!PageCgroupUsed(pc))
>> + return;
>> +
>> + lock_page_cgroup(pc);
>> + if (PageCgroupUsed(pc)) {
>> + memcg = pc->mem_cgroup;
>> + ClearPageCgroupUsed(pc);
>> + }

>> + unlock_page_cgroup(pc);
>> +
>> + /*
>> + * We trust that only if there is a memcg associated with the page, it
>> + * is a valid allocation
>> + */
>> + if (!memcg)
>> + return;
>> +
>> + VM_BUG_ON(mem_cgroup_is_root(memcg));
>> + memcg_uncharge_kmem(memcg, PAGE_SIZE << order);
>> + mem_cgroup_put(memcg);
>
> Should this mem_cgroup_put() be done conditionally on
> memcg->kmem_accounted & KMEM_ACCOUNTED_MASK?
>
> The next patch in the series does memcg_kmem_newpage_charge() in the page
> allocator which will return true for memcg_can_account_kmem() without
> doing mem_cgroup_get().
>

And then this put will go away as well.

I am not testing for memcg_can_account_kmem in here, because having or
not having the PageCgroupUsed bit set (and therefore, a valid memcg) in
page_cgroup should be the most robust test here.

Subject: Re: [PATCH v5 07/14] mm: Allocate kernel pages to the right memcg
Posted by Glauber Costa on Thu, 18 Oct 2012 09:24:47 GMT
View Forum Message <> Reply to Message

On 10/18/2012 02:12 AM, Andrew Morton wrote:
> On Tue, 16 Oct 2012 14:16:44 +0400
> Glauber Costa <glommer@parallels.com> wrote:
>
>> When a process tries to allocate a page with the __GFP_KMEMCG flag, the
>> page allocator will call the corresponding memcg functions to validate
>> the allocation. Tasks in the root memcg can always proceed.
>>
>> To avoid adding markers to the page - and a kmem flag that would
>> necessarily follow, as much as doing page_cgroup lookups for no reason,
>> whoever is marking its allocations with __GFP_KMEMCG flag is responsible
>> for telling the page allocator that this is such an allocation at
>> free_pages() time.
>
> Well, why? Was that the correct decision?
>

I don't fully understand your question. Is this the same question you
posed in patch 0, about marking some versus marking all? If so, I
believe I should have answered it there.

If not, please explain.

>> This is done by the invocation of
>> __free_accounted_pages() and free_accounted_pages().
>
> These are very general-sounding names. I'd expect the identifiers to
> contain "memcg" and/or "kmem", to identify what's going on.
>

Changed.

Subject: Re: [PATCH v5 11/14] memcg: allow a memcg with kmem charges to be destructed.
Posted by Glauber Costa on Thu, 18 Oct 2012 09:33:54 GMT
View Forum Message <> Reply to Message

On 10/18/2012 02:12 AM, Andrew Morton wrote:
> On Tue, 16 Oct 2012 14:16:48 +0400
> Glauber Costa <glommer@parallels.com> wrote:
>
>> Because the ultimate goal of the kmem tracking in memcg is to track slab
>> pages as well,
>
> It is? For a major patchset such as this, it's pretty important to
> discuss such long-term plans in the top-level discussion. Covering
> things such as expected complexity, expected performance hit, how these
> plans affected the current implementation, etc.
>
> The main reason for this is that if the future plans appear to be of
> doubtful feasibility and the current implementation isn't sufficiently
> useful without the future stuff, we shouldn't merge the current
> implementation. It's a big issue!
>

Not really. I am not talking about plans when it comes to slab. The code
is there, and usually always posted to linux-mm a few days after I post
this series. It also lives in the kmemcg-slab branch in my git tree.

I am trying to logically split it in two to aid reviewers work. I may
have made a mistake by splitting it this way, but so far I think it was
the right decision: it allowed people to focus on a part of the work
first, instead of going all the way in a 30-patch patch series that

would be merged atomically.

I believe they should be merged separately, to allow us to find any issues easier. But I also believe that this "separate" should ultimately live in the same merge window.

Pekka, from the slab side, already stated that 3.8 would not be unreasonable.

As for the perfomance hit, my latest benchmark, quoted in the opening mail of this series already include results for both patchsets.

Subject: Re: [PATCH v5 13/14] protect architectures where THREAD_SIZE >= PAGE_SIZE against fork bombs
Posted by Glauber Costa on Thu, 18 Oct 2012 09:37:39 GMT
View Forum Message <> Reply to Message

On 10/18/2012 02:12 AM, Andrew Morton wrote:
> On Tue, 16 Oct 2012 14:16:50 +0400
> Glauber Costa <glommer@parallels.com> wrote:
>
>> @@ -146,7 +146,7 @@ void __weak arch_release_thread_info(struct thread_info *ti)
>> static struct thread_info *alloc_thread_info_node(struct task_struct *tsk,
>> int node)
>> {
>> - struct page *page = alloc_pages_node(node, THREADINFO_GFP,
>> + struct page *page = alloc_pages_node(node, THREADINFO_GFP_ACCOUNTED,
>> THREAD_SIZE_ORDER);
>>
> yay, we actually used all this code for something ;)
>
Happy to be of use, sir!

> I don't think we really saw a comprehensive list of what else the kmem
> controller will be used for, but I believe that all other envisaged
> applications will require slab accounting, yes?
>
>
> So it appears that all we have at present is a
> yet-another-fork-bomb-preventer, but one which requires that the
> culprit be in a container? That's reasonable, given your
> hosted-environment scenario. It's unclear (to me) that we should merge
> all this code for only this feature. Again, it would be good to have a
> clear listing of and plan for other applications of this code.
>

I agree. This doesn't buy me much without slab accounting. But

reiterating what I've just said in another e-mail, slab accounting is
not really in plan stage, but had also been through extensive development.

As a matter of fact, it used to be only "slab accounting" in the
beginning, without this. I've split it more recently because I believe
it would allow people to do a more focused review, leading to better code.

Subject: Re: [PATCH v5 14/14] Add documentation about the kmem controller
Posted by Glauber Costa on Thu, 18 Oct 2012 09:38:08 GMT
View Forum Message <> Reply to Message

On 10/18/2012 02:12 AM, Andrew Morton wrote:
> On Tue, 16 Oct 2012 14:16:51 +0400
> Glauber Costa <glommer@parallels.com> wrote:
>
>> +Kernel memory won't be accounted at all until limit on a group is set. This
>> +allows for existing setups to continue working without disruption. The limit
>> +cannot be set if the cgroup have children, or if there are already tasks in the
>> +cgroup.
>
> What behaviour will usersapce see if "The limit cannot be set"?
> write() returns -EINVAL, something like that?
>
-EBUSY.

Will update.

Subject: Re: [PATCH v5 09/14] memcg: kmem accounting lifecycle management
Posted by Glauber Costa on Thu, 18 Oct 2012 09:42:11 GMT
View Forum Message <> Reply to Message

On 10/18/2012 03:28 AM, David Rientjes wrote:
> On Tue, 16 Oct 2012, Glauber Costa wrote:
>
>> diff --git a/mm/memcontrol.c b/mm/memcontrol.c
>> index 1182188..e24b388 100644
>> --- a/mm/memcontrol.c
>> +++ b/mm/memcontrol.c
>> @@ -344,6 +344,7 @@ struct mem_cgroup {
>> /* internal only representation about the status of kmem accounting. */
>> enum {
>>   KMEM_ACCOUNTED_ACTIVE = 0, /* accounted by this cgroup itself */
>> + KMEM_ACCOUNTED_DEAD, /* dead memcg, pending kmem charges */
>
> "dead memcg with pending kmem charges" seems better.

>
ok.

>> };
>>
>> #define KMEM_ACCOUNTED_MASK (1 << KMEM_ACCOUNTED_ACTIVE)
>> @@ -353,6 +354,22 @@ static void memcg_kmem_set_active(struct mem_cgroup *memcg)
>> {
>>   set_bit(KMEM_ACCOUNTED_ACTIVE, &memcg->kmem_accounted);
>> }
>> +
>> +static bool memcg_kmem_is_active(struct mem_cgroup *memcg)
>> +{
>> + return test_bit(KMEM_ACCOUNTED_ACTIVE, &memcg->kmem_accounted);
>> +}
>
> I think all of these should be inline.
>
They end up being, to be best of my knowledge the compiler can and will
inline such simple functions regardless of their marking, unless you
explicitly mark them noinline.

>> +
>> +static void memcg_kmem_mark_dead(struct mem_cgroup *memcg)
>> +{
>> + if (test_bit(KMEM_ACCOUNTED_ACTIVE, &memcg->kmem_accounted))
>> + set_bit(KMEM_ACCOUNTED_DEAD, &memcg->kmem_accounted);
>> +}
>
> The set_bit() doesn't happen atomically with the test_bit(), what
> synchronization is required for this?
>

I believe the explanation Michal gave in answer to this is comprehensive.

Subject: Re: [PATCH v5 07/14] mm: Allocate kernel pages to the right memcg
Posted by Glauber Costa on Thu, 18 Oct 2012 11:53:37 GMT
View Forum Message <> Reply to Message

On 10/18/2012 02:12 AM, Andrew Morton wrote:
> On Tue, 16 Oct 2012 14:16:44 +0400
> Glauber Costa <glommer@parallels.com> wrote:
>
>> When a process tries to allocate a page with the __GFP_KMEMCG flag, the
>> page allocator will call the corresponding memcg functions to validate
>> the allocation. Tasks in the root memcg can always proceed.

>>
>> To avoid adding markers to the page - and a kmem flag that would
>> necessarily follow, as much as doing page_cgroup lookups for no reason,
>> whoever is marking its allocations with __GFP_KMEMCG flag is responsible
>> for telling the page allocator that this is such an allocation at
>> free_pages() time.
>
> Well, why? Was that the correct decision?
>
>> This is done by the invocation of
>> __free_accounted_pages() and free_accounted_pages().
>
> These are very general-sounding names. I'd expect the identifiers to
> contain "memcg" and/or "kmem", to identify what's going on.
>
I've just changed to free_memcg_kmem_pages.
Let me know if the name is better.

Subject: Re: [PATCH v5 00/14] kmem controller for memcg.
Posted by akpm on Thu, 18 Oct 2012 19:21:05 GMT
View Forum Message <> Reply to Message

On Thu, 18 Oct 2012 20:51:05 +0400
Glauber Costa <glommer@parallels.com> wrote:

> On 10/18/2012 02:11 AM, Andrew Morton wrote:
> > On Tue, 16 Oct 2012 14:16:37 +0400
> > Glauber Costa <glommer@parallels.com> wrote:
> >
> >> ...
> >>
> >> A general explanation of what this is all about follows:
> >>
> >> The kernel memory limitation mechanism for memcg concerns itself with
> >> disallowing potentially non-reclaimable allocations to happen in exaggerate
> >> quantities by a particular set of processes (cgroup). Those allocations could
> >> create pressure that affects the behavior of a different and unrelated set of
> >> processes.
> >>
> >> Its basic working mechanism is to annotate some allocations with the
> >> _GFP_KMEMCG flag. When this flag is set, the current process allocating will
> >> have its memcg identified and charged against. When reaching a specific limit,
> >> further allocations will be denied.
> >
> > The need to set _GFP_KMEMCG is rather unpleasing, and makes one wonder
> > "why didn't it just track all allocations".
> >

> This was raised as well by Peter Zijlstra during the memcg summit.

Firstly: please treat any question from a reviewer as an indication
that information was missing from the changelog or from code comments.
Ideally all such queries are addressed in later version of the patch
and changelog.

> The
> answer I gave to him still stands: There is a cost associated with it.
> We believe it comes down to a trade off situation. How much tracking a
> particular kind of allocation help vs how much does it cost.
>
> The free path is specially more expensive, since it will always incur in
> a page_cgroup lookup.

OK. But that is a quantitative argument, without any quantities! Do
we have even an estimate of what this cost will be? Perhaps it's the
case that, if well implemented, that cost will be acceptable. How do
we tell?

> > Does this mean that over time we can expect more sites to get the
> > _GFP_KMEMCG tagging?
>
> We have being doing kernel memory limitation for OpenVZ for a lot of
> times, using a quite different mechanism. What we do in this work (with
> slab included), allows us to achieve feature parity with that. It means
> it is good enough for production environments.

That's really good info.

> Whether or not more people will want other allocations to be tracked, I
> can't predict. What I do can say is that stack + slab is a very
> significant part of the memory one potentially cares about, and if
> anyone else ever have the need for more, it will come down to a
> trade-off calculation.

OK.

> > If so, are there any special implications, or do
> > we just go in, do the one-line patch and expect everything to work?
>
> With the infrastructure in place, it shouldn't be hard. But it's not
> necessarily a one-liner either. It depends on what are the pratical
> considerations for having that specific kind of allocation tied to a
> memcg. The slab, for instance, that follows this series, is far away
> from a one-liner: it is in fact, a 19-patch patch series.
>
>

>
> >
> > And how *accurate* is the proposed code? What percentage of kernel
> > memory allocations are unaccounted, typical case and worst case?
>
> With both patchsets applied, all memory used for the stack and most of
> the memory used for slab objects allocated in userspace process contexts
> are accounted.
>
> I honestly don't know which percentage of the total kernel memory this
> represents.

It sounds like the coverage will be good. What's left over? Random
get_free_pages() calls and interrupt-time slab allocations?

I suppose that there are situations in which network rx could consume
significant amounts of unaccounted memory?

> The accuracy for stack pages is very high: In this series, we don't move
> stack pages around when moving a task to other cgroups (for stack, it
> could be done), but other than that, all processes that pops up in a
> cgroup and stay there will have its memory accurately accounted.
>
> The slab is more complicated, and depends on the workload. It will be
> more accurate in workloads in which the level of object-sharing among
> cgroups is low. A container, for instance, is the perfect example of
> where this happen.
>
> >
> > All sorts of questions come to mind over this decision, but it was
> > unexplained. It should be, please. A lot!
> >
> >>
> >> ...
> >>
> >> Limits lower than
> >> the user limit effectively means there is a separate kernel memory limit that
> >> may be reached independently than the user limit. Values equal or greater than
> >> the user limit implies only that kernel memory is tracked. This provides a
> >> unified vision of "maximum memory", be it kernel or user memory.
> >>
> >
> > I'm struggling to understand that text much at all. Reading the
> > Documentation/cgroups/memory.txt patch helped.
> >
>
> Great. If you have any specific suggestions I can change that. Maybe I
> should just paste the documentation bit in here...

That's not a bad idea.

Subject: Re: [PATCH v5 04/14] kmem accounting basic infrastructure
Posted by Tejun Heo on Thu, 18 Oct 2012 19:37:17 GMT
View Forum Message <> Reply to Message

Hello, David.

On Wed, Oct 17, 2012 at 03:08:04PM -0700, David Rientjes wrote:
> > +static int memcg_update_kmem_limit(struct cgroup *cont, u64 val)
> > +{
> > + int ret = -EINVAL;
> > +#ifdef CONFIG_MEMCG_KMEM
> > + struct mem_cgroup *memcg = mem_cgroup_from_cont(cont);
> > + /*
> > + * For simplicity, we won't allow this to be disabled. It also can't
> > + * be changed if the cgroup has children already, or if tasks had
> > + * already joined.
> > + *
> > + * If tasks join before we set the limit, a person looking at
> > + * kmem.usage_in_bytes will have no way to determine when it took
> > + * place, which makes the value quite meaningless.
> > + *
> > + * After it first became limited, changes in the value of the limit are
> > + * of course permitted.
> > + *
> > + * Taking the cgroup_lock is really offensive, but it is so far the only
> > + * way to guarantee that no children will appear. There are plenty of
> > + * other offenders, and they should all go away. Fine grained locking
> > + * is probably the way to go here. When we are fully hierarchical, we
> > + * can also get rid of the use_hierarchy check.
> >
> Not sure it's so offensive, it's a pretty standard way of ensuring that
> cont->children doesn't get manipulated in a race.

cgroup_lock is inherently one of the outermost locks as it protects
cgroup hierarchy and modifying cgroup hierarchy involves invoking
subsystem callbacks which may grab subsystem locks. Grabbing it
directly from subsystems thus creates high likelihood of creating a
dependency loop and it's nasty to break.

And I'm unsure whether making cgroup locks finer grained would help as
cpuset grabs cgroup_lock to perform actual task migration which would
require the outermost cgroup locking anyway. This one already has
showed up in a couple lockdep warnings involving static_key usages.

A couple days ago, I posted a patchset to remove cgroup_lock usage
from cgroup_freezer and at least cgroup_freezer seems like it was
aiming for the wrong behavior which led to the wrong locking behavior
requiring grabbing cgroup_lock. I can't say whether others will be
that easy tho.

Anyways, so, cgroup_lock is in the process of being unexported and I'd
really like another usage isn't added but maybe that requires larger
changes to memcg and not something which can be achieved here. Dunno.
Will think more about it.

Thanks.

--
tejun

Subject: Re: [PATCH v5 04/14] kmem accounting basic infrastructure
Posted by Tejun Heo on Thu, 18 Oct 2012 19:47:27 GMT
View Forum Message <> Reply to Message

Hey, Glauber.

On Thu, Oct 18, 2012 at 09:01:23PM +0400, Glauber Costa wrote:
> That is the offensive part. But it is also how things are done in memcg
> right now, and there is nothing fundamentally different in this one.
> Whatever lands in the remaining offenders, can land in here.

I think the problem here is that we don't have "you're committing to
creation of a new cgroup" callback and thus subsystem can't
synchronize locally against cgroup creation. For task migration
->attach() does that but cgroup creation may fail after ->create()
succeeded so that doesn't work.

We'll probably need to add ->post_create() which is invoked after
creation is complete. Li?

Thanks.

--
tejun

Subject: Re: [PATCH v5 07/14] mm: Allocate kernel pages to the right memcg
Posted by akpm on Thu, 18 Oct 2012 20:44:57 GMT
View Forum Message <> Reply to Message

On Thu, 18 Oct 2012 13:24:47 +0400
Glauber Costa <glommer@parallels.com> wrote:

> On 10/18/2012 02:12 AM, Andrew Morton wrote:
> > On Tue, 16 Oct 2012 14:16:44 +0400
> > Glauber Costa <glommer@parallels.com> wrote:
> >
> >> When a process tries to allocate a page with the __GFP_KMEMCG flag, the
> >> page allocator will call the corresponding memcg functions to validate
> >> the allocation. Tasks in the root memcg can always proceed.
> >>
> >> To avoid adding markers to the page - and a kmem flag that would
> >> necessarily follow, as much as doing page_cgroup lookups for no reason,
> >> whoever is marking its allocations with __GFP_KMEMCG flag is responsible
> >> for telling the page allocator that this is such an allocation at
> >> free_pages() time.
> >
> > Well, why? Was that the correct decision?
> >
>
> I don't fully understand your question. Is this the same question you
> posed in patch 0, about marking some versus marking all? If so, I
> believe I should have answered it there.

Yes, it's the same question. The one which has not yet been fully answered ;)

Subject: Re: [PATCH v5 06/14] memcg: kmem controller infrastructure
Posted by David Rientjes on Thu, 18 Oct 2012 21:59:44 GMT
View Forum Message <> Reply to Message

On Thu, 18 Oct 2012, Glauber Costa wrote:

> >> @@ -2630,6 +2634,171 @@ static void __mem_cgroup_commit_charge(struct
mem_cgroup *memcg,
> >>   memcg_check_events(memcg, page);
> >> }
> >>
> >> +#ifdef CONFIG_MEMCG_KMEM
> >> +static inline bool memcg_can_account_kmem(struct mem_cgroup *memcg)
> >> +{
> >> + return !mem_cgroup_disabled() && !mem_cgroup_is_root(memcg) &&
> >> + (memcg->kmem_accounted & KMEM_ACCOUNTED_MASK);
> >> +}
> >> +
> >> +static int memcg_charge_kmem(struct mem_cgroup *memcg, gfp_t gfp, u64 size)
> >> +{
> >> + struct res_counter *fail_res;

> >> + struct mem_cgroup *_memcg;
> >> + int ret = 0;
> >> + bool may_oom;
> >> +
> >> + ret = res_counter_charge(&memcg->kmem, size, &fail_res);
> >> + if (ret)
> >> + return ret;
> >> +
> >> + /*
> >> + * Conditions under which we can wait for the oom_killer.
> >> + * We have to be able to wait, but also, if we can't retry,
> >> + * we obviously shouldn't go mess with oom.
> >> + */
> >> + may_oom = (gfp & __GFP_WAIT) && !(gfp & __GFP_NORETRY);
> >
> > What about gfp & __GFP_FS?
> >
>
> Do you intend to prevent or allow OOM under that flag? I personally
> think that anything that accepts to be OOM-killed should have GFP_WAIT
> set, so that ought to be enough.
>

The oom killer in the page allocator cannot trigger without __GFP_FS
because direct reclaim has little chance of being very successful and
thus we end up needlessly killing processes, and that tends to happen
quite a bit if we dont check for it. Seems like this would also happen
with memcg if mem_cgroup_reclaim() has a large probability of failing?

Subject: Re: [PATCH v5 06/14] memcg: kmem controller infrastructure
Posted by David Rientjes on Thu, 18 Oct 2012 22:06:54 GMT
View Forum Message <> Reply to Message

On Thu, 18 Oct 2012, Glauber Costa wrote:

> > Do we actually need to test PF_KTHREAD when current->mm == NULL?
> > Perhaps because of aio threads whcih temporarily adopt a userspace mm?
>
> I believe so. I remember I discussed this in the past with David
> Rientjes and he advised me to test for both.
>

PF_KTHREAD can do use_mm() to assume an ->mm but hopefully they aren't
allocating slab while doing so. Have you considered actually charging
current->mm->owner for that memory, though, since the kthread will have
freed the memory before unuse_mm() or otherwise have charged it on behalf
of a user process, i.e. only exempting PF_KTHREAD?

Subject: Re: [PATCH v5 06/14] memcg: kmem controller infrastructure
Posted by Glauber Costa on Fri, 19 Oct 2012 09:10:27 GMT
View Forum Message <> Reply to Message

On 10/19/2012 02:06 AM, David Rientjes wrote:
> On Thu, 18 Oct 2012, Glauber Costa wrote:
>
>>> Do we actually need to test PF_KTHREAD when current->mm == NULL?
>>> Perhaps because of aio threads whcih temporarily adopt a userspace mm?
>>
>> I believe so. I remember I discussed this in the past with David
>> Rientjes and he advised me to test for both.
>>
>
> PF_KTHREAD can do use_mm() to assume an ->mm but hopefully they aren't
> allocating slab while doing so. Have you considered actually charging
> current->mm->owner for that memory, though, since the kthread will have
> freed the memory before unuse_mm() or otherwise have charged it on behalf
> of a user process, i.e. only exempting PF_KTHREAD?
>
I always charge current->mm->owner.

Subject: Re: [PATCH v5 06/14] memcg: kmem controller infrastructure
Posted by David Rientjes on Fri, 19 Oct 2012 09:31:33 GMT
View Forum Message <> Reply to Message

On Fri, 19 Oct 2012, Glauber Costa wrote:

> >>> Do we actually need to test PF_KTHREAD when current->mm == NULL?
> >>> Perhaps because of aio threads whcih temporarily adopt a userspace mm?
> >>
> >> I believe so. I remember I discussed this in the past with David
> >> Rientjes and he advised me to test for both.
> >>
> >
> > PF_KTHREAD can do use_mm() to assume an ->mm but hopefully they aren't
> > allocating slab while doing so. Have you considered actually charging
> > current->mm->owner for that memory, though, since the kthread will have
> > freed the memory before unuse_mm() or otherwise have charged it on behalf
> > of a user process, i.e. only exempting PF_KTHREAD?
> >
> I always charge current->mm->owner.
>

Yeah, I'm asking have you considered charging current->mm->owner for the
memory when a kthread (current) assumes the mm of a user process via
use_mm()? It may free the memory before calling unuse_mm(), but it's also

allocating the memory on behalf of a user so this exemption might be
dangerous if use_mm() becomes more popular. I don't think there's
anything that prevents that charge, I'm just wondering if you considered
doing it even for kthreads with an mm.

Subject: Re: [PATCH v5 00/14] kmem controller for memcg.
Posted by Glauber Costa on Fri, 19 Oct 2012 09:55:05 GMT
View Forum Message <> Reply to Message

On 10/18/2012 11:21 PM, Andrew Morton wrote:
> On Thu, 18 Oct 2012 20:51:05 +0400
> Glauber Costa <glommer@parallels.com> wrote:
>
>> On 10/18/2012 02:11 AM, Andrew Morton wrote:
>>> On Tue, 16 Oct 2012 14:16:37 +0400
>>> Glauber Costa <glommer@parallels.com> wrote:
>>>
>>>> ...
>>>>
>>>> A general explanation of what this is all about follows:
>>>>
>>>> The kernel memory limitation mechanism for memcg concerns itself with
>>>> disallowing potentially non-reclaimable allocations to happen in exaggerate
>>>> quantities by a particular set of processes (cgroup). Those allocations could
>>>> create pressure that affects the behavior of a different and unrelated set of
>>>> processes.
>>>>
>>>> Its basic working mechanism is to annotate some allocations with the
>>>> _GFP_KMEMCG flag. When this flag is set, the current process allocating will
>>>> have its memcg identified and charged against. When reaching a specific limit,
>>>> further allocations will be denied.
>>>
>>> The need to set _GFP_KMEMCG is rather unpleasing, and makes one wonder
>>> "why didn't it just track all allocations".
>>>
>> This was raised as well by Peter Zijlstra during the memcg summit.
>
> Firstly: please treat any question from a reviewer as an indication
> that information was missing from the changelog or from code comments.
> Ideally all such queries are addressed in later version of the patch
> and changelog.
>
This is in no opposition with me telling a bit that this has been raised
before! =)

>> The
>> answer I gave to him still stands: There is a cost associated with it.

>> We believe it comes down to a trade off situation. How much tracking a
>> particular kind of allocation help vs how much does it cost.
>>
>> The free path is specially more expensive, since it will always incur in
>> a page_cgroup lookup.
>
> OK. But that is a quantitative argument, without any quantities! Do
> we have even an estimate of what this cost will be? Perhaps it's the
> case that, if well implemented, that cost will be acceptable. How do
> we tell?
>

There are two ways:
1) Measuring on various workloads. The workload I measured particularly in here (link in the beginning of this e-mail), showed a 2 - 3 % penalty with the whole thing applied. Truth be told, this was mostly pin-pointed to the slab part, which gets most of its cost from a relay function, and not from the page allocation per-se. But for me, this is enough to tell that there is a cost high enough to bother some.

2) We can infer from past behavior of memcg. It always shown itself as quite an expensive beast. Making it suck faster is a completely separate endeavor. It seems only natural to me to reduce its reach even without specific number for each of the to-be-tracked candidates.

Moreover, there is the cost question, but cost is not *the only* question, as I underlined a few paragraphs below. It is not always obvious how to pinpoint a kernel page to a specific process, so this need to be analyzed on a case-by-case basis. The slab is the hardest one, and it is done. But even then...

If this is still not good enough, and you would like me to measure something else, just let me know.

>>> Does this mean that over time we can expect more sites to get the
>>> _GFP_KMEMCG tagging?
>>
>> We have being doing kernel memory limitation for OpenVZ for a lot of
>> times, using a quite different mechanism. What we do in this work (with
>> slab included), allows us to achieve feature parity with that. It means
>> it is good enough for production environments.
>
> That's really good info.
>
>> Whether or not more people will want other allocations to be tracked, I
>> can't predict. What I do can say is that stack + slab is a very
>> significant part of the memory one potentially cares about, and if
>> anyone else ever have the need for more, it will come down to a

>> trade-off calculation.
>
> OK.
>
>>> If so, are there any special implications, or do
>>> we just go in, do the one-line patch and expect everything to work?
>>
>> With the infrastructure in place, it shouldn't be hard. But it's not
>> necessarily a one-liner either. It depends on what are the pratical
>> considerations for having that specific kind of allocation tied to a
>> memcg. The slab, for instance, that follows this series, is far away
>> from a one-liner: it is in fact, a 19-patch patch series.
>>
>>
>>
>>>
>>> And how *accurate* is the proposed code? What percentage of kernel
>>> memory allocations are unaccounted, typical case and worst case?
>>
>> With both patchsets applied, all memory used for the stack and most of
>> the memory used for slab objects allocated in userspace process contexts
>> are accounted.
>>
>> I honestly don't know which percentage of the total kernel memory this
>> represents.
>
> It sounds like the coverage will be good. What's left over? Random
> get_free_pages() calls and interrupt-time slab allocations?
>

random get_free_pages, vmalloc, ptes. interrupt is left out on purpose,
because we can't cgroup-track something that doesn't have a process context.

> I suppose that there are situations in which network rx could consume
> significant amounts of unaccounted memory?
>

Not unaccounted. This is merged already =)

Subject: Re: [PATCH v5 06/14] memcg: kmem controller infrastructure
Posted by Glauber Costa on Fri, 19 Oct 2012 10:00:23 GMT
View Forum Message <> Reply to Message

On 10/19/2012 01:31 PM, David Rientjes wrote:
> On Fri, 19 Oct 2012, Glauber Costa wrote:
>

>>>>> Do we actually need to test PF_KTHREAD when current->mm == NULL?
>>>>> Perhaps because of aio threads whcih temporarily adopt a userspace mm?
>>>>
>>>> I believe so. I remember I discussed this in the past with David
>>>> Rientjes and he advised me to test for both.
>>>>
>>>
>>> PF_KTHREAD can do use_mm() to assume an ->mm but hopefully they aren't
>>> allocating slab while doing so. Have you considered actually charging
>>> current->mm->owner for that memory, though, since the kthread will have
>>> freed the memory before unuse_mm() or otherwise have charged it on behalf
>>> of a user process, i.e. only exempting PF_KTHREAD?
>>>
>> I always charge current->mm->owner.
>>
>
> Yeah, I'm asking have you considered charging current->mm->owner for the
> memory when a kthread (current) assumes the mm of a user process via
> use_mm()? It may free the memory before calling unuse_mm(), but it's also
> allocating the memory on behalf of a user so this exemption might be
> dangerous if use_mm() becomes more popular. I don't think there's
> anything that prevents that charge, I'm just wondering if you considered
> doing it even for kthreads with an mm.
>
Well, I thought about it.

And I personally don't like it. I think all kthreads should be treated
the same. We have control over it, unlike any userspace application. We
never expect its memory consumption to explode.

Specially considering that those allocations are supposed to be
short-lived, we are only paying the res_counters count for no reason.

Subject: Re: [PATCH v5 06/14] memcg: kmem controller infrastructure
Posted by Glauber Costa on Fri, 19 Oct 2012 10:08:27 GMT
View Forum Message <> Reply to Message

On 10/19/2012 01:59 AM, David Rientjes wrote:
> On Thu, 18 Oct 2012, Glauber Costa wrote:
>
>>>> @@ -2630,6 +2634,171 @@ static void __mem_cgroup_commit_charge(struct
mem_cgroup *memcg,
>>>>   memcg_check_events(memcg, page);
>>>> }
>>>>
>>>> +#ifdef CONFIG_MEMCG_KMEM
>>>> +static inline bool memcg_can_account_kmem(struct mem_cgroup *memcg)

>>>> +{
>>>> + return !mem_cgroup_disabled() && !mem_cgroup_is_root(memcg) &&
>>>> + (memcg->kmem_accounted & KMEM_ACCOUNTED_MASK);
>>>> +}
>>>> +
>>>> +static int memcg_charge_kmem(struct mem_cgroup *memcg, gfp_t gfp, u64 size)
>>>> +{
>>>> + struct res_counter *fail_res;
>>>> + struct mem_cgroup *_memcg;
>>>> + int ret = 0;
>>>> + bool may_oom;
>>>> +
>>>> + ret = res_counter_charge(&memcg->kmem, size, &fail_res);
>>>> + if (ret)
>>>> + return ret;
>>>> +
>>>> + /*
>>>> + * Conditions under which we can wait for the oom_killer.
>>>> + * We have to be able to wait, but also, if we can't retry,
>>>> + * we obviously shouldn't go mess with oom.
>>>> + */
>>>> + may_oom = (gfp & __GFP_WAIT) && !(gfp & __GFP_NORETRY);
>>>
>>> What about gfp & __GFP_FS?
>>>
>>
>> Do you intend to prevent or allow OOM under that flag? I personally
>> think that anything that accepts to be OOM-killed should have GFP_WAIT
>> set, so that ought to be enough.
>>
>
> The oom killer in the page allocator cannot trigger without __GFP_FS
> because direct reclaim has little chance of being very successful and
> thus we end up needlessly killing processes, and that tends to happen
> quite a bit if we dont check for it. Seems like this would also happen
> with memcg if mem_cgroup_reclaim() has a large probability of failing?
>

I can indeed see tests for GFP_FS in some key locations in mm/ before
calling the OOM Killer.

Should I test for GFP_IO as well? If the idea is preventing OOM to
trigger for allocations that can write their pages back, how would you
feel about the following test:

may_oom = (gfp & GFP_KERNEL) && !(gfp & __GFP_NORETRY) ?

Michal, what is your take in here?

Subject: Re: [PATCH v5 06/14] memcg: kmem controller infrastructure
Posted by David Rientjes on Fri, 19 Oct 2012 20:34:04 GMT
View Forum Message <> Reply to Message

On Fri, 19 Oct 2012, Glauber Costa wrote:

> >>> What about gfp & __GFP_FS?
> >>>
> >>
> >> Do you intend to prevent or allow OOM under that flag? I personally
> >> think that anything that accepts to be OOM-killed should have GFP_WAIT
> >> set, so that ought to be enough.
> >>
> >
> > The oom killer in the page allocator cannot trigger without __GFP_FS
> > because direct reclaim has little chance of being very successful and
> > thus we end up needlessly killing processes, and that tends to happen
> > quite a bit if we dont check for it. Seems like this would also happen
> > with memcg if mem_cgroup_reclaim() has a large probability of failing?
> >
>
> I can indeed see tests for GFP_FS in some key locations in mm/ before
> calling the OOM Killer.
>
> Should I test for GFP_IO as well?

It's not really necessary, if __GFP_IO isn't set then it wouldn't make
sense for __GFP_FS to be set.

> If the idea is preventing OOM to
> trigger for allocations that can write their pages back, how would you
> feel about the following test:
>
> may_oom = (gfp & GFP_KERNEL) && !(gfp & __GFP_NORETRY) ?
>

I would simply copy the logic from the page allocator and only trigger oom
for __GFP_FS and !__GFP_NORETRY.

Subject: Re: [PATCH v5 06/14] memcg: kmem controller infrastructure
Posted by Glauber Costa on Mon, 22 Oct 2012 12:34:15 GMT
View Forum Message <> Reply to Message

On 10/20/2012 12:34 AM, David Rientjes wrote:
> On Fri, 19 Oct 2012, Glauber Costa wrote:
>
>>>>> What about gfp & __GFP_FS?

>>>>>
>>>>
>>>> Do you intend to prevent or allow OOM under that flag? I personally
>>>> think that anything that accepts to be OOM-killed should have GFP_WAIT
>>>> set, so that ought to be enough.
>>>>
>>>
>>> The oom killer in the page allocator cannot trigger without __GFP_FS
>>> because direct reclaim has little chance of being very successful and
>>> thus we end up needlessly killing processes, and that tends to happen
>>> quite a bit if we dont check for it. Seems like this would also happen
>>> with memcg if mem_cgroup_reclaim() has a large probability of failing?
>>>
>>
>> I can indeed see tests for GFP_FS in some key locations in mm/ before
>> calling the OOM Killer.
>>
>> Should I test for GFP_IO as well?
>
> It's not really necessary, if __GFP_IO isn't set then it wouldn't make
> sense for __GFP_FS to be set.
>
>> If the idea is preventing OOM to
>> trigger for allocations that can write their pages back, how would you
>> feel about the following test:
>>
>> may_oom = (gfp & GFP_KERNEL) && !(gfp & __GFP_NORETRY) ?
>>
>
> I would simply copy the logic from the page allocator and only trigger oom
> for __GFP_FS and !__GFP_NORETRY.
>

That seems reasonable to me. Michal ?

Subject: Re: [PATCH v5 06/14] memcg: kmem controller infrastructure
Posted by Michal Hocko on Mon, 22 Oct 2012 12:51:24 GMT
View Forum Message <> Reply to Message

[Sorry for the late reply]

On Mon 22-10-12 16:34:15, Glauber Costa wrote:
> On 10/20/2012 12:34 AM, David Rientjes wrote:
> > On Fri, 19 Oct 2012, Glauber Costa wrote:
> >
> > >>>>> What about gfp & __GFP_FS?
> >>>>>

> >>>>
> >>>> Do you intend to prevent or allow OOM under that flag? I personally
> >>>> think that anything that accepts to be OOM-killed should have GFP_WAIT
> >>>> set, so that ought to be enough.
> >>>>
> >>>
> >>> The oom killer in the page allocator cannot trigger without __GFP_FS
> >>> because direct reclaim has little chance of being very successful and
> >>> thus we end up needlessly killing processes, and that tends to happen
> >>> quite a bit if we dont check for it. Seems like this would also happen
> >>> with memcg if mem_cgroup_reclaim() has a large probability of failing?
> >>>
> >>
> >> I can indeed see tests for GFP_FS in some key locations in mm/ before
> >> calling the OOM Killer.
> >>
> >> Should I test for GFP_IO as well?
> >
> > It's not really necessary, if __GFP_IO isn't set then it wouldn't make
> > sense for __GFP_FS to be set.
> >
> >> If the idea is preventing OOM to
> >> trigger for allocations that can write their pages back, how would you
> >> feel about the following test:
> >>
> >> may_oom = (gfp & GFP_KERNEL) && !(gfp & __GFP_NORETRY) ?
> >>
> >
> > I would simply copy the logic from the page allocator and only trigger oom
> > for __GFP_FS and !__GFP_NORETRY.
> >
>
> That seems reasonable to me. Michal ?

Yes it makes sense to be consistent with the global case. While we are
at it, do we need to consider PF_DUMPCORE resp. !__GFP_NOFAIL?
--
Michal Hocko
SUSE Labs

Subject: Re: [PATCH v5 06/14] memcg: kmem controller infrastructure
Posted by Glauber Costa on Mon, 22 Oct 2012 12:52:41 GMT
View Forum Message <> Reply to Message

On 10/22/2012 04:51 PM, Michal Hocko wrote:
> [Sorry for the late reply]
>

> On Mon 22-10-12 16:34:15, Glauber Costa wrote:
>> On 10/20/2012 12:34 AM, David Rientjes wrote:
>>> On Fri, 19 Oct 2012, Glauber Costa wrote:
>>>
>>>>>> What about gfp & __GFP_FS?
>>>>>>
>>>>>> Do you intend to prevent or allow OOM under that flag? I personally
>>>>>> think that anything that accepts to be OOM-killed should have GFP_WAIT
>>>>>> set, so that ought to be enough.
>>>>>>
>>>>> The oom killer in the page allocator cannot trigger without __GFP_FS
>>>>> because direct reclaim has little chance of being very successful and
>>>>> thus we end up needlessly killing processes, and that tends to happen
>>>>> quite a bit if we dont check for it. Seems like this would also happen
>>>>> with memcg if mem_cgroup_reclaim() has a large probability of failing?
>>>>>
>>>> I can indeed see tests for GFP_FS in some key locations in mm/ before
>>>> calling the OOM Killer.
>>>>
>>>> Should I test for GFP_IO as well?
>>>
>>> It's not really necessary, if __GFP_IO isn't set then it wouldn't make
>>> sense for __GFP_FS to be set.
>>>
>>>> If the idea is preventing OOM to
>>>> trigger for allocations that can write their pages back, how would you
>>>> feel about the following test:
>>>>
>>>> may_oom = (gfp & GFP_KERNEL) && !(gfp & __GFP_NORETRY) ?
>>>>
>>>
>>> I would simply copy the logic from the page allocator and only trigger oom
>>> for __GFP_FS and !__GFP_NORETRY.
>>>
>>
>> That seems reasonable to me. Michal ?
>
> Yes it makes sense to be consistent with the global case. While we are
> at it, do we need to consider PF_DUMPCORE resp. !__GFP_NOFAIL?
>
at least from kmem, GFP_NOFAIL will not reach this codepath. We will
ditch it to the root in memcontrol.h